Device Abstraction Layer

Draft

86 Pages

Abstract

Defines a new device abstraction API in OSGi platform. It provides a simple access to the devices and their functionality.

0 Document Information

0.1 License

DISTRIBUTION AND FEEDBACK LICENSE, Version 2.0

The OSGi Alliance hereby grants you a limited copyright license to copy and display this document (the “Distribution”) in any medium without fee or royalty. This Distribution license is exclusively for the purpose of reviewing and providing feedback to the OSGi Alliance. You agree not to modify the Distribution in any way and further agree to not participate in any way in the making of derivative works thereof, other than as a necessary result of reviewing and providing feedback to the Distribution. You also agree to cause this notice, along with the accompanying consent, to be included on all copies (or portions thereof) of the Distribution. The OSGi Alliance also grants you a perpetual, non-exclusive, worldwide, fully paid-up, royalty free, limited license (without the right to sublicense) under any applicable copyrights, to create and/or distribute an implementation of the Distribution that: (i) fully implements the Distribution including all its required interfaces and functionality; (ii) does not modify, subset, superset or otherwise extend the OSGi Name Space, or include any public or protected packages, classes, Java interfaces, fields or methods within the OSGi Name Space other than those required and authorized by the Distribution. An implementation that does not satisfy limitations (i)-(ii) is not considered an implementation of the Distribution, does not receive the benefits of this license, and must not be described as an implementation of the Distribution. "OSGi Name Space" shall mean the public class or interface declarations whose names begin with "org.osgi" or any recognized successors or replacements thereof. The OSGi Alliance expressly reserves all rights not granted pursuant to these limited copyright licenses including termination of the license at will at any time.

EXCEPT FOR THE LIMITED COPYRIGHT LICENSES GRANTED ABOVE, THE OSGi ALLIANCE DOES NOT GRANT, EITHER EXPRESSLY OR IMPLIEDLY, A LICENSE TO ANY INTELLECTUAL PROPERTY IT, OR ANY THIRD PARTIES, OWN OR CONTROL. Title to the copyright in the Distribution will at all times remain with the OSGi Alliance. The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted therein are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, place, or event is intended or should be inferred.

THE DISTRIBUTION IS PROVIDED "AS IS," AND THE OSGi ALLIANCE (INCLUDING ANY THIRD PARTIES THAT HAVE CONTRIBUTED TO THE DISTRIBUTION) MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, OR TITLE; THAT THE CONTENTS OF THE DISTRIBUTION ARE SUITABLE FOR ANY PURPOSE; NOR THAT THE IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.
NEITHER THE OSGi ALLIANCE NOR ANY THIRD PARTY WILL BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF OR RELATING TO ANY USE OR DISTRIBUTION OF THE DISTRIBUTION.

Implementation of certain elements of this Distribution may be subject to third party intellectual property rights, including without limitation, patent rights (such a third party may or may not be a member of the OSGi Alliance). The OSGi Alliance is not responsible and shall not be held responsible in any manner for identifying or failing to identify any or all such third party intellectual property rights.

The Distribution is a draft. As a result, the final product may change substantially by the time of final publication, and you are cautioned against relying on the content of this Distribution. You are encouraged to update any implementation of the Distribution if and when such Distribution becomes a final specification.

The OSGi Alliance is willing to receive input, suggestions and other feedback (“Feedback”) on the Distribution. By providing such Feedback to the OSGi Alliance, you grant to the OSGi Alliance and all its Members a non-exclusive, non-transferable,

worldwide, perpetual, irrevocable, royalty-free copyright license to copy, publish, license, modify, sublicense or otherwise distribute and exploit your Feedback for any purpose. Likewise, if incorporation of your Feedback would cause an implementation of the Distribution, including as it may be modified, amended, or published at any point in the future ("Future Specification"), to necessarily infringe a patent or patent application that you own or control, you hereby commit to grant to all implementers of such Distribution or Future Specification an irrevocable, worldwide, sublicenseable, royalty free license under such patent or patent application to make, have made, use, sell, offer for sale, import and export products or services that implement such Distribution or Future Specification. You warrant that (a) to the best of your knowledge you have the right to provide this Feedback, and if you are providing Feedback on behalf of a company, you have the rights to provide Feedback on behalf of your company; (b) the Feedback is not confidential to you and does not violate the copyright or trade secret interests of another; and (c) to the best of your knowledge, use of the Feedback would not cause an implementation of the Distribution or a Future Specification to necessarily infringe any third-party patent or patent application known to you. You also acknowledge that the OSGi Alliance is not required to incorporate your Feedback into any version of the Distribution or a Future Specification.

I HEREBY ACKNOWLEDGE AND AGREE TO THE TERMS AND CONDITIONS DELINEATED ABOVE.

0.2 Trademarks

OSGi™ is a trademark, registered trademark, or service mark of the OSGi Alliance in the US and other countries. Java is a trademark, registered trademark, or service mark of Oracle Corporation in the US and other countries. All other trademarks, registered trademarks, or service marks used in this document are the property of their respective owners and are hereby recognized.

0.3 Feedback

This document can be downloaded from the OSGi Alliance design repository at https://github.com/osgi/design The public can provide feedback about this document by opening a bug at https://www.osgi.org/bugzilla/.

0.4 Table of Contents

0.5 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.`

0.6 Revision History

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

Revision	Date	Comments
Initial	Jan 22 2013	Initial draft version. Evgeni Grigorov, ProSyst Software, e.grigorov@prosyst.com
2nd draft	Feb 13 2013	Updated Considered Alternatives and Security Considerations after F2F meeting in Austin, TX. Provide more details about device management. Evgeni Grigorov, ProSyst Software, e.grigorov@prosyst.com
3rd draft	Mar 08 2013	Remove DeviceAdmin service. Describe DeviceFunction and FunctionalDevice interfaces. Evgeni Grigorov, ProSyst Software, e.grigorov@prosyst.com
4th draft	Apr 08 2013	Rename the package and some constants. Merge the AbstractDevice and FunctionalDevice to FunctionalDevice. Add Functional Device Permission. Add Device Function Event. Minor fixes: renamed Device Access category, fixed unit representation and some clarifications. Add a suggestion about Device Functions to be discussed on F2F in Cologne. Evgeni Grigorov, ProSyst Software, e.grigorov@prosyst.com
5th draft	Jun 12 2013	Add a basic set of Device Functions. Include the device status transitions. Update the illustrations. Add a status detail mapping. Add some snippets. Remove the device helper methods for an access to parent, children and reference devices. Add a Functional Device and Device Function descriptions. Add error codes to DeviceFunctionException. Update the javadoc. Evgeni Grigorov, ProSyst Software, e.grigorov@prosyst.com

Revision	Date	Comments
6th draft	Jul 02 2013	Describe the status transitions in detail. FunctionalDeviceException.CODE_UNKNOW fixed to CODE_UNKNOWN. Functional Group is introduced. Functional Device, Functional Group and Device Function are in the service registry. New service properties are introduced. Parent-child relation is removed. Add more details to the descriptions. Evgeni Grigorov, ProSyst Software, e.grigorov@prosyst.com

1 Introduction

OSGi is gaining popularity as enabling technology for building embedded system in residential and M2M markets. In these contexts it is often necessary to communicate with IP and non-IP devices by using various protocols such as ZigBee, Z-Wave, KNX, UPnP etc. In order to provide a convenient programming model suitable for the realization of end-to-end services it is very useful to define and apply an abstraction layer which unifies the work with devices supporting different protocols.

This RFC defines a new device abstraction API in OSGi.

2 Application Domain

Currently there are several standardization bodies such as OSGi Alliance, HGI, BBF, ETSI M2M which deal with the deployment of services in an infrastructure based on the usage of a Residential Gateway running OSGi as Execution Platform. The picture on Illustration 1 shows a reference architecture which is valid in the majority of cases under consideration.

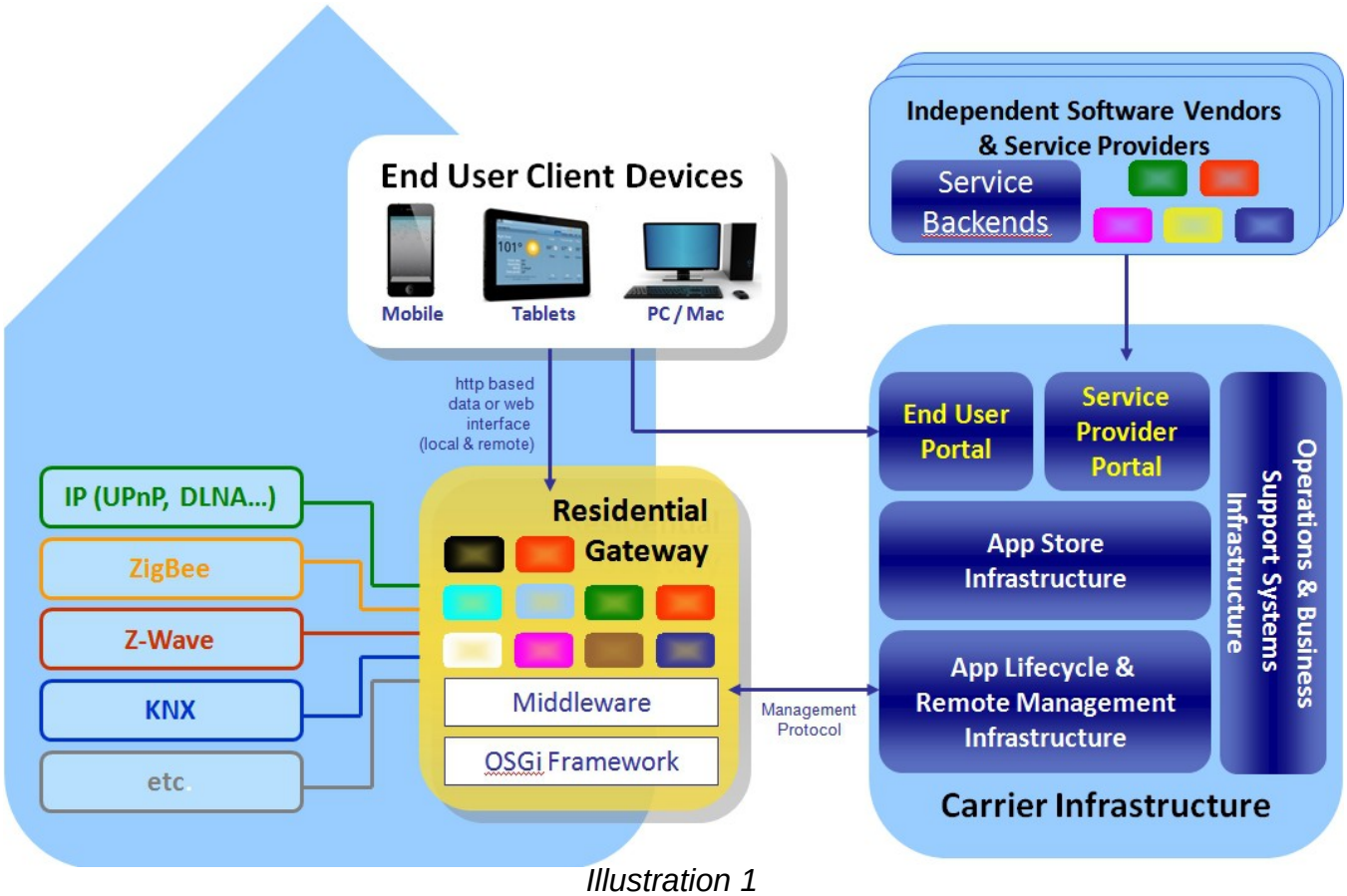

Illustration 1

In this architecture the application logic is distributed between:

- Applications running on the residential gateways
- Applications running in the cloud, e.g. on the service provider's backend
- Applications on the devices providing UI (e.g. tablets, mobile phones, desktops).

In order to realize services which access other IP and non-IP devices connected to the residential gateway, those applications must be able to read information from the devices and perform operations on them through software APIs. Such an access is essential for services in the area of smart metering, entertainment, home automation, assisted living and security.

The existing OSGi specifications which address related topics are:

- Device Access Specification – focuses on the dynamic discovery of the proper driver when a new device is attached/connected to the residential gateway. The device access is limited to attend the driver installation needs.
- UPnP™ Device Service Specification – defines among the other OSGi API for work with UPnP devices accessible from the residential gateway. API is specified in the scope of UPnP Device Access category.

3 Problem Description

Normally the residential gateways operate in heterogeneous environment including devices that support different protocols. It's not trivial to provide interoperability of the applications and the devices under such circumstances. The existing OSGi Device Access Specification solves the driver installation problems but currently there is no complete API that can be used for accessing the device data and for invoking actions on the devices.

Illustration 2 shows one possible approach for working with heterogeneous devices in an OSGi environment:

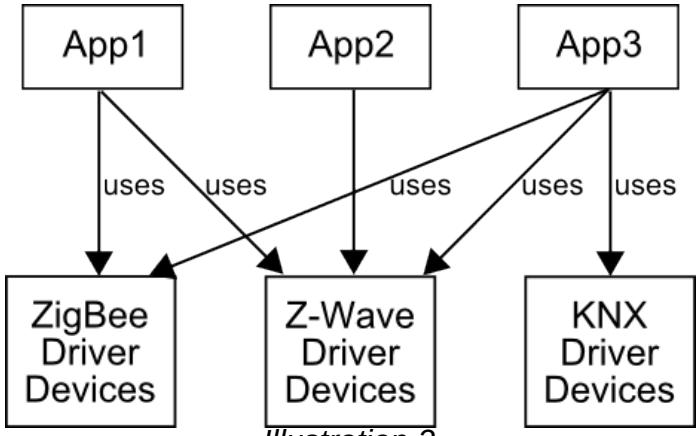

Illustration 2

In this case each application which accesses devices of a given type must use API specific for this type. One obvious disadvantage of this model is that when a new device protocol is added the applications must be modified in order to support this protocol.

Much better is the approach from Illustration 3 which is defined by this RFC.

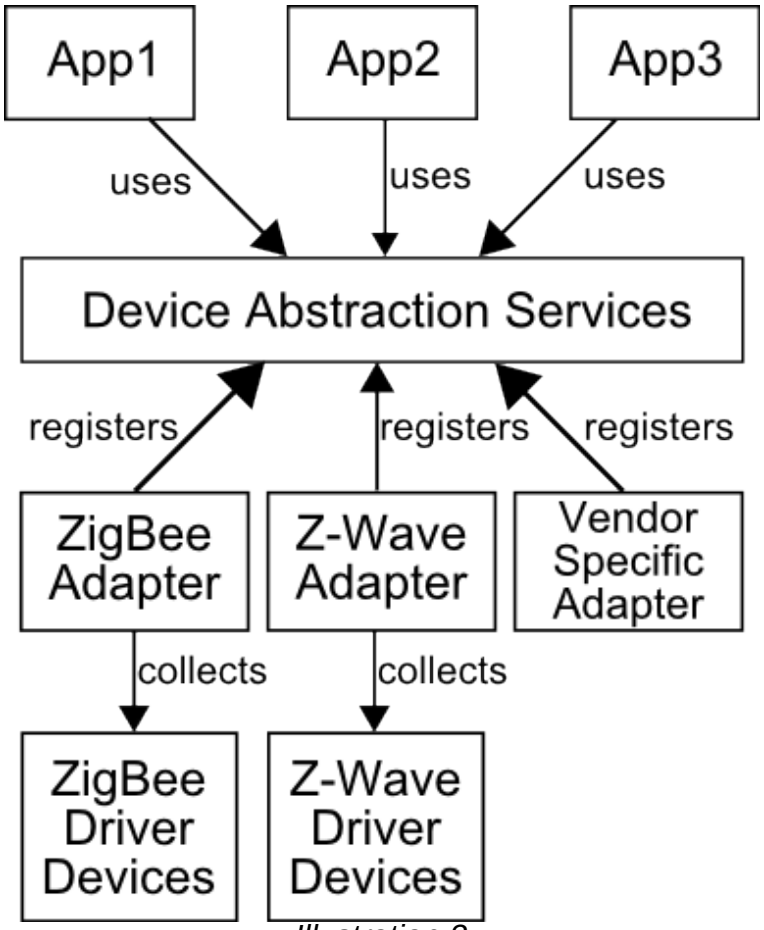

Illustration 3

In this case an additional device abstraction layer is introduced which unifies the work with the devices provided by the different underlying protocols. Thus the following advantages are achieved:

- The application programmers can work with devices provided by different protocols exactly in the same way and by applying the same program interface. The protocol adapters and device abstraction API hide the complexity/differences of the device protocols.
- The applications can work without modification when new hardware controllers and protocol adapters are dynamically added.
- When remote access to the devices connected to the gateway is necessary (e.g. in m2m and management scenarios) it's much easier to provide mapping to one API then to a set of protocol dependent APIs.
- It is much easier to build UI for remote browsers or for apps running on mobile devices if just one mapping to one unified device abstraction API is necessary.

4 Requirements

Requirement 1. The solution MUST define API for controlling devices which is applicable for all relevant device protocols.

Requirement 2. The solution MUST define API for controlling devices which is independent from the device protocols.

Requirement 3. The solution MUST include device access control based on user and application permissions compliant with the OSGi security model.

Requirement 4. The solution MUST take advantage of the security features available in the device protocols.

Requirement 5. The solution MUST include a device protocol independent notification mechanism realized according to the OSGi event mechanisms.

Requirement 6. The solution SHOULD be mappable to other relevant standards such as HGI, ETSI M2M and BBF handling the remote access to device networks.

Requirement 7. The solution MUST provide configurable device data and metadata model.

Requirement 8. The solution MUST be applicable to the changeable device behavior. Sleeping/power saving devices can go and stay offline for a long time, but should be available in the defined API.

Requirement 9. The solution MUST provide an extension mechanism to support devices provided by new protocols.

Requirement 10. The solution MAY provide means to access the protocol specific device object.

Requirement 11. The solution MUST register device or/and device related instance to the OSGi service registry.

Requirement 12. The solution MAY update OSGi Device Access Specification.

5 Technical Solution

Residential devices become more and more complicated. They can play different roles in the home networks. As a dynamic member of secure or unsecure network, they provide rich functionality. The device dynamic nature is well mappable to the OSGi service registry. That's why the technical solution is based on OSGi service registry. There is a registration of Functional Device service. It realizes basic set of management operation and provides meta information about the device. The applications are allowed to track the device statuses, to read descriptive information and to follow the device relations. Each Functional Device can have a set of functions i.e. Device Function services. Device Function represents the device operations and related properties. They are accessed from the OSGi service registry. The applications are allowed to get directly the required functions if they don't need information about the device. For example, light device is registered as a Functional Device. The light has a Device Function to turn on and turn off the light.

5.1 Device Access category

The device access category is called “FunctionalDevice”. The category name is defined as a value of FunctionalDevice.`DEVICE_CATEGORY` constant. It can be used as a part of `org.osgi.service.device.Constants.DEVICE_CATEGORY` service key value. The category impose this specification rules.

5.2 Functional Device Service

Functional Device is dedicated for a common access to devices provided by different protocols. It can be mapped one to one with the physical device, but can be mapped only with a given functional part of the device. In this scenario, the physical device can be realized with a set of Functional Device services and different relations between them. Functional Device service can represent pure software unit. For example, it can simulate the real device work. There are basic management operations for enable, disable, remove, property access and property update. New protocol devices can be supported with a registration of new Functional Device services.

If the underlying protocol and the implementation allow, the Functional Device services must be registered again after reboot of the OSGi framework. The service properties must be restored, the supported Device Functions must be provided and Functional Device relations must be visible to the applications.

The OSGi service registry has the advantage of being easily accessible. The services can be filtered and accessed with their properties. The device service has a rich set of such properties as it is on Illustration 4:

- `FunctionalDevice.PROPERTY_UID` – Specifies the device unique identifier. It's a mandatory property. The property value cannot be externally set. The value type is `java.lang.String`. To simplify the unique identifier generation, the property value must follow the rule:

  UID ::= communication-type ':' device-id

  UID – device unique identifier

  communication-type – the value of the `FunctionalDevice.PROPERTY_COMMUNICATION` service property

  device-id – device unique identifier in the scope of the communication type

- `FunctionalDevice.PROPERTY_REFERENCE_UIDS` – Specifies the reference device unique identifiers. It's an optional property. The property value cannot be externally set. The value type is `java.lang.String[]`. It can be used to represent different relationships between the devices. For example, The ZigBee controller can have a reference to the USB dongle.

- `FunctionalDevice.PROPERTY_COMMUNICATION` – Specifies the device communication possibility. It's a mandatory property. The property value cannot be externally set. The value type is `java.lang.String`. The communication interface can vary depending on the device. On protocol level, it can represent the used protocol like Zig-Bee, Z-Wave etc. The peripheral device can be registered with the used communication interface.

- `FunctionalDevice.PROPERTY_NAME` – Specifies the device name. It's an optional property. The property value can be externally set. The value type is `java.lang.String`.

- `FunctionalDevice.PROPERTY_STATUS` – Specifies the current device status. It's a mandatory property. The property value cannot be externally set. The value type `java.lang.Integer`. The possible values are:

  - `FunctionalDevice.STATUS_REMOVED` – Indicates that the device is removed. The status is available for stale device services, which are unregistered from the OSGi service registry. All transitions to and from this status are described in Transitions to STATUS_REMOVED section.

  - `FunctionalDevice.STATUS_OFFLINE` – Indicates that the device is currently not available for operations. The end device is still installed in the network and can become online later. The

controller is unplugged or there is no connection. All transitions to and from this status are described in detail in Transitions to and from STATUS_OFFLINE section.

- `FunctionalDevice.STATUS_ONLINE` – Indicates that the device is currently available for operations. All transitions to and from this status are described in detail in Transitions to and from STATUS_ONLINE section.
- `FunctionalDevice.STATUS_PROCESSING` – Indicates that the device is currently busy with an operation. All transitions to and from this status are described in detail in Transitions to and from STATUS_PROCESSING section.
- `FunctionalDevice.STATUS_DISABLED` – Indicates that the device is currently disabled. The device is not available for operations. All transitions to and from this status are described in detail in Transitions to and from STATUS_DISABLED section.
- `FunctionalDevice.STATUS_NOT_INITIALIZED` – Indicates that the device is currently not initialized. Some protocols don't provide device information right after the device is connected. The device can be initialized later when it's awakened. All transitions to and from this status are described in detail in Transitions to and from STATUS_NOT_INITIALIZED section.
- `FunctionalDevice.STATUS_NOT_CONFIGURED` – Indicates that the device is currently not configured. The device can require additional actions to become completely connected to the network. All transitions to and from this status are described in detail in Transitions to and from STATUS_NOT_CONFIGURED section.

- `FunctionalDevice.PROPERTY_STATUS_DETAIL` – Provides the reason for the current device status. It's an optional property. The property value cannot be externally set or modified. The value type is `java.lang.Integer`. There are two value categories. Positive values indicate the reason for the current status like `FunctionalDevice.STATUS_DETAIL_CONNECTING`. Negative values indicate errors related to the current device status like `FunctionalDevice.STATUS_DETAIL_DEVICE_BROKEN`. The list with defined status details is:
  - `FunctionalDevice.STATUS_DETAIL_CONNECTING` – The reason for the current device status is that the device is currently connecting to the network. It indicates the reason with a positive value `1`. The device status must be `STATUS_PROCESSING`.
  - `FunctionalDevice.STATUS_DETAIL_INITIALIZING` – The reason for the current device status is that the device is currently in process of initialization. It indicates the reason with a positive value `2`. The network controller initializing means that information about the network is currently read. The device status must be `STATUS_PROCESSING`.
  - `FunctionalDevice.STATUS_DETAIL_CONFIGURATION_NOT_APPLIED` – The reason for the current device status is that the device configuration is not applied. It indicates an error with a negative value `-1`. The device status must be `STATUS_NOT_CONFIGURED`.
  - `FunctionalDevice.STATUS_DETAIL_DEVICE_BROKEN` – The reason for the offline device is that the device is broken. It indicates an error with a negative value `-2`. The device status must be `STATUS_OFFLINE`.
  - `FunctionalDevice.STATUS_DETAIL_DEVICE_COMMUNICATION_ERROR` – The reason for the current device status is that the device communication is problematic. It indicates an error with a negative value `-3`. The device status must be `STATUS_ONLINE` or `STATUS_NOT_INITIALIZED`.
  - `FunctionalDevice.STATUS_DETAIL_DEVICE_DATA_INSUFFICIENT` – The reason for the uninitialized device is that the device doesn't provide enough information and cannot be determined. It indicates an error with a negative value `-4`. The device status must be `STATUS_NOT_INITIALIZED`.

- `FunctionalDevice.STATUS_DETAIL_DEVICE_NOT_ACCESSIBLE` – The reason for the offline device is that the device is not accessible and further communication is not possible. It indicates an error with a negative value `-5`. The device status must be `STATUS_OFFLINE`.
- `FunctionalDevice.STATUS_DETAIL_ERROR_APPLYING_CONFIGURATION` – The reason for the current device status is that the device cannot be configured. It indicates an error with a negative value `-6`. The device status must be `STATUS_NOT_CONFIGURED`.
- `FunctionalDevice.STATUS_DETAIL_IN_DUTY_CYCLE` – The reason for the offline device is that the device is in duty cycle. It indicates an error with a negative value `-7`. The device status must be `STATUS_OFFLINE`.

Custom status details are allowed, but they must not overlap the specified codes. Table 1 contains the mapping of the status details to the statuses.

Status Detail	Status
STATUS_DETAIL_CONNECTING	STATUS_PROCESSING
STATUS_DETAIL_INITIALIZING	STATUS_PROCESSING
STATUS_DETAIL_CONFIGURATION_NOT_APPLIED	STATUS_NOT_CONFIGURED
STATUS_DETAIL_DEVICE_BROKEN	STATUS_OFFLINE
STATUS_DETAIL_DEVICE_COMMUNICATION_ERROR	STATUS_ONLINE, STATUS_NOT_INITIALIZED
STATUS_DETAIL_DEVICE_DATA_INSUFFICIENT	STATUS_NOT_INITIALIZED
STATUS_DETAIL_DEVICE_NOT_ACCESSIBLE	STATUS_OFFLINE
STATUS_DETAIL_ERROR_APPLYING_CONFIGURATION	STATUS_NOT_CONFIGURED
STATUS_DETAIL_IN_DUTY_CYCLE	STATUS_OFFLINE

Table 1

- `FunctionalDevice.PROPERTY_HARDWARE_VENDOR` – Specifies the device hardware vendor. It's an optional property. The property value can be externally set. The value type is `java.lang.String`.
- `FunctionalDevice.PROPERTY_HARDWARE_VERSION` – Specifies the device hardware version. It's an optional property. The property value can be externally set. The value type is `java.lang.String`.
- `FunctionalDevice.PROPERTY_FIRMWARE_VENDOR` – Specifies the device firmware vendor. It's an optional property. The property value can be externally set. The value type is `java.lang.String`.
- `FunctionalDevice.PROPERTY_FIRMWARE_VERSION` – Specifies the device firmware version. It's an optional property. The property value can be externally set. The value type is `java.lang.String`.
- `FunctionalDevice.PROPERTY_TYPES` – Specified the device types. It's an optional property. The property value can be externally set or modified. The value type is `java.lang.String[]`. Custom types are allowed, but they must not overlap the specified types. Currently, only one type is specified:
  - `FunctionalDevice.TYPE_PERIPHERAL` – Indicates that the device is peripheral. Usually, those devices are base and contains some meta information.
- `FunctionalDevice.PROPERTY_MODEL` – Specifies the device model. It's an optional property. The property value can be externally set. The value type is `java.lang.String`.
- `FunctionalDevice.PROPERTY_SERIAL_NUMBER` – Specifies the device serial number. It's an optional property. The property value can be externally set. The value type is `java.lang.String`.

- `FunctionalDevice.PROPERTY_FUNCTION_UIDS` – Specifies the device function unique identifiers of the supported Device Function services. It's an optional property. The property value cannot be externally set. The value type is `java.lang.String[]`.
- `FunctionalDevice.PROPERTY_GROUP_UIDS` – Specifies the device functional group unique identifiers of the supported Functional Group services. It's an optional property. The value type is `java.lang.String[]`.

The device services are registered in the OSGi service registry with `org.osgi.services.functionaldevice.FunctionalDevice` interface. The next code snippet prints the online devices.

```
final ServiceReference[] deviceSRefs = context.getServiceReferences(
  FunctionalDevice.class.getName(),
  '(' + FunctionalDevice.PROPERTY_STATUS + '=' + FunctionalDevice.STATUS_ONLINE +
')');
if (null == deviceSRefs) {
  return; // no such services
}
for (int i = 0; i < deviceSRefs.length; i++) {
  printDevice(deviceSRefs[i]);
}
```

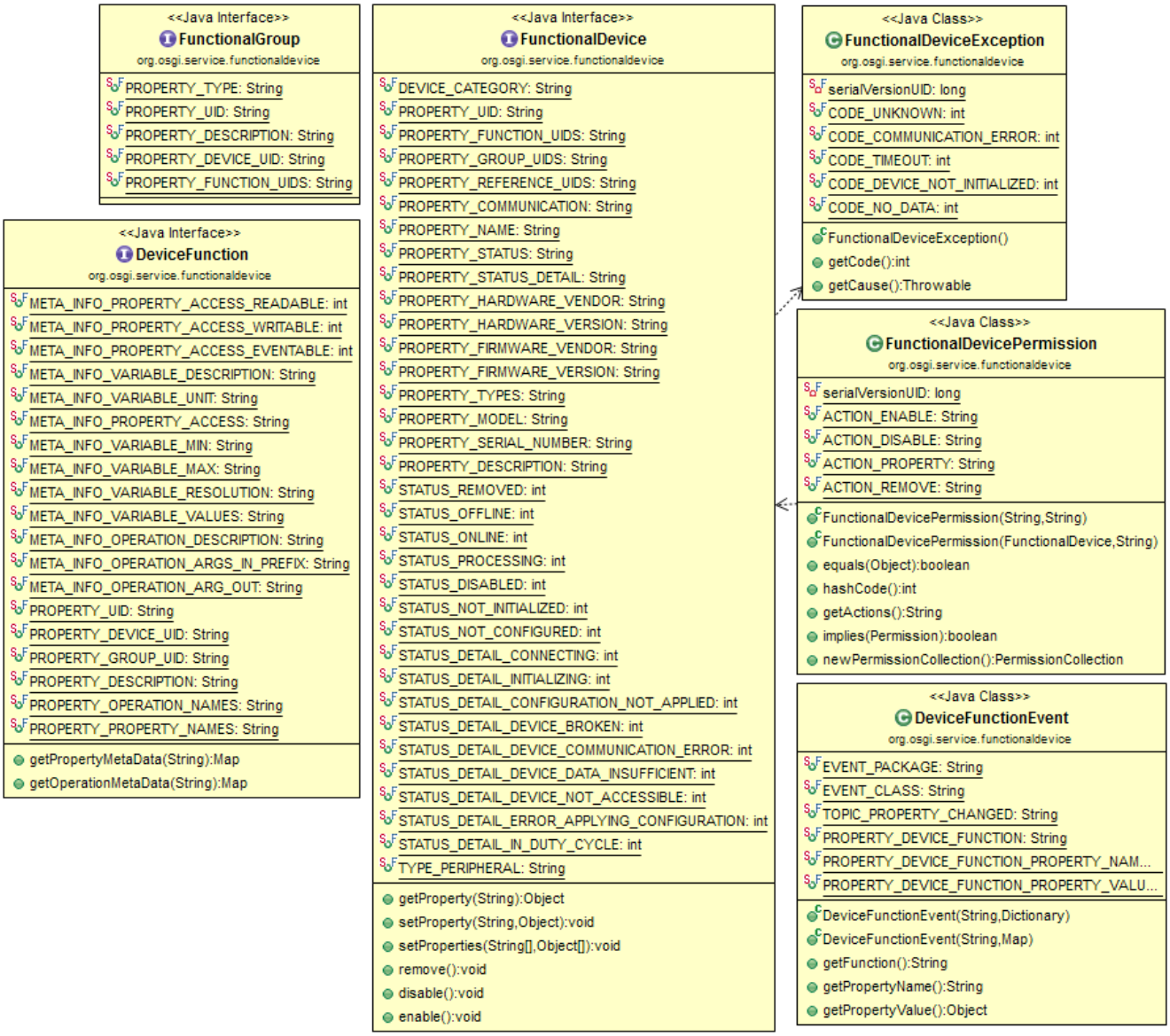

Illustration 4

Applications often require an option to modify the device property set. In OSGi, only the owner of the service registration can perform such updates. In case of device service, the `FunctionalDevice` interface gives such opportunities. They are:

- `getProperty(String propName)` – Returns the current value of the specified property. The method will return the same value as `org.osgi.framework.ServiceReference.getProperty(String)` for the service reference of this device.
- `setProperty(String propName, Object propValue)` – Sets the given property name to the given property value. The method can be used for:
  - Update – if the property name exists, the value will be updated.
  - Add – if the property name doesn't exists, a new property will be added.
  - Remove – if the property name exists and the given property value is `null`, then the property will be removed.

`java.lang.UnsupportedOperationException` will be thrown when the method is not supported.

- `setProperties(String[] propNames, Object[] propValues)` – Sets the given property names to the given property values. The method is similar to `setProperty(String, Object)`, but can update all properties with one bulk operation. `java.lang.UnsupportedOperationException` will be thrown when the method is not supported.

5.2.1 Reference Functional Device Services

`FunctionalDevice` service can have a reference to other devices. That link can be used to represent different relationships between devices. For example, the ZigBee dongle can be used as USB `FunctionalDevice` and ZigBee network controller `FunctionalDevice`. The network controller device can have a reference to the physical USB device as it's depicted on Illustration 5.

The related service property is `FunctionalDevice.PROPERTY_REFERENCE_UIDS`.

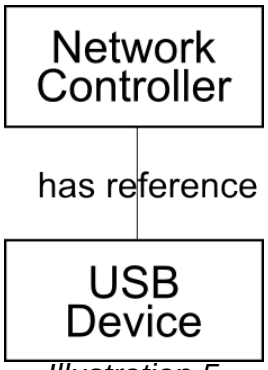

Illustration 5

5.2.2 Functional Device Service disabling and enabling

The `FunctionalDevice` service can be temporary disabled for operations with `FunctionalDevice.disable()` method call. The device will move to `FunctionalDevice.STATUS_DISABLED` status. The device can leave the disabled status with `FunctionalDevice.enable()` method call. The implementation can throw `java.lang.UnsupportedOperationException`, if `enable()` or `disable()` method is not supported.

5.2.3 Functional Device Service Adding

The devices are registered as services in the OSGi service registry. The service interface is `org.osgi.services.functionaldevice.FunctionalDevice`. There is a registration order. `FunctionalDevice` services are registered last. Before their registration, there is `DeviceFunction` and `FunctionalGroup` registration.

5.2.4 Functional Device Service Removing

OSGi service registry is only about the read only access to the services. There are no control operations. The service provider is responsible to register, update or unregister the services. That design is not very convenient for the device life cycle. The `FunctionalDevice` interface provides a callback method `remove()`. The method can be optionally implemented by the device provider. `java.lang.UnsupportedOperationException` can be thrown if the method is not supported. When the remove callback is called, an appropriate command will be synchronously send to the device. As a result it can leave the network and device related service will be unregistered. There is an unregistration order. The registration reverse order is used when the services are unregistered. `FunctionalDevice` services are unregistered first before `FunctionalGroup` and `DeviceFunction` services.

5.3 Functional Device status transitions

The Functional Device status uncover the device availability. It can demonstrate that device is currently not available for operations or that the device requires some additional configuration steps. The status can jump over the different values according to the rules defined in this section. The status transitions are summarized in Table 2 and described in detail in the next sections.

From \ To	**STATUS_PROCESSING**	**STATUS_ONLINE**	**STATUS_OFFLINE**	**STATUS_DISABLED**	**STATUS_NOT_INITIALIZED**	**STATUS_NOT_CONFIGURED**	**STATUS_REMOVED**
STATUS_PROCESSING	-	Initial device data has been read.	Device is not accessible.	Device data indicates that the device is disabled.	Initial device data is partially read.	Device has a pending configuration.	Device is removed.
STATUS_ONLINE	Device data is processing.	-	Device is not accessible.	Device is disabled.	-	Device has a new pending configuration.	Device is removed.
STATUS_OFFLINE	Device data is processing.	Device data has been read.	-	Device data indicates that the device is disabled.	-	Device has a pending configuration.	Device is removed.
STATUS_DISABLED	Device data is processing.	Device is enabled.	Device is enabled, but not accessible.	-	-	Device has a pending configuration.	Device is removed.
STATUS_NOT_INITIALIZED	Device data is processing.	-	Device is not accessible.	-	-	-	Device is removed.
STATUS_NOT_CONFIGURED	Device data is processing.	Device pending configuration is satisfied.	Device is not accessible.	Device data indicates that the device is disabled.	-	-	Device is removed.
STATUS_REMOVED	-	-	-	-	-	-	-

Table 2

5.3.1 Transitions to STATUS_REMOVED

The Functional Device can go to `FunctionalDevice.STATUS_REMOVED` from any other status. Once reached, the device status cannot be updated any more. The device is removed from the network and the device service is unregistered from the OSGi service registry. If there are stale references to the Functional Device service, their status will be set to `STATUS_REMOVED`.

The common way for a given device to be removed is `FunctionalDevice.remove()`. When the method returns, the device status should be `STATUS_REMOVED`. It requires a synchronous execution of the operation.

5.3.2 Transitions to and from STATUS_OFFLINE

The `STATUS_OFFLINE` indicates that the Functional Device is currently not available for operations. That status can be set, because of different reasons. The network controller can be unplugged, connection to the device is lost etc. This variety provides an access to that status from any other except `STATUS_REMOVED`. Transitions to and from this status are:

- From `STATUS_OFFLINE` to `STATUS_REMOVED` – Functional Device is removed. The status can be set as a result of `FunctionalDevice.remove()` method call.
- From `STATUS_OFFLINE` to `STATUS_PROCESSING` – Functional Device data is processing.
- From `STATUS_OFFLINE` to `STATUS_NOT_CONFIGURED` – Functional Device has a pending configuration.
- From `STATUS_OFFLINE` to `STATUS_DISABLED` – Functional Device is currently disabled. The status can be set as a result of `FunctionalDevice.disable()` method call.
- From `STATUS_OFFLINE` to `STATUS_ONLINE` – Functional Device data has been read and the device is currently available for operations.
- From `STATUS_OFFLINE` to `STATUS_NOT_INITIALIZED` – That transition is not possible, because the status have to go through `STATUS_PROCESSING`. If the processing is unsuccessful, `STATUS_NOT_INITIALIZED` will be set.
- To `STATUS_OFFLINE` from `STATUS_REMOVED` – That transition is not possible. If Functional Device is removed, the service will be unregistered from the service registry.
- To `STATUS_OFFLINE` from `STATUS_PROCESSING` – Functional Device is not accessible any more while device data is processing.
- To `STATUS_OFFLINE` from `STATUS_NOT_CONFIGURED` – Not configured Functional Device is not accessible any more.
- To `STATUS_OFFLINE` from `STATUS_DISABLED` – Disabled Functional Device is not accessible any more. The status can be set as a result of `FunctionalDevice.enable()` method call.
- To `STATUS_OFFLINE` from `STATUS_ONLINE` – Online Functional Device is not accessible any more.
- To `STATUS_OFFLINE` from `STATUS_NOT_INITIALIZED` – Not initialized Functional Device is not accessible any more.

The possible transitions are summarized on Illustration 6.

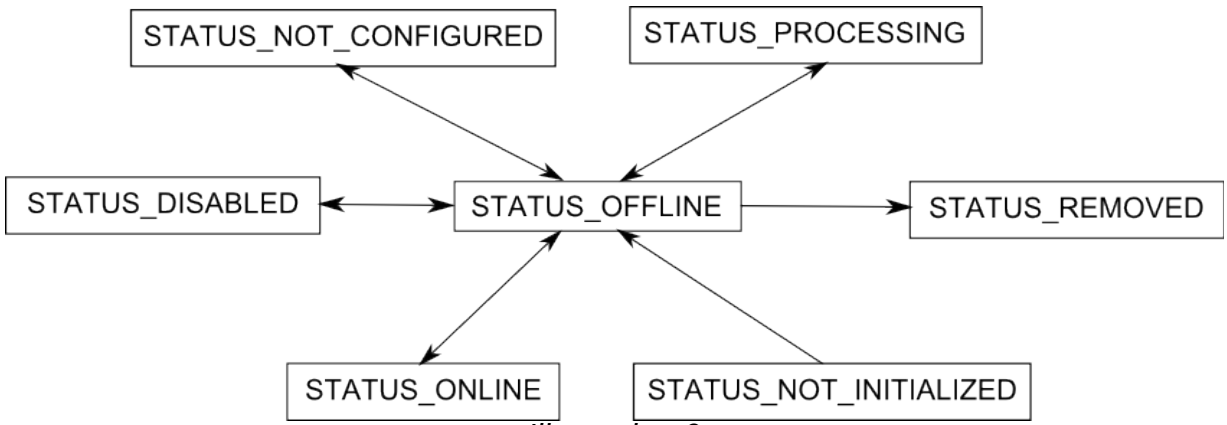

Illustration 6

5.3.3 Transitions to and from STATUS_ONLINE

The `STATUS_ONLINE` indicates that the Functional Device is currently available for operations. The online devices are initialized and ready for use. Transitions to and from this status are:

- From `STATUS_ONLINE` to `STATUS_REMOVED` – Functional Device is removed. The status can be set as a result of `FunctionalDevice.remove()` method call.
- From `STATUS_ONLINE` to `STATUS_PROCESSING` – Functional Device data is processing.
- From `STATUS_ONLINE` to `STATUS_NOT_CONFIGURED` – Functional Device has a pending configuration.
- From `STATUS_ONLINE` to `STATUS_DISABLED` – Functional Device is currently disabled. The status can be set as a result of `FunctionalDevice.disable()` method call.
- From `STATUS_ONLINE` to `STATUS_OFFLINE` – Online Functional Device is not accessible any more.
- From `STATUS_ONLINE` to `STATUS_NOT_INITIALIZED` – That transition is not possible. Online devices are initialized.
- To `STATUS_ONLINE` from `STATUS_REMOVED` – That transition is not possible. If Functional Device is removed, the service will be unregistered from the service registry.
- To `STATUS_ONLINE` from `STATUS_PROCESSING` – Initial Functional Device data has been read. The device is available for operations.
- To `STATUS_ONLINE` from `STATUS_NOT_CONFIGURED` – The Functional Device pending configuration is satisfied.
- To `STATUS_ONLINE` from `STATUS_DISABLED` – The Functional Device is enabled. The status can be set as a result of `FunctionalDevice.enable()` method call.
- To `STATUS_ONLINE` from `STATUS_OFFLINE` – Functional Device is accessible for operations.
- To `STATUS_ONLINE` from `STATUS_NOT_INITIALIZED` – That transition is not possible. The device data has to be processed and then the device can become online. Intermediate status `STATUS_PROCESSING` will be used.

The possible transitions are summarized on Illustration 7.

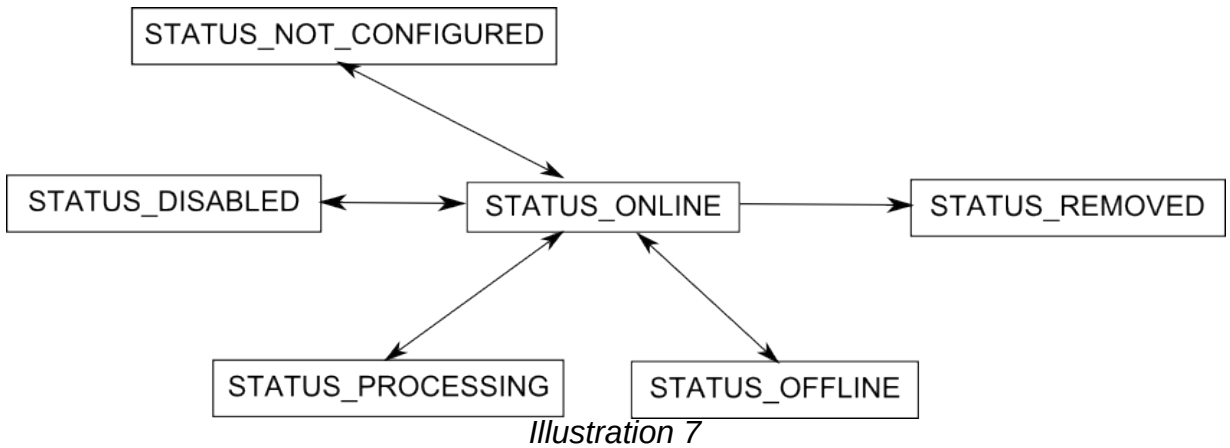

Illustration 7

5.3.4 Transitions to and from STATUS_PROCESSING

The status indicates that the device is currently busy with an operation. It can be time consuming operation and can result to any other status. The operation processing can be reached by any other status except `STATUS_REMOVED`. An example, offline device requires some data processing to become online. It will apply the

statuses STATUS_OFFLINE, STATUS_PROCESSING and STATUS_ONLINE. Transitions to and from this status are:

- From STATUS_PROCESSING to STATUS_REMOVED – Functional Device is removed. The status can be set as a result of FunctionalDevice.remove() method call.
- From STATUS_PROCESSING to STATUS_ONLINE – Initial Functional Device data has been read. The device is available for operations.
- From STATUS_PROCESSING to STATUS_NOT_CONFIGURED – Functional Device has a pending configuration.
- From STATUS_PROCESSING to STATUS_DISABLED – Functional Device is currently disabled. The status can be set as a result of FunctionalDevice.disable() method call.
- From STATUS_PROCESSING to STATUS_OFFLINE – Online Functional Device is not accessible any more.
- From STATUS_PROCESSING to STATUS_NOT_INITIALIZED – Functional Device initial data is partially read.
- To STATUS_PROCESSING from STATUS_REMOVED – That transition is not possible. If Functional Device is removed, the service will be unregistered from the service registry.
- To STATUS_PROCESSING from STATUS_ONLINE – Functional Device is busy with an operation.
- To STATUS_PROCESSING from STATUS_NOT_CONFIGURED – The Functional Device pending configuration is satisfied and the device is busy with an operation.
- To STATUS_PROCESSING from STATUS_DISABLED – The Functional Device is enabled and busy with an operation. The status can be set as a result of FunctionalDevice.enable() method call.
- To STATUS_PROCESSING from STATUS_OFFLINE – Functional Device is busy with an operation.
- To STATUS_PROCESSING from STATUS_NOT_INITIALIZED – Functional Device initial data is processing.

The possible transitions are summarized on Illustration 8.

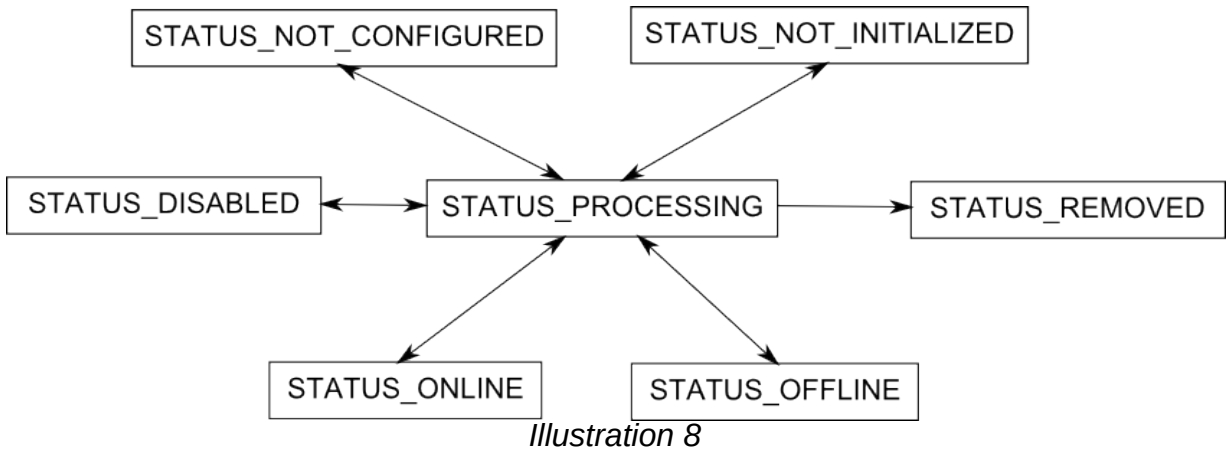

Illustration 8

5.3.5 Transitions to and from STATUS_DISABLED

The status indicates that the device is currently disabled for operations. Such devices can be only enabled. While enabling the device can go in different status depending on the required operations. Transitions to and from this status are:

- From `STATUS_DISABLED` to `STATUS_REMOVED` – Functional Device is removed. The status can be set as a result of `FunctionalDevice.remove()` method call.
- From `STATUS_DISABLED` to `STATUS_PROCESSING` – Functional Device is enabled and device data is processing. The status can be set as a result of `FunctionalDevice.enable()` method call.
- From `STATUS_DISABLED` to `STATUS_NOT_CONFIGURED` – Functional Device is enabled but it has a pending configuration. The status can be set as a result of `FunctionalDevice.enable()` method call.
- From `STATUS_DISABLED` to `STATUS_ONLINE` – Functional Device is enabled. The status can be set as a result of `FunctionalDevice.enable()` method call.
- From `STATUS_DISABLED` to `STATUS_OFFLINE` – Functional Device is not accessible any more.
- From `STATUS_DISABLED` to `STATUS_NOT_INITIALIZED` – That transition is not possible. Functional Device has to be initialized to be operable.
- To `STATUS_DISABLED` from `STATUS_REMOVED` – That transition is not possible. If Functional Device is removed, the service will be unregistered from the service registry.
- To `STATUS_DISABLED` from `STATUS_PROCESSING` – Functional Device data indicates that the device is currently disabled.
- To `STATUS_DISABLED` from `STATUS_NOT_CONFIGURED` – Functional Device data indicates that the device is currently disabled.
- To `STATUS_DISABLED` from `STATUS_ONLINE` – The Functional Device is disabled. The status can be set as a result of `FunctionalDevice.disable()` method call.
- To `STATUS_DISABLED` from `STATUS_OFFLINE` – Functional Device data indicates that the device is disabled.
- To `STATUS_DISABLED` from `STATUS_NOT_INITIALIZED` – That transition is not possible. Not initialized device requires data processing to become operable.

The possible transitions are summarized on Illustration 9.

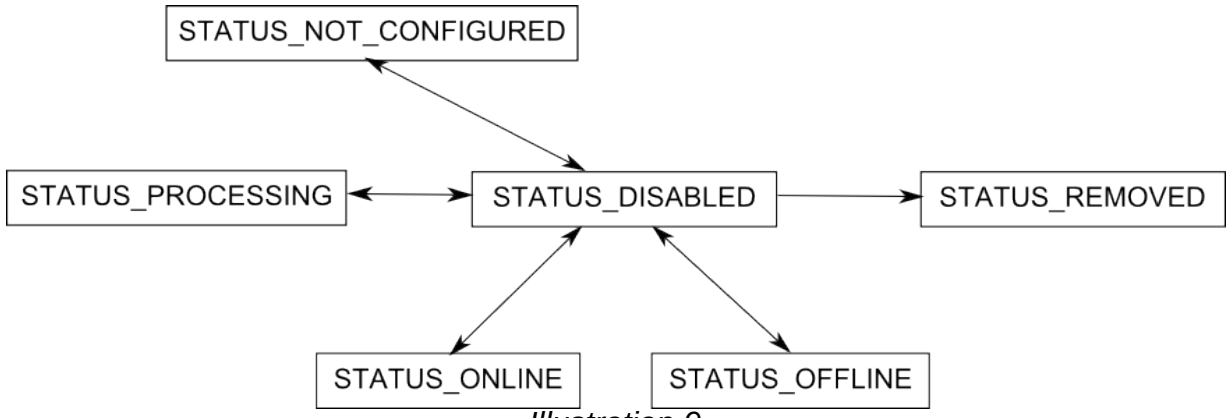

Illustration 9

5.3.6 Transitions to and from STATUS_NOT_INITIALIZED

The status indicates that the device is currently not initialized. Some protocols don't provide device information right after the device is connected. The device can be initialized later when it's awakened. Not initialized device requires some data processing to become online. `STATUS_PROCESSING` is used as an intermediate status. Transitions to and from this status are:

- From `STATUS_NOT_INITIALIZED` to `STATUS_REMOVED` – Functional Device is removed. The status can be set as a result of `FunctionalDevice.remove()` method call.
- From `STATUS_NOT_INITIALIZED` to `STATUS_PROCESSING` – Functional Device data is processing.
- From `STATUS_NOT_INITIALIZED` to `STATUS_NOT_CONFIGURED` – That transition is not possible. Functional Device requires some data processing.
- From `STATUS_NOT_INITIALIZED` to `STATUS_DISABLED` – That transition is not possible. Not initialized device requires data processing to become operable.
- From `STATUS_NOT_INITIALIZED` to `STATUS_OFFLINE` – Functional Device is not accessible any more.
- From `STATUS_NOT_INITIALIZED` to `STATUS_ONLINE` – That transition is not possible. Functional Device requires some data processing to become online.
- To `STATUS_NOT_INITIALIZED` from `STATUS_REMOVED` – That transition is not possible. If Functional Device is removed, the service will be unregistered from the service registry.
- To `STATUS_NOT_INITIALIZED` from `STATUS_PROCESSING` – Functional Device data is partially read.
- To `STATUS_NOT_INITIALIZED` from `STATUS_NOT_CONFIGURED` – That transition is not possible. When Functional Device pending configuration is satisfied, the device requires additional data processing.
- To `STATUS_NOT_INITIALIZED` from `STATUS_DISABLED` – That transition is not possible. Functional Device has to be initialized to be operable.
- To `STATUS_NOT_INITIALIZED` from `STATUS_OFFLINE` – That transition is not possible. Functional Device requires some data processing and then can become not initialized.
- To `STATUS_NOT_INITIALIZED` from `STATUS_ONLINE` – That transition is not possible. Online Functional Device is initialized.

The possible transitions are summarized on Illustration 10.

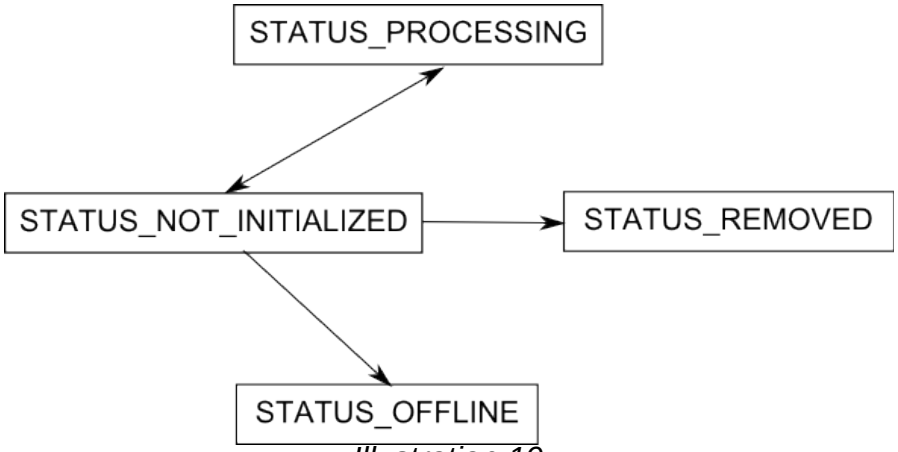

Illustration 10

5.3.7 Transitions to and from STATUS_NOT_CONFIGURED

Indicates that the device is currently not configured. The device can require additional actions to become completely connected to the network. For example, a given device button has to be pushed. That status doesn't have transitions with `STATUS_NOT_INITIALIZED`, because some data processing is required. Transitions to and from this status are:

- From `STATUS_NOT_CONFIGURED` to `STATUS_REMOVED` – Functional Device is removed. The status can be set as a result of `FunctionalDevice.remove()` method call.
- From `STATUS_NOT_CONFIGURED` to `STATUS_PROCESSING` – Functional Device pending configuration is satisfied and some additional data processing is required.
- From `STATUS_NOT_CONFIGURED` to `STATUS_ONLINE` – Functional Device pending configuration is satisfied.
- From `STATUS_NOT_CONFIGURED` to `STATUS_DISABLED` – Functional Device is currently disabled. The status can be set as a result of `FunctionalDevice.disable()` method call.
- From `STATUS_NOT_CONFIGURED` to `STATUS_OFFLINE` – Functional Device is not accessible any more.
- From `STATUS_NOT_CONFIGURED` to `STATUS_NOT_INITIALIZED` – That transition is not possible. When Functional Device pending configuration is satisfied, the device requires additional data processing.
- To `STATUS_NOT_CONFIGURED` from `STATUS_REMOVED` – That transition is not possible. If Functional Device is removed, the service will be unregistered from the service registry.
- To `STATUS_NOT_CONFIGURED` from `STATUS_PROCESSING` – Initial Functional Device data has been read but there is a pending configuration.
- To `STATUS_NOT_CONFIGURED` from `STATUS_ONLINE` – Functional Device has a pending configuration.
- To `STATUS_NOT_CONFIGURED` from `STATUS_DISABLED` – The Functional Device is enabled but has a pending configuration. The status can be set as a result of `FunctionalDevice.enable()` method call.
- To `STATUS_NOT_CONFIGURED` from `STATUS_OFFLINE` – Functional Device is going to be online, but has a pending configuration.
- To `STATUS_NOT_CONFIGURED` from `STATUS_NOT_INITIALIZED` – That transition is not possible. That transition is not possible. Functional Device requires some data processing.

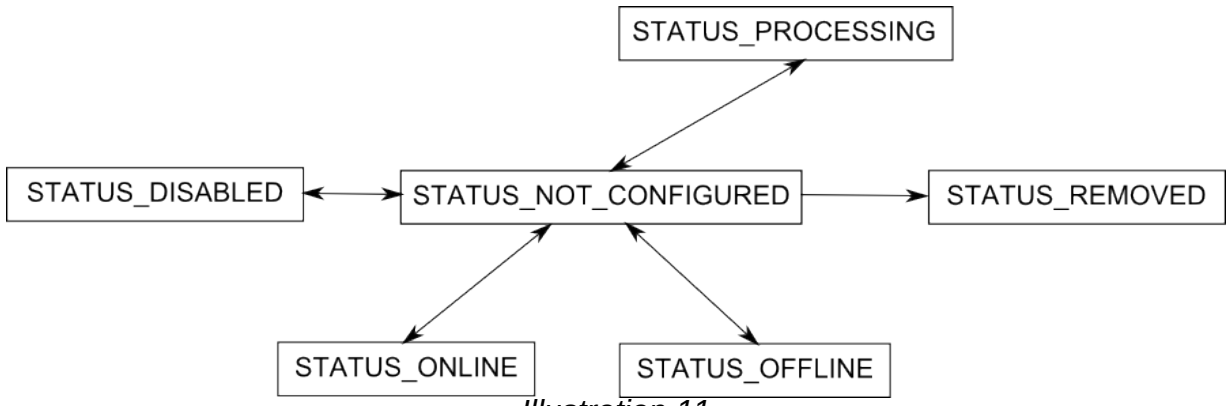

Illustration 11

The possible transitions are summarized on Illustration 11.

5.4 Functional Groups

TODO: Discuss on Paris F2F.

5.5 Device Functions

The user applications can execute the device operations and manage the device properties. That control is realized with the help of Device Function services. The Device Function service can be registered in the service registry with those service properties:

- `DeviceFunction.PROPERTY_UID` – mandatory service property. The property value is the Device Function unique identifier. `The value type is java.lang.String`. To simplify the unique identifier generation, the property value must follow the rule:

  function UID ::= device-id ':' function-id

  function UID – device function unique identifier

  device-id – the value of the `FunctionalDevice.PROPERTY_UID` Functional Device service property

  function-id – device function identifier in the scope of the device

- `DeviceFunction.PROPERTY_DEVICE_UID` – optional service property. The property value is the Functional Device identifier. The Device Function belongs to this device. The value type is `java.lang.String`.
- `DeviceFunction.PROPERTY_GROUP_UID` – optional service property. The property value is the Functional Group identifier. The Device Function belongs to this group. The value type is `java.lang.String.`
- `DeviceFunction.PROPERTY_DESCRIPTION` – optional service property. The property value is the device function description. The value type is `java.lang.String`.
- `DeviceFunction.PROPERTY_OPERATION_NAMES` – optional service property. The property value is the Device Function operation names. The value type is `java.lang.String[].`
- `DeviceFunction.PROPERTY_PROPERTY_NAMES` – optional service property. The property value is the Device Function property names. The value type is `java.lang.String[].`

The Device Function services are registered before the Functional Device service. It's possible that `DeviceFunction.PROPERTY_DEVICE_UID` and `DeviceFunction.PROPERTY_GROUP_UID` point to missing services at the moment of the registration. The reverse order is used when the services are unregistered. Functional Device and Functional Group service are unregistered before the Device Function services.

Device Function service must be registered only under concrete Device Function classes. It's not allowed to register Device Function service under classes, which are not concrete Device Functions. For example, those registrations are not allowed:

- `context.registerService(new String[] {ManagedService.class.getName(), OnOff.class.getName()}, this, regProps);` - `ManagedService` interface is not a Device Function interface;
- `context.registerService(new String[] {DeviceFunction.class.getName(), OnOff.class.getName()}, this, regProps);` - `DeviceFunction` interface is not concrete Device Function.

That one is valid `context.registerService(new String[] {Meter.class.getName(), OnOff.class.getName()}, this, regProps);`. `Meter` and `OnOff` are valid Device Function interfaces. That rule helps to the applications to find all supported Device Function classes. Otherwise the Device Function services can be accesses, but it's not clear which are the Device Function classes.

5.5.1 Device Function interface

Device Function is built by a set of properties and operations. The function can have name, description and link to the Functional Devices. `DeviceFunction` interface must be the base interface for all functions. If the device provider defines custom functions, they must extend `DeviceFunction` interface. It provides a common access to the operations and properties meta data.

There are some general type rules, which unifies the access to the Device Function data. They make easier the transfer over different protocols. All properties and operation arguments must use:

- Java primitive type or corresponding reference type.

- `java.lang.String`
- Java Beans, but their properties must use those rules. Java Beans are defined in JavaBeans specification [3].
- `java.util.Map` instances. The map keys can be any reference type of Java primitive types or `java.lang.String`. The values must use those rules.
- Arrays of defined types.

In order to provide common behavior, all Device Functions must follow a set of common rules related to the implementation of their setters, getters, operations and events:

- The setter method must be executed synchronously. If the underlying protocol can return response to the setter call, it must be awaited. It simplifies the property value modifications and doesn't require asynchronous call back.
- The operation method must be executed synchronously. If the underlying protocol can return an operation confirmation or response, they must be awaited. It simplifies the operation execution and doesn't require asynchronous call back.
- The getter must return the last know cached property value. The device implementation is responsible to keep that value up to date. It'll speed up the applications when the Device Function property values are collected. The same cached value can be shared between a few requests instead of a few calls to the real device.
- If a given Device Function operation, getter or setter is not supported, `java.lang.UnsupportedOperationException` must be thrown. It indicates that Device Function is partially supported.
- Device Function operations, getters and setters must not override `java.lang.Object` and `org.osgi.services.functionaldevice.DeviceFunction` methods. For example:
  - `hashCode()` – it's `java.lang.Object` method and invalid Device Function operation;
  - `wait()` – it's `java.lang.Object` method and invalid Device Function operation;
  - `getClass()` – it's `java.lang.Object` method and invalid Device Function getter;
  - `getPropertyMetaData(String propertyName)` – it's `org.osgi.service.functionaldevice.DeviceFunction` method and invalid Device Function getter.

5.5.2 Device Function operations

`DeviceFunction` operations are general callable units. They can perform a specific task on the device like turn on or turn off. They can be used by the applications to control the device. Operation names are available as a value of the service property `DeviceFunction.PROPERTY_OPERATION_NAMES`. The operations are identified by their names. It's not possible to exist two operations with the same name i.e. overloaded operations are not allowed.

The operations can be optionally described with a set of meta data properties. The property values can be collected with `DeviceFunction.getOperationMetaData(String)` method. The method result is `java.util.Map` with:

- `DeviceFunction.META_INFO_OPERATION_DESCRIPTION` – Specifies a user readable description of the operation. It's an optional property. The property value type is `java.lang.String`.
- `DeviceFunction.META_INFO_OPERATION_ARG_OUT` – Specifies the operation output argument metadata. If the operation doesn't have return value, the property is missing. The value type is `java.util.Map`. The keys of the map value can be one of:

- `DeviceFunction.META_INFO_VARIABLE_DESCRIPTION` – Specifies a user readable description of the operation output argument. There are no additional property limitations.
- `DeviceFunction.META_INFO_VARIABLE_UNIT` – Specifies the measurement unit of the operation output argument as it's defined in Device Function properties. There are no additional property limitations.
- `DeviceFunction.META_INFO_VARIABLE_MIN` – Specifies the operation output argument minimum value. There are no additional property limitations.
- `DeviceFunction.META_INFO_VARIABLE_MAX` – Specifies the operation output argument maximum value. There are no additional property limitations.
- `DeviceFunction.META_INFO_VARIABLE_RESOLUTION` – Specifies the difference between two values in series of the operation output argument. There are no additional property limitations.
- `DeviceFunction.META_INFO_VARIABLE_VALUES` – Specifies the valid values of the operation output argument. There are no additional property limitations.
- Custom key – Any custom key can define additional metadata.

- `DeviceFunction.META_INFO_OPERATION_ARGS_IN_PREFIX` – A meta data key prefix. It marks the operation input argument metadata, if any. The operation can have zero or more input arguments. The property value type is `java.util.Map`. The keys of the map value can be one of:
  - `DeviceFunction.META_INFO_VARIABLE_DESCRIPTION` – Specifies a user readable description of the operation input argument. There are no additional property limitations.
  - `DeviceFunction.META_INFO_VARIABLE_UNIT` – Specifies the measurement unit of the operation input argument as it's defined in Device Function properties. There are no additional property limitations.
  - `DeviceFunction.META_INFO_VARIABLE_MIN` – Specifies the operation input argument minimum value. There are no additional property limitations.
  - `DeviceFunction.META_INFO_VARIABLE_MAX` – Specifies the operation input argument maximum value. There are no additional property limitations.
  - `DeviceFunction.META_INFO_VARIABLE_RESOLUTION` – Specifies the difference between two values in series of the operation input argument. There are no additional property limitations.
  - `DeviceFunction.META_INFO_VARIABLE_VALUES` – Specifies the valid values of the operation input argument. There are no additional property limitations.
  - Custom key – Any custom key can define additional metadata.

  The input argument prefix must be used in the form:

  operation input argument name ::= device.function.operation.arguments.in.<argument-index>,

  <argument_index> - the input argument index.

  For example, device.function.operation.arguments.in.1 can be used for the first operation input argument.

The operation arguments must follow the general type rules.

5.5.3 Device Function properties

`DeviceFunction` properties are data fields. Their values can be read with getter methods and can be set with setter methods. The property names are available as a value of the service property

`DeviceFunction.PROPERTY_PROPERTY_NAMES`. The properties are identified by their names. It's not possible to exist two properties with the same name.

The properties can be optionally described with a set of meta data properties. The property values can be collected with `DeviceFunction.getPropertyMetaData(String)` method. The method result is `java.util.Map` with:

- `DeviceFunction.META_INFO_PROPERTY_ACCESS` – Specifies the access to the device property. It's a bitmap of `java.lang.Integer` type. The bitmap can be any combination of:
  - `DeviceFunction.META_INFO_PROPERTY_ACCESS_READABLE` – Marks the property as a readable. Device Function must provide a getter method for this property according to JavaBeans specification [3]. Device Function operations must not be overridden by this getter method.
  - `DeviceFunction.META_INFO_PROPERTY_ACCESS_WRITABLE` – Marks the property as writable. Device Function must provide a setter method for this property according to JavaBeans specification [3]. Device Function operations must not be overridden by this setter method.
  - `DeviceFunction.META_INFO_PROPERTY_ACCESS_EVENTABLE` – Marks the property as eventable. Device Function must not provide special methods because of this access type. `DeviceFunctionEvent` is sent on property change. Note that the event can be sent when there is no value change.
- `DeviceFunction.META_INFO_VARIABLE_DESCRIPTION` – Specifies a user readable description of the property or the operation argument. It's an optional property. The property value type is `java.lang.String`.
- `DeviceFunction.META_INFO_VARIABLE_UNIT` – Specifies the property or the operation argument value unit. The property value is `java.lang.String` type. These rules must be applied to unify the representation:
  - SI units (The International System of Units) must be used where it's applicable.
  - The unit must use Unicode symbols normalized with NFKD (Compatibility Decomposition) normalization form [4].

  For example, degrees Celsius will not be represent as U+2103 (degree celsius), but will be U+00B0 degree sign + U+0043 latin capital letter c.
- `DeviceFunction.META_INFO_VARIABLE_MIN` – Specifies the property or the operation argument minimum value. The value type depends on the property or argument type.
- `DeviceFunction.META_INFO_VARIABLE_MAX` – Specifies the property or the operation argument maximum value. The value type depends on the property or argument type.
- `DeviceFunction.META_INFO_VARIABLE_RESOLUTION` – Specifies the resolution value of a specific range. The value type depends on the property or orgument type. For example, the resolution in [0, 100] can be 10. That's the difference between two values in series.
- `DeviceFunction.META_INFO_VARIABLE_VALUES` – Specifies the property or the operation argument possible values. The value type is `java.util.Map`, where the keys are the possible values and the values are their string representation.

5.5.4 Device Function property event

The eventable Device Function properties can trigger a new event on each property value touch. It doesn't require a modification of the value. For example, the motion sensor can send a few events with no property value change when motion is detected and continued to be detected. The event must implement `DeviceFunctionEvent` interface. The event properties are:

- All event source device properties.

- `DeviceFunctionEvent.PROPERTY_DEVICE_FUNCTION` – the event source function.
- `DeviceFunctionEvent.PROPERTY_DEVICE_FUNCTION_PROPERTY_NAME` – the property name.
- `DeviceFunctionEvent.PROPERTY_DEVICE_FUNCTION_PROPERTY_VALUE` – the property value.

For example, there is Device Function with an eventable boolean property called “state”. When “state” value is changed to `false`, Device Function implementation can post:

```
DeviceFunctionEvent {
    functional.device.UID=ACME:1
    …
    device.function=acme.function
    device.function.property.name=”state”
    device.function.property.value=java.lang.Boolean.FALSE
}
```

5.6 Basic Device Functions

Concrete Device Function interfaces have to be defined to unify the access and control of the basic operations and related properties. The current section specifies the minimal basic set of such functionality. It can be reused and extended to cover more specific scenarios. There are concentrated about the control, monitoring and metering information.

5.6.1 OnOff Device Function

`OnOff` Device Function represents turn on and off functionality. The function doesn't provide an access to properties, there are only operations. `turnOn()` method turns the device on. The device can be turned off with `turnOff()` method. The function class diagram is depicted on Error: Reference source not found. The next code snippet turns on all OnOff Device Functions.

```
final ServiceReference[] onOffSRefs = context.getServiceReferences
(OnOff.class.getName(),null);
if (null == onOffSRefs) {
  return; // no such services
}
for (int i = 0; i < onOffSRefs.length; i++) {
  final OnOff onOff = (OnOff)context.getService(onOffSRefs[i]);
  if (null != onOff) {
    onOff.turnOn();
  }
}
```

5.6.2 OpenClose Device Function

`OpenClose` Device Function represents open and close functionality. The function doesn't provide an access to properties, there are only operations. `open()` method will trigger open operation. `close()` will trigger close operation. The function class diagram is depicted on Error: Reference source not found.

5.6.3 LockUnlock Device Function

`LockUnlock` Device Function represents lock and unlock functionality. The function doesn't provide an access to properties, there are only operations. `lock()` method will trigger lock operation. `unlock()` will trigger unlock operation. The function class diagram is depicted on Error: Reference source not found.

5.6.4 BinarySwitch Device Function

`BinarySwitch` Device Function provides a binary switch control. It extends `OnOff` Device Function with a property `state`. The state value is accessible with `getState()` getter. The state can be reversed with `toggle()` method. `STATE_ON` state can be reached with the inherited method `turnOn()`. `STATE_OFF` state can be reached with the inherited method `turnOff()`. The property eventing must follow the definition in Device Function property event. Device Function class diagram is depicted on Error: Reference source not found.

5.6.5 MultiLevelSwitch Device Function

`MultiLevelSwitch` Device Function has a level, can increase or decrease the level with a given step and can set the level to a specific value. The level is accessible with `getLevel()` getter and can be set with `setLevel(int)` setter. The step is accessible with `getStep()` getter. `stepUp()` and `stepDown()` can increase and decrease the level with a given step. The property eventing must follow the definition in Device Function property event. Device Function class diagram is depicted on Error: Reference source not found.

5.6.6 BinarySensor Device Function

`BinarySensor` Device Function provides binary sensor monitoring. It reports its state when an important event is available. The state is accessible with `getState()` getter. The method returns a boolean value. There are no operations. The property eventing must follow the definition in Device Function property event. Device Function class diagram is depicted on Error: Reference source not found.

5.6.7 MultiLevelSensor Device Function

`MultiLevelSensor` Device Function provides a multi-level sensor monitoring. It reports its state when an important event is available. The state is accessible with `getState()` getter. The valid value can be every `double` value. There are no operations. The property eventing must follow the definition in Device Function property event. Device Function class diagram is depicted on Error: Reference source not found.

5.6.8 Meter Device Function

`Meter` Device Function can measure metering information. The function provides two properties:

- `PROPERTY_CURRENT` – accessible with `getCurrent()` getter. The property contains the current consumption.
- `PROPERTY_TOTAL` – property accessible with `getTotal()` getter. The property contains the total consumption. It has been measured since the last call of `resetTotal()` or device initial run.

There is an operation `OPERATION_RESET_TOTAL`, which can be executed with `resetTotal()` method. The operation will clean up `PROPERTY_TOTAL` property value.

The property eventing must follow the definition in Device Function property event. Device Function class diagram is depicted on Error: Reference source not found.

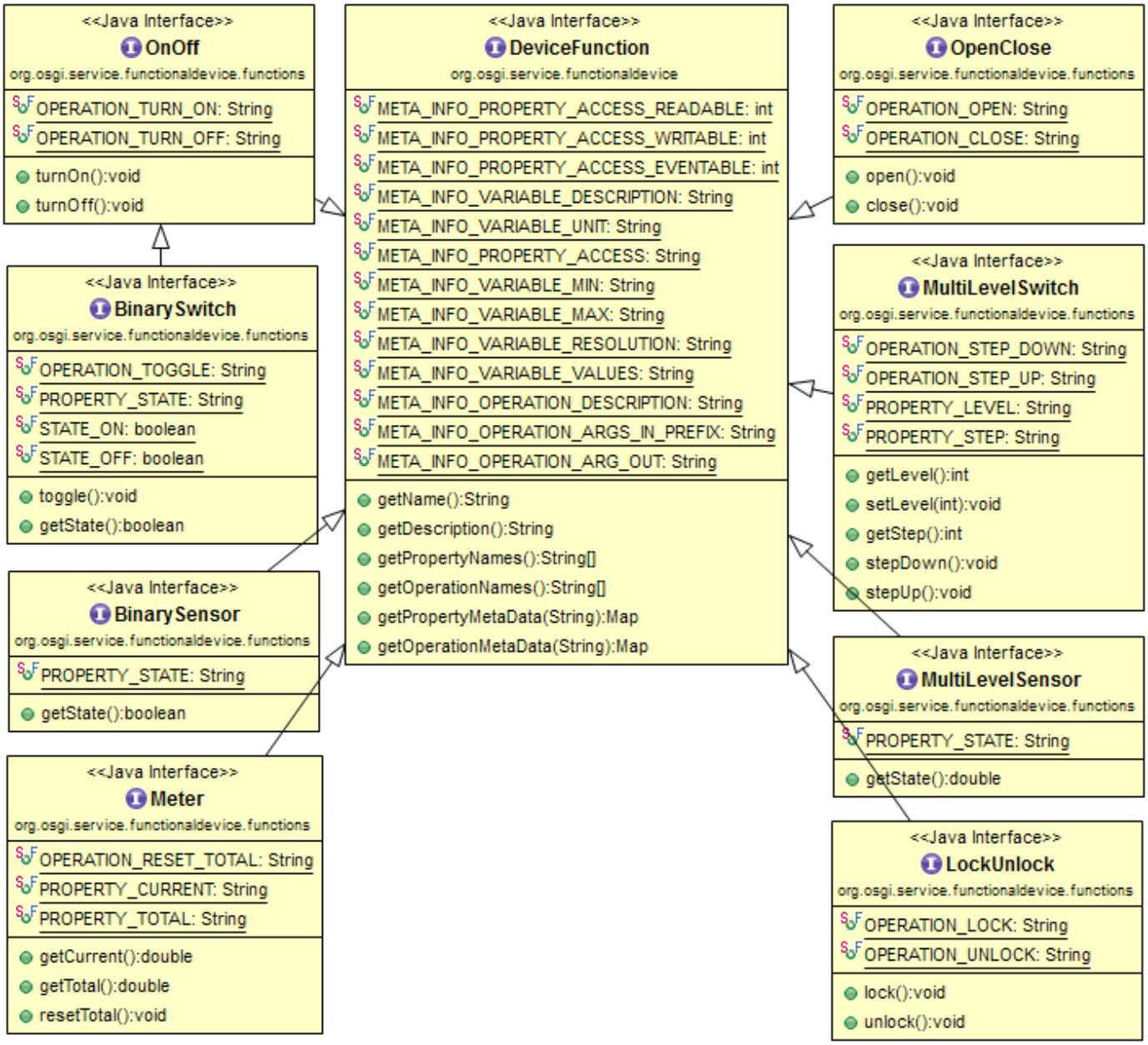

Illustration 12

6 Data Transfer Objects

TODO: Do we need those objects?

7 Javadoc

OSGi Javadoc

7/2/13 4:06 PM

Package Summary		Page
org.osgi.service.functionaldevice	Functional Device Package Version 1.0.	33
org.osgi.service.functionaldevice.functions	Functional Device Functions 1.0.	63

Package org.osgi.service.functionaldevice

Functional Device Package Version 1.0.

See:
Description

Interface Summary		Page
DeviceFunction	Device Function service provides specific device operations and properties.	34
FunctionalDevice	Represents the functional device in the OSGi service registry.	45
FunctionalGroup	The functional group can unite similar functionality based on the group type.	61

Class Summary		Page
DeviceFunctionEvent	Asynchronous event, which marks a Device Function property value modification.	42
FunctionalDevicePermission	A bundle's authority to perform specific privileged administrative operations on the devices.	57

Exception Summary		Page
FunctionalDeviceException	Thrown to indicate that there is a device operation fail.	55

Package org.osgi.service.functionaldevice Description

Functional Device Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.functionaldevice; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.functionaldevice; version="[1.0,1.1)"
```

Interface DeviceFunction

org.osgi.service.functionaldevice

All Known Subinterfaces:
BinarySensor, BinarySwitch, LockUnlock, Meter, MultiLevelSensor, MultiLevelSwitch, OnOff, OpenClose

---

```
public interface DeviceFunction
```

Device Function service provides specific device operations and properties. Each Device Function service must implement this interface. In additional to this interface, the implementation can provide own:

- properties;
- operations.

The Device Function service can be registered in the service registry with those service properties:

- `PROPERTY_DEVICE_UID` - optional service property. The property value is the Functional Device identifiers. The Device Function belongs to those devices.
- `PROPERTY_DESCRIPTION` - optional service property. The property value is the device function description.
- `PROPERTY_OPERATION_NAMES` - optional service property. The property value is the Device Function operation names.
- `PROPERTY_PROPERTY_NAMES` - optional service property. The property value is the Device Function property names.

The `DeviceFunction` services are registered before the `FunctionalDevice` and `FunctionalGroup` services. It's possible that `PROPERTY_DEVICE_UID` and `PROPERTY_GROUP_UID` point to missing services at the moment of the registration. The reverse order is used when the services are unregistered. `DeviceFunction` services are unregistered last after `FunctionalDevice` and `FunctionalGroup` services.

Device Function service must be registred only under concrete Device Function classes. It's not allowed to register Device Function service under classes, which are not concrete Device Functions. For example, those registrations are not allowed:

- `context.registerService(new String[] {ManagedService.class.getName(), OnOff.class.getName()}, this, regProps);` - `ManagedService` interface is not a Device Function interface;
- `context.registerService(new String[] {DeviceFunction.class.getName(), OnOff.class.getName()}, this, regProps);` - `DeviceFunction` interface is not concrete Device Function.

That one is valid `context.registerService(new String[] {Meter.class.getName(), OnOff.class.getName()}, this, regProps);`. `Meter` and `OnOff` are concrete Device Function interfaces. That rule helps to the applications to find all supported Device Function classes. Otherwise the Device Function services can be accesses, but it's not clear which are the Device Function classes.

The Device Function properties must be integrated according to these rules:

- getter methods must be available for all properties with `META_INFO_PROPERTY_ACCESS_READABLE` access;
- setter methods must be available for all properties with `META_INFO_PROPERTY_ACCESS_WRITABLE` access;
- no methods are required for properties with `META_INFO_PROPERTY_ACCESS_EVENTABLE` access.

The accessor methods must be defined according JavaBeans specification.

The Device Function operations are java methods, which cannot override the property accessor methods. They can have zero or more input arguments and zero or one output argument.

Operation arguments share the same metadata with Device Function properties. The data type can be one of the following types:

- Java primitive type or corresponding reference type.
- `java.lang.String`.
- `Beans`, but the beans properties must use those rules. Java Beans are defined in JavaBeans specification.

- `java.util.Map`s. The keys can be any reference type of Java primitive types or `java.lang.String`. The values must use those rules.
- Arrays of defined types.

The properties and the operation arguments have some common metadata. It's provided with:

- META_INFO_VARIABLE_DESCRIPTION
- META_INFO_VARIABLE_UNIT
- META_INFO_VARIABLE_MIN
- META_INFO_VARIABLE_MAX
- META_INFO_VARIABLE_RESOLUTION
- META_INFO_VARIABLE_VALUES

The access to the Device Function properties is a bitmap value of META_INFO_PROPERTY_ACCESS meta data key. Device Function properties can be accessed in three ways. Any combinations between them are possible:

- META_INFO_PROPERTY_ACCESS_READABLE - available for all properties, which can be read. Device Function must provide a getter method for an access to the property value.
- META_INFO_PROPERTY_ACCESS_WRITABLE - available for all properties, which can be modified. Device Function must provide a setter method for a modification of the property value.
- META_INFO_PROPERTY_ACCESS_EVENTABLE - available for all properties, which can report the property value. DeviceFunctionEvents are sent on property change.

In order to provide common behavior, all Device Functions must follow a set of common rules related to the implementation of their setters, getters, operations and events:

- The setter method must be executed synchronously. If the underlying protocol can return response to the setter call, it must be awaited. It simplifies the property value modifications and doesn't require asynchronous call back.
- The operation method must be executed synchronously. If the underlying protocol can return an operation confirmation or response, they must be awaited. It simplifies the operation execution and doesn't require asynchronous call back.
- The getter must return the last know cached property value. The device implementation is responsible to keep that value up to date. It'll speed up the applications when the Device Function property values are collected. The same cached value can be shared between a few requests instead of a few calls to the real device.
- If a given Device Function operation, getter or setter is not supported, java.lang.UnsupportedOperationException must be thrown. It indicates that Device Function is partially supported.
- The Device Function operations, getters and setters must not override `java.lang.Object` and this interface methods.

Field Summary		Page
String	**META_INFO_OPERATION_ARG_OUT** Meta data key, which value represents the operation output argument metadata.	39
String	**META_INFO_OPERATION_ARGS_IN_PREFIX** Meta data key prefix, which key value represents the operation input argument metadata.	38
String	**META_INFO_OPERATION_DESCRIPTION** Meta data key, which value contains the function operation description.	38
String	**META_INFO_PROPERTY_ACCESS** Meta data key, which value represents the access to the Device Function property.	37
int	**META_INFO_PROPERTY_ACCESS_EVENTABLE** Marks the eventable Device Function properties.	37
int	**META_INFO_PROPERTY_ACCESS_READABLE** Marks the readable Device Function properties.	36
int	**META_INFO_PROPERTY_ACCESS_WRITABLE** Marks the writable Device Function properties.	36
String	**META_INFO_VARIABLE_DESCRIPTION** Meta data key, which value represents the Device Function property or the operation argument description.	37

String	**META_INFO_VARIABLE_MAX** Meta data key, which value represents the Device Function property or the operation argument maximum value.	38
String	**META_INFO_VARIABLE_MIN** Meta data key, which value represents the Device Function property or the operation argument minimum value.	37
String	**META_INFO_VARIABLE_RESOLUTION** Meta data key, which value represents the resolution value of specific range of the Device Function property or the operation argument.	38
String	**META_INFO_VARIABLE_UNIT** Meta data key, which value represents the Device Function property or the operation argument unit.	37
String	**META_INFO_VARIABLE_VALUES** Meta data key, which value represents the Device Function property or the operation argument possible values.	38
String	**PROPERTY_DESCRIPTION** The service property value contains the device function description.	40
String	**PROPERTY_DEVICE_UID** The service property value contains the function device unique identifier.	39
String	**PROPERTY_GROUP_UID** The service property value contains the function group unique identifier.	40
String	**PROPERTY_OPERATION_NAMES** The service property value contains the device function operation names.	40
String	**PROPERTY_PROPERTY_NAMES** The service property value contains the device function property names.	40
String	**PROPERTY_UID** The service property value contains the device function unique identifier.	39

Method Summary		Page
Map	**getOperationMetaData**(String operationName) Provides meta data about the given function operation.	41
Map	**getPropertyMetaData**(String propertyName) Provides meta data about the given function property.	40

Field Detail

META_INFO_PROPERTY_ACCESS_READABLE

```
public static final int META_INFO_PROPERTY_ACCESS_READABLE = 1
```

Marks the readable Device Function properties. The flag can be used as a part of bitmap value of META_INFO_PROPERTY_ACCESS. The readable access mandates Device Function to provide a property getter method.

META_INFO_PROPERTY_ACCESS_WRITABLE

```
public static final int META_INFO_PROPERTY_ACCESS_WRITABLE = 2
```

Marks the writable Device Function properties. The flag can be used as a part of bitmap value of META_INFO_PROPERTY_ACCESS. The writable access mandates Device Function to provide a property setter method.

META_INFO_PROPERTY_ACCESS_EVENTABLE

```
public static final int META_INFO_PROPERTY_ACCESS_EVENTABLE = 4
```

Marks the eventable Device Function properties. The flag can be used as a part of bitmap value of META_INFO_PROPERTY_ACCESS.

META_INFO_VARIABLE_DESCRIPTION

```
public static final String META_INFO_VARIABLE_DESCRIPTION =
"device.function.variable.description"
```

Meta data key, which value represents the Device Function property or the operation argument description. The property value type is `java.lang.String`.

See Also:
getPropertyMetaData(String)

META_INFO_VARIABLE_UNIT

```
public static final String META_INFO_VARIABLE_UNIT = "device.function.variable.unit"
```

Meta data key, which value represents the Device Function property or the operation argument unit. The property value type is `java.lang.String`. These rules must be applied to unify the representation:

- SI units (The International System of Units) must be used where it's applicable.
- The unit must use Unicode symbols normalized with NFKD (Compatibility Decomposition) normalization form. (see Unicode Standard Annex #15, Unicode Normalization Forms)

For example, degrees Celsius will not be represent as U+2103 (degree celsius), but will be U+00B0 degree sign + U+0043 latin capital letter c.

See Also:
getPropertyMetaData(String)

META_INFO_PROPERTY_ACCESS

```
public static final String META_INFO_PROPERTY_ACCESS = "device.function.property.access"
```

Meta data key, which value represents the access to the Device Function property. The property value is a bitmap of `Integer` type. The bitmap can be any combination of:

- META_INFO_PROPERTY_ACCESS_READABLE
- META_INFO_PROPERTY_ACCESS_WRITABLE
- META_INFO_PROPERTY_ACCESS_EVENTABLE

For example, value Integer(3) means that the property is readable and writable, but not eventable.

See Also:
getPropertyMetaData(String)

META_INFO_VARIABLE_MIN

```
public static final String META_INFO_VARIABLE_MIN = "device.function.variable.min"
```

Meta data key, which value represents the Device Function property or the operation argument minimum value. The property value type depends on the property or argument type.

See Also:
getPropertyMetaData(String)

META_INFO_VARIABLE_MAX

```
public static final String META_INFO_VARIABLE_MAX = "device.function.variable.max"
```

Meta data key, which value represents the Device Function property or the operation argument maximum value. The property value type depends on the property or argument type.

See Also:
getPropertyMetaData(String)

META_INFO_VARIABLE_RESOLUTION

```
public static final String META_INFO_VARIABLE_RESOLUTION =
"device.function.variable.resolution"
```

Meta data key, which value represents the resolution value of specific range of the Device Function property or the operation argument. The property value type depends on the property or argument type. For example, if the range is [0, 100], the resolution can be 10. That's the difference between two values in series.

See Also:
getPropertyMetaData(String)

META_INFO_VARIABLE_VALUES

```
public static final String META_INFO_VARIABLE_VALUES = "device.function.variable.values"
```

Meta data key, which value represents the Device Function property or the operation argument possible values. The property value type is `java.util.Map`, where the keys are the possible values and the values are their string representations.

See Also:
getPropertyMetaData(String)

META_INFO_OPERATION_DESCRIPTION

```
public static final String META_INFO_OPERATION_DESCRIPTION =
"device.function.operation.description"
```

Meta data key, which value contains the function operation description. The property value type is `java.lang.String`.

See Also:
getOperationMetaData(String)

META_INFO_OPERATION_ARGS_IN_PREFIX

```
public static final String META_INFO_OPERATION_ARGS_IN_PREFIX =
"device.function.operation.arguments.in."
```

Meta data key prefix, which key value represents the operation input argument metadata. The property value type is `java.util.Map`. The value map key can be one of:

- ≅ META_INFO_VARIABLE_DESCRIPTION
- ≅ META_INFO_VARIABLE_UNIT
- ≅ META_INFO_VARIABLE_MIN
- ≅ META_INFO_VARIABLE_MAX
- ≅ META_INFO_VARIABLE_RESOLUTION
- ≅ META_INFO_VARIABLE_VALUES
- ≅ custom key

The prefix must be used in the form:

operation input argument name ::= value of META_INFO_OPERATION_ARGS_IN_PREFIXargument-index

argument-index - input argument index. For example, device.function.operation.arguments.in.1 can be used for the first operation input argument.

See Also:
getOperationMetaData(String)

META_INFO_OPERATION_ARG_OUT

```
public static final String META_INFO_OPERATION_ARG_OUT = "device.function.operation.argument.out"
```

Meta data key, which value represents the operation output argument metadata. The property value type is `java.util.Map`. The value map key can be one of:

- ≅ META_INFO_VARIABLE_DESCRIPTION
- ≅ META_INFO_VARIABLE_UNIT
- ≅ META_INFO_VARIABLE_MIN
- ≅ META_INFO_VARIABLE_MAX
- ≅ META_INFO_VARIABLE_RESOLUTION
- ≅ META_INFO_VARIABLE_VALUES
- ≅ custom key

See Also:
getOperationMetaData(String)

PROPERTY_UID

```
public static final String PROPERTY_UID = "device.function.UID"
```

The service property value contains the device function unique identifier. It's a mandatory property. The value type is `java.lang.String`. To simplify the unique identifier generation, the property value must follow the rule:

function UID ::= device-id ':' function-id

function UID - device function unique identifier

device-id - the value of the FunctionalDevice.PROPERTY_UID Functional Device service property

function-id - device function identifier in the scope of the device

---

PROPERTY_DEVICE_UID

```
public static final String PROPERTY_DEVICE_UID = "device.function.device.UID"
```

The service property value contains the function device unique identifier. The function belongs to this device. It's an optional property. The value type is `java.lang.String`.

PROPERTY_GROUP_UID

```
public static final String PROPERTY_GROUP_UID = "device.function.group.UID"
```

The service property value contains the function group unique identifier. The function belongs to this functional group. It's an optional property. The value type is `java.lang.String`.

PROPERTY_DESCRIPTION

```
public static final String PROPERTY_DESCRIPTION = "device.function.description"
```

The service property value contains the device function description. It's an optional property. The value type is `java.lang.String`.

PROPERTY_OPERATION_NAMES

```
public static final String PROPERTY_OPERATION_NAMES = "device.function.operation.names"
```

The service property value contains the device function operation names. It's an optional property. The value type is `java.lang.String[]`. It's not possible to exist two or more Device Function operations with the same name i.e. the operation overloading is not allowed.

PROPERTY_PROPERTY_NAMES

```
public static final String PROPERTY_PROPERTY_NAMES = "device.function.property.names"
```

The service property value contains the device function property names. It's an optional property. The value type is `java.lang.String[]`. It's not possible to exist two or more Device Function properties with the same name.

Method Detail

getPropertyMetaData

```
Map getPropertyMetaData(String propertyName)
                throws IllegalArgumentException
```

Provides meta data about the given function property. The keys of the `java.util.Map` result must be of `java.lang.String` type. Possible keys:

- META_INFO_VARIABLE_DESCRIPTION
- META_INFO_PROPERTY_ACCESS
- META_INFO_VARIABLE_UNIT
- META_INFO_VARIABLE_MIN
- META_INFO_VARIABLE_MAX
- META_INFO_VARIABLE_RESOLUTION
- META_INFO_VARIABLE_VALUES
- custom key

This method must continue to return the operation names after the device service has been unregistered.

Parameters:
`propertyName` - The function property name, which meta data is requested.
Returns:
The property meta data for the given property name. `null` if the property meta data is not supported.
Throws:
`IllegalArgumentException` - If the function property with the specified name is not supported.

---

getOperationMetaData

```
Map getOperationMetaData(String operationName)
                  throws IllegalArgumentException
```

Provides meta data about the given function operation. The keys of the `java.util.Map` result must be of `java.lang.String` type. Possible keys:

- ≅ META_INFO_OPERATION_DESCRIPTION
- ≅ META_INFO_OPERATION_ARG_OUT
- ≅ Different input arguments with prefix META_INFO_OPERATION_ARGS_IN_PREFIX
- ≅ custom key

This method must continue to return the operation names after the device service has been unregistered.

Parameters:
`operationName` - The function operation name, which meta data is requested.

Returns:
The operation meta data for the given operation name. `null` if the operation meta data is not supported.

Throws:
`IllegalArgumentException` - If the function operation with the specified name is not supported.

Class DeviceFunctionEvent

org.osgi.service.functionaldevice

```
java.lang.Object
  └org.osgi.service.event.Event
      └org.osgi.service.functionaldevice.DeviceFunctionEvent
```

```
final public class DeviceFunctionEvent
extends org.osgi.service.event.Event
```

Asynchronous event, which marks a Device Function property value modification. The event can be triggered when there is a new property value, but it's possible to have events in series with no value change. The event properties must contain all device properties and:

- PROPERTY_DEVICE_FUNCTION - the event source function.
- PROPERTY_DEVICE_FUNCTION_PROPERTY_NAME - the property name.
- PROPERTY_DEVICE_FUNCTION_PROPERTY_VALUE - the property value.

Field Summary		Page
static String	**EVENT_CLASS** Represents the event class.	43
static String	**EVENT_PACKAGE** Represents the event package.	43
static String	**PROPERTY_DEVICE_FUNCTION** Represents an event property key for Device Function.	43
static String	**PROPERTY_DEVICE_FUNCTION_PROPERTY_NAME** Represents an event property key for the Device Function property name.	43
static String	**PROPERTY_DEVICE_FUNCTION_PROPERTY_VALUE** Represents an event property key for the Device Function property value.	43
static String	**TOPIC_PROPERTY_CHANGED** Represents the event topic for the Device Function property changed.	43

Constructor Summary	Page
DeviceFunctionEvent(String topic, Dictionary properties) Constructs a new event with the specified topic and properties.	43
DeviceFunctionEvent(String topic, Map properties) Constructs a new event with the specified topic and properties.	44

Method Summary		Page
String	**getFunction**() Returns the property value change source function.	44
String	**getPropertyName**() Returns the property name.	44
Object	**getPropertyValue**() Returns the property value.	44

Methods inherited from class org.osgi.service.event.Event
equals, getProperty, getPropertyNames, getTopic, hashCode, matches, toString

Field Detail

EVENT_PACKAGE

```
public static final String EVENT_PACKAGE = "org/osgi/services/abstractdevice/"
```

Represents the event package. That constant can be useful for the event handlers depending on the event filters.

EVENT_CLASS

```
public static final String EVENT_CLASS = "org/osgi/services/abstractdevice/DeviceFunctionEvent/"
```

Represents the event class. That constant can be useful for the event handlers depending on the event filters.

TOPIC_PROPERTY_CHANGED

```
public static final String TOPIC_PROPERTY_CHANGED = "org/osgi/services/abstractdevice/DeviceFunctionEvent/PROPERTY_CHANGED"
```

Represents the event topic for the Device Function property changed.

PROPERTY_DEVICE_FUNCTION

```
public static final String PROPERTY_DEVICE_FUNCTION = "device.function"
```

Represents an event property key for Device Function. The property value type is `java.lang.String`. The value represents the property value change source function.

PROPERTY_DEVICE_FUNCTION_PROPERTY_NAME

```
public static final String PROPERTY_DEVICE_FUNCTION_PROPERTY_NAME = "device.function.property.name"
```

Represents an event property key for the Device Function property name. The property value type is `java.lang.String`. The value represents the property name.

PROPERTY_DEVICE_FUNCTION_PROPERTY_VALUE

```
public static final String PROPERTY_DEVICE_FUNCTION_PROPERTY_VALUE = "device.function.property.value"
```

Represents an event property key for the Device Function property value. The property value type depends on the property type. The value represents the property value.

Constructor Detail

DeviceFunctionEvent

```
public DeviceFunctionEvent(String topic,
                           Dictionary properties)
```

Constructs a new event with the specified topic and properties.

Parameters:
`topic` - The event topic.
`properties` - The event properties.

DeviceFunctionEvent

```
public DeviceFunctionEvent(String topic,
                           Map properties)
```

Constructs a new event with the specified topic and properties.

Parameters:
`topic` - The event topic.
`properties` - The event properties.

Method Detail

getFunction

```
public String getFunction()
```

Returns the property value change source function. The value is same as the value of PROPERTY_DEVICE_FUNCTION property.

Returns:
The property value change source function.

---

getPropertyName

```
public String getPropertyName()
```

Returns the property name. The value is same as the value of PROPERTY_DEVICE_FUNCTION_PROPERTY_NAME.

Returns:
The property name.

---

getPropertyValue

```
public Object getPropertyValue()
```

Returns the property value. The value is same as the value of PROPERTY_DEVICE_FUNCTION_PROPERTY_VALUE.

Returns:
The property value.

Interface FunctionalDevice

org.osgi.service.functionaldevice

public interface **FunctionalDevice**

Represents the functional device in the OSGi service registry. Note that `FunctionalDevice` services are registered last. Before their registration, there is `DeviceFunction` and `FunctionalGroup` registration. The reverse order is used when the services are unregistered. `FunctionalDevice` services are unregistered first before `FunctionalGroup` and `DeviceFunction` services.

Field Summary		Page
String	**DEVICE_CATEGORY** Constant for the value of the `org.osgi.service.device.Constants.DEVICE_CATEGORY` service property.	46
String	**PROPERTY_COMMUNICATION** The service property value contains the device communication possibility.	47
String	**PROPERTY_DESCRIPTION** The service property value contains the device description.	49
String	**PROPERTY_FIRMWARE_VENDOR** The service property value contains the device firmware vendor.	48
String	**PROPERTY_FIRMWARE_VERSION** The service property value contains the device firmware version.	49
String	**PROPERTY_FUNCTION_UIDS** The service property value contains the function unique identifiers of the supported `DeviceFunctions`.	47
String	**PROPERTY_GROUP_UIDS** The service property value contains the functional group unique identifiers of the supported `FunctionalGroups`.	47
String	**PROPERTY_HARDWARE_VENDOR** The service property value contains the device hardware vendor.	48
String	**PROPERTY_HARDWARE_VERSION** The service property value contains the device hardware version.	48
String	**PROPERTY_MODEL** The service property value contains the device model.	49
String	**PROPERTY_NAME** The service property value contains the device name.	48
String	**PROPERTY_REFERENCE_UIDS** The service property value contains the reference device unique identifiers.	47
String	**PROPERTY_SERIAL_NUMBER** The service property value contains the device serial number.	49
String	**PROPERTY_STATUS** The service property value contains the device status.	48
String	**PROPERTY_STATUS_DETAIL** The service property value contains the device status detail.	48
String	**PROPERTY_TYPES** The service property value contains the device types like DVD, TV etc.	49
String	**PROPERTY_UID** The service property value contains the device unique identifier.	47
int	**STATUS_DETAIL_CONFIGURATION_NOT_APPLIED** Device status detail indicates that the device configuration is not applied.	51
int	**STATUS_DETAIL_CONNECTING** Device status detail indicates that the device is currently connecting to the network.	50

int	**STATUS_DETAIL_DEVICE_BROKEN** Device status detail indicates that the device is broken.	51
int	**STATUS_DETAIL_DEVICE_COMMUNICATION_ERROR** Device status detail indicates that the device communication is problematic.	51
int	**STATUS_DETAIL_DEVICE_DATA_INSUFFICIENT** Device status detail indicates that the device doesn't provide enough information and cannot be determined.	51
int	**STATUS_DETAIL_DEVICE_NOT_ACCESSIBLE** Device status detail indicates that the device is not accessible and further communication is not possible.	51
int	**STATUS_DETAIL_ERROR_APPLYING_CONFIGURATION** Device status detail indicates that the device cannot be configured.	51
int	**STATUS_DETAIL_IN_DUTY_CYCLE** Device status detail indicates that the device is in duty cycle.	51
int	**STATUS_DETAIL_INITIALIZING** Device status detail indicates that the device is currently in process of initialization.	50
int	**STATUS_DISABLED** Device status indicates that the device is currently disabled.	50
int	**STATUS_NOT_CONFIGURED** Device status indicates that the device is currently not configured.	50
int	**STATUS_NOT_INITIALIZED** Device status indicates that the device is currently not initialized.	50
int	**STATUS_OFFLINE** Device status indicates that the device is currently not available for operations.	49
int	**STATUS_ONLINE** Device status indicates that the device is currently available for operations.	50
int	**STATUS_PROCESSING** Device status indicates that the device is currently busy with an operation.	50
int	**STATUS_REMOVED** Device status indicates that the device is removed.	49
String	**TYPE_PERIPHERAL** Device type indicates that the device is peripheral.	52

Method Summary		Page
void	**disable**() Disables this device.	53
void	**enable**() Enables this device.	54
Object	**getProperty**(String propName) Returns the current value of the specified property.	52
void	**remove**() Removes this device.	53
void	**setProperties**(String[] propNames, Object[] propValues) Sets the given property names to the given property values.	53
void	**setProperty**(String propName, Object propValue) Sets the given property name to the given property value.	52

Field Detail

DEVICE_CATEGORY

```
public static final String DEVICE_CATEGORY = "FunctionalDevice"
```

Constant for the value of the `org.osgi.service.device.Constants.DEVICE_CATEGORY` service property. That category is used by all abstract devices.

See Also:
`org.osgi.service.device.Constants.DEVICE_CATEGORY`

PROPERTY_UID

```
public static final String PROPERTY_UID = "functional.device.UID"
```

The service property value contains the device unique identifier. It's a mandatory property. The value type is `java.lang.String`. The property value cannot be set. To simplify the unique identifier generation, the property value must follow the rule:

UID ::= communication-type ':' device-id

UID - device unique identifier

communication-type - the value of the `PROPERTY_COMMUNICATION` service property

device-id - device unique identifier in the scope of the communication type

---

PROPERTY_FUNCTION_UIDS

```
public static final String PROPERTY_FUNCTION_UIDS = "functional.device.function.UIDs"
```

The service property value contains the function unique identifiers of the supported `DeviceFunctions`. It's an optional property. The value type is `java.lang.String[]`.

PROPERTY_GROUP_UIDS

```
public static final String PROPERTY_GROUP_UIDS = "functional.device.group.UIDs"
```

The service property value contains the functional group unique identifiers of the supported `FunctionalGroups`. It's an optional property. The value type is `java.lang.String[]`.

PROPERTY_REFERENCE_UIDS

```
public static final String PROPERTY_REFERENCE_UIDS = "functional.device.reference.UIDs"
```

The service property value contains the reference device unique identifiers. It's an optional property. The value type is `java.lang.String[]`. The property value cannot be set. It can be used to represent different relationships between the devices. For example, the ZigBee controller can have a reference to the USB dongle.

PROPERTY_COMMUNICATION

```
public static final String PROPERTY_COMMUNICATION = "functional.device.communication"
```

The service property value contains the device communication possibility. It can vary depending on the device. On protocol level, it can represent the used protocol. The peripheral device property can explore the used communication interface. It's a mandatory property. The value type is `java.lang.String`. The property value cannot be set.

PROPERTY_NAME

```
public static final String PROPERTY_NAME = "functional.device.name"
```

The service property value contains the device name. It's an optional property. The value type is `java.lang.String`. The property value can be read and set.

PROPERTY_STATUS

```
public static final String PROPERTY_STATUS = "functional.device.status"
```

The service property value contains the device status. It's a mandatory property. The value type is `java.lang.Integer`. The property value cannot be set. The possible values are:

- STATUS_ONLINE
- STATUS_OFFLINE
- STATUS_REMOVED
- STATUS_PROCESSING
- STATUS_DISABLED
- STATUS_NOT_INITIALIZED
- STATUS_NOT_CONFIGURED

PROPERTY_STATUS_DETAIL

```
public static final String PROPERTY_STATUS_DETAIL = "functional.device.status.detail"
```

The service property value contains the device status detail. It holds the reason for the current device status. It's an optional property. The value type is `java.lang.Integer`. The property value cannot be set. There are two value categories:

- positive values i.e. > 0
- - Those values contain details related to the current status. Examples: STATUS_DETAIL_CONNECTING and STATUS_DETAIL_INITIALIZING.
- negative values i.e. 0
- - Those values contain errors related to the current status. Examples: STATUS_DETAIL_CONFIGURATION_NOT_APPLIED, STATUS_DETAIL_DEVICE_BROKEN and STATUS_DETAIL_DEVICE_COMMUNICATION_ERROR.

PROPERTY_HARDWARE_VENDOR

```
public static final String PROPERTY_HARDWARE_VENDOR = "functional.device.hardware.vendor"
```

The service property value contains the device hardware vendor. It's an optional property. The value type is `java.lang.String`. The property value can be read and set.

PROPERTY_HARDWARE_VERSION

```
public static final String PROPERTY_HARDWARE_VERSION = "functional.device.hardware.version"
```

The service property value contains the device hardware version. It's an optional property. The value type is `java.lang.String`. The property value can be read and set.

PROPERTY_FIRMWARE_VENDOR

```
public static final String PROPERTY_FIRMWARE_VENDOR = "functional.device.firmware.vendor"
```

The service property value contains the device firmware vendor. It's an optional property. The value type is `java.lang.String`. The property value can be read and set.

PROPERTY_FIRMWARE_VERSION

```
public static final String PROPERTY_FIRMWARE_VERSION = "functional.device.firmware.version"
```

The service property value contains the device firmware version. It's an optional property. The value type is `java.lang.String`. The property value can be read and set.

PROPERTY_TYPES

```
public static final String PROPERTY_TYPES = "functional.device.types"
```

The service property value contains the device types like DVD, TV etc. It's an optional property. The value type is `java.lang.String[]`. The property value can be read and set.

PROPERTY_MODEL

```
public static final String PROPERTY_MODEL = "functional.device.model"
```

The service property value contains the device model. It's an optional property. The value type is `java.lang.String`. The property value can be read and set.

PROPERTY_SERIAL_NUMBER

```
public static final String PROPERTY_SERIAL_NUMBER = "functional.device.serial.number"
```

The service property value contains the device serial number. It's an optional property. The value type is `java.lang.String`. The property value can be read and set.

PROPERTY_DESCRIPTION

```
public static final String PROPERTY_DESCRIPTION = "functional.device.description"
```

The service property value contains the device description. It's an optional property. The value type is `java.lang.String`. The property value can be read and set.

STATUS_REMOVED

```
public static final int STATUS_REMOVED = 0
```

Device status indicates that the device is removed. It can be used as a value of PROPERTY_STATUS service property.

STATUS_OFFLINE

```
public static final int STATUS_OFFLINE = 2
```

Device status indicates that the device is currently not available for operations. It can be used as a value of PROPERTY_STATUS service property.

STATUS_ONLINE

```
public static final int STATUS_ONLINE = 3
```

Device status indicates that the device is currently available for operations. It can be used as a value of PROPERTY_STATUS service property.

STATUS_PROCESSING

```
public static final int STATUS_PROCESSING = 5
```

Device status indicates that the device is currently busy with an operation. It can be used as a value of PROPERTY_STATUS service property.

STATUS_DISABLED

```
public static final int STATUS_DISABLED = 6
```

Device status indicates that the device is currently disabled. It can be used as a value of PROPERTY_STATUS service property.

STATUS_NOT_INITIALIZED

```
public static final int STATUS_NOT_INITIALIZED = 7
```

Device status indicates that the device is currently not initialized. Some protocols don't provide device information right after the device is connected. The device can be initialized later when it's awakened. It can be used as a value of PROPERTY_STATUS service property.

STATUS_NOT_CONFIGURED

```
public static final int STATUS_NOT_CONFIGURED = 8
```

Device status indicates that the device is currently not configured. The device can require additional actions to become completely connected to the network. It can be used as a value of PROPERTY_STATUS service property.

STATUS_DETAIL_CONNECTING

```
public static final int STATUS_DETAIL_CONNECTING = 1
```

Device status detail indicates that the device is currently connecting to the network. It can be used as a value of PROPERTY_STATUS_DETAIL service property. The device status must be STATUS_PROCESSING.

STATUS_DETAIL_INITIALIZING

```
public static final int STATUS_DETAIL_INITIALIZING = 2
```

Device status detail indicates that the device is currently in process of initialization. It can be used as a value of PROPERTY_STATUS_DETAIL service property. The device status must be STATUS_PROCESSING.

STATUS_DETAIL_CONFIGURATION_NOT_APPLIED

```
public static final int STATUS_DETAIL_CONFIGURATION_NOT_APPLIED = -1
```

Device status detail indicates that the device configuration is not applied. It can be used as a value of PROPERTY_STATUS_DETAIL service property. The device status must be STATUS_NOT_CONFIGURED.

STATUS_DETAIL_DEVICE_BROKEN

```
public static final int STATUS_DETAIL_DEVICE_BROKEN = -2
```

Device status detail indicates that the device is broken. It can be used as a value of PROPERTY_STATUS_DETAIL service property. The device status must be STATUS_OFFLINE.

STATUS_DETAIL_DEVICE_COMMUNICATION_ERROR

```
public static final int STATUS_DETAIL_DEVICE_COMMUNICATION_ERROR = -3
```

Device status detail indicates that the device communication is problematic. It can be used as a value of PROPERTY_STATUS_DETAIL service property. The device status must be STATUS_ONLINE or STATUS_NOT_INITIALIZED.

STATUS_DETAIL_DEVICE_DATA_INSUFFICIENT

```
public static final int STATUS_DETAIL_DEVICE_DATA_INSUFFICIENT = -4
```

Device status detail indicates that the device doesn't provide enough information and cannot be determined. It can be used as a value of PROPERTY_STATUS_DETAIL service property. The device status must be STATUS_NOT_INITIALIZED.

STATUS_DETAIL_DEVICE_NOT_ACCESSIBLE

```
public static final int STATUS_DETAIL_DEVICE_NOT_ACCESSIBLE = -5
```

Device status detail indicates that the device is not accessible and further communication is not possible. It can be used as a value of PROPERTY_STATUS_DETAIL service property. The device status must be STATUS_OFFLINE.

STATUS_DETAIL_ERROR_APPLYING_CONFIGURATION

```
public static final int STATUS_DETAIL_ERROR_APPLYING_CONFIGURATION = -6
```

Device status detail indicates that the device cannot be configured. It can be used as a value of PROPERTY_STATUS_DETAIL service property. The device status must be STATUS_NOT_CONFIGURED.

STATUS_DETAIL_IN_DUTY_CYCLE

```
public static final int STATUS_DETAIL_IN_DUTY_CYCLE = -7
```

Device status detail indicates that the device is in duty cycle. It can be used as a value of PROPERTY_STATUS_DETAIL service property. The device status must be STATUS_OFFLINE.

TYPE_PERIPHERAL

```
public static final String TYPE_PERIPHERAL = "type.peripheral"
```

Device type indicates that the device is peripheral. Usually, those devices are base and contains some meta information. It can be used as a value of PROPERTY_TYPES service property.

Method Detail

getProperty

```
Object getProperty(String propName)
            throws IllegalArgumentException
```

Returns the current value of the specified property. The method will return the same value as `org.osgi.framework.ServiceReference.getProperty(String)` for the service reference of this device.

This method must continue to return property values after the device service has been unregistered.

Parameters:
`propName` - The property name.
Returns:
The property value
Throws:
`IllegalArgumentException` - If the property name cannot be mapped to value.

---

setProperty

```
void setProperty(String propName,
                 Object propValue)
          throws FunctionalDeviceException,
                 IllegalArgumentException,
                 UnsupportedOperationException,
                 SecurityException,
                 IllegalStateException
```

Sets the given property name to the given property value. The method can be used for:

- Update - if the property name exists, the value will be updated.
- Add - if the property name doesn't exists, a new property will be added.
- Remove - if the property name exists and the given property value is `null`, then the property will be removed.

Parameters:
`propName` - The property name.
`propValue` - The property value.
Throws:
`FunctionalDeviceException` - If an operation error is available.
`IllegalArgumentException` - If the property name or value aren't correct.
`UnsupportedOperationException` - If the operation is not supported over this device.
`SecurityException` - If the caller does not have the appropriate `FunctionalDevicePermission[this device, FunctionalDevicePermission.ACTION_PROPERTY]` and the Java Runtime Environment supports permissions.
`IllegalStateException` - If this device service object has already been unregistered.

---

setProperties

```
void setProperties(String[] propNames,
                   Object[] propValues)
            throws FunctionalDeviceException,
                   IllegalArgumentException,
                   UnsupportedOperationException,
                   SecurityException,
                   IllegalStateException
```

Sets the given property names to the given property values. The method is similar to setProperty(String, Object), but can update all properties with one bulk operation.

Parameters:
propNames - The property names.
propValues - The property values.

Throws:
FunctionalDeviceException - If an operation error is available.
IllegalArgumentException - If the property values or names aren't correct.
UnsupportedOperationException - If the operation is not supported over this device.
SecurityException - If the caller does not have the appropriate FunctionalDevicePermission[this device, FunctionalDevicePermission.ACTION_PROPERTY] and the Java Runtime Environment supports permissions.
IllegalStateException - If this device service object has already been unregistered.

---

remove

```
void remove()
     throws FunctionalDeviceException,
            UnsupportedOperationException,
            SecurityException,
            IllegalStateException
```

Removes this device. The method must synchronously remove the device from the device network.

Throws:
FunctionalDeviceException - If an operation error is available.
UnsupportedOperationException - If the operation is not supported over this device.
SecurityException - If the caller does not have the appropriate FunctionalDevicePermission[this device, FunctionalDevicePermission.ACTION_REMOVE] and the Java Runtime Environment supports permissions.
IllegalStateException - If this device service object has already been unregistered.

---

disable

```
void disable()
      throws FunctionalDeviceException,
             UnsupportedOperationException,
             IllegalStateException
```

Disables this device. The disabled device status is set to STATUS_DISABLED. The device is not available for operations.

Throws:
FunctionalDeviceException - If an operation error is available.
UnsupportedOperationException - If the operation is not supported over this device.
IllegalStateException - If this device service object has already been unregistered.
SecurityException - If the caller does not have the appropriate FunctionalDevicePermission[this device, FunctionalDevicePermission.ACTION_DISABLE] and the Java Runtime Environment supports permissions.

---

enable

```
void enable()
     throws FunctionalDeviceException,
            UnsupportedOperationException,
            SecurityException,
            IllegalStateException
```

Enables this device. The device is available for operations.

Throws:

`FunctionalDeviceException` - If an operation error is available.
`UnsupportedOperationException` - If the operation is not supported over this device.
`SecurityException` - If the caller does not have the appropriate `FunctionalDevicePermission[this device, FunctionalDevicePermission.ACTION_ENABLE]` and the Java Runtime Environment supports permissions.
`IllegalStateException` - If this device service object has already been unregistered.

Class FunctionalDeviceException

org.osgi.service.functionaldevice

```
java.lang.Object
  └ java.lang.Throwable
      └ java.lang.Exception
          └ org.osgi.service.functionaldevice.FunctionalDeviceException
```

All Implemented Interfaces:
Serializable

```
public class FunctionalDeviceException
extends Exception
```

Thrown to indicate that there is a device operation fail. The error reason can be located with `getCode()` method.

Field Summary		Page
static int	**CODE_COMMUNICATION_ERROR** An exception code indicates that there is an error in the communication.	55
static int	**CODE_DEVICE_NOT_INITIALIZED** An exception code indicates that the device is not initialized.	56
static int	**CODE_NO_DATA** An exception code indicates that the requested value is currently not available.	56
static int	**CODE_TIMEOUT** An exception code indicates that the response is not produced within a given timeout.	56
static int	**CODE_UNKNOWN** An exception code indicates that the error is unknown.	55

Constructor Summary	Page
FunctionalDeviceException()	56

Method Summary		Page
Throwable	**getCause**() Returns the cause for this throwable or `null` if the cause is missing.	56
int	**getCode**() Returns the exception error code.	56

Field Detail

CODE_UNKNOWN

```
public static final int CODE_UNKNOWN = 1
```

An exception code indicates that the error is unknown.

CODE_COMMUNICATION_ERROR

```
public static final int CODE_COMMUNICATION_ERROR = 2
```

An exception code indicates that there is an error in the communication.

CODE_TIMEOUT

```
public static final int CODE_TIMEOUT = 3
```

An exception code indicates that the response is not produced within a given timeout.

CODE_DEVICE_NOT_INITIALIZED

```
public static final int CODE_DEVICE_NOT_INITIALIZED = 4
```

An exception code indicates that the device is not initialized. It indicates that the device status is FunctionalDevice.STATUS_NOT_INITIALIZED.

CODE_NO_DATA

```
public static final int CODE_NO_DATA = 5
```

An exception code indicates that the requested value is currently not available.

Constructor Detail

FunctionalDeviceException

```
public FunctionalDeviceException()
```

Method Detail

getCode

```
public int getCode()
```

Returns the exception error code. It indicates the reason for this error.

Returns:
An exception code.

getCause

```
public Throwable getCause()
```

Returns the cause for this throwable or `null` if the cause is missing. The cause can be protocol specific exception with an appropriate message and error code.

Overrides:
`getCause` in class `Throwable`
Returns:
An throwable cause.

Class FunctionalDevicePermission

org.osgi.service.functionaldevice

```
java.lang.Object
  └ java.security.Permission
      └ java.security.BasicPermission
          └ org.osgi.service.functionaldevice.FunctionalDevicePermission
```

All Implemented Interfaces:
Guard, Serializable

```
final public class FunctionalDevicePermission
extends BasicPermission
```

A bundle's authority to perform specific privileged administrative operations on the devices. The actions for this permission are:

Action	Method
ACTION_REMOVE	FunctionalDevice.remove()
ACTION_ENABLE	FunctionalDevice.enable()
ACTION_DISABLE	FunctionalDevice.disable()
ACTION_PROPERTY	FunctionalDevice.setProperty(String, Object) FunctionalDevice.setProperties(String[], Object[])

The name of the permission is a filter based. See OSGi Core Specification, Filter Based Permissions. The filter gives an access to all device service properties. The service property names are case insensitive. The filter attribute names are processed in a case insensitive manner.

Field Summary		Page
static String	**ACTION_DISABLE** A permission action to disable the device.	58
static String	**ACTION_ENABLE** A permission action to enable the device.	58
static String	**ACTION_PROPERTY** A permission action to modify the device properties.	58
static String	**ACTION_REMOVE** A permission action to remove the device.	58

Constructor Summary	Page
FunctionalDevicePermission(String filter, String actions) Creates a new FunctionalDevicePermission with the given filter and actions.	58
FunctionalDevicePermission(FunctionalDevice device, String actions) Creates a new FunctionalDevicePermission with the given device and actions.	59

Method Summary		Page
boolean	**equals**(Object obj) Two FunctionalDevicePermission instances are equal if: ≅ represents the same filter and actions ≅ represents the same device and actions	59
String	**getActions**() Returns the canonical string representation of the actions.	59
int	**hashCode**() Returns the hash code value for this object.	59

boolean	**implies**(Permission p) Determines if the specified permission is implied by this object.	60
PermissionCollection	**newPermissionCollection**() Returns a new PermissionCollection suitable for storing FunctionalDevicePermission instances.	60

Field Detail

ACTION_ENABLE

```
public static final String ACTION_ENABLE = "enable"
```

A permission action to enable the device.

ACTION_DISABLE

```
public static final String ACTION_DISABLE = "disable"
```

A permission action to disable the device.

ACTION_PROPERTY

```
public static final String ACTION_PROPERTY = "property"
```

A permission action to modify the device properties.

ACTION_REMOVE

```
public static final String ACTION_REMOVE = "remove"
```

A permission action to remove the device.

Constructor Detail

FunctionalDevicePermission

```
public FunctionalDevicePermission(String filter,
                                  String actions)
```

Creates a new FunctionalDevicePermission with the given filter and actions. The constructor must only be used to create a permission that is going to be checked.

An filter example: (abstract.device.hardware.vendor=acme)

An action list example: property, remove

Parameters:

filter - A filter expression that can use any device service property. The filter attribute names are processed in a case insensitive manner. A special value of "*" can be used to match akk devices.
actions - A comma-separated list of ACTION_DISABLE, ACTION_ENABLE ACTION_PROPERTY and ACTION_REMOVE. Any combinations are allowed.

Throws:

IllegalArgumentException - If the filter syntax is not correct or invalid actions are specified.

FunctionalDevicePermission

```
public FunctionalDevicePermission(FunctionalDevice device,
                                  String actions)
```

Creates a new `FunctionalDevicePermission` with the given device and actions. The permission must be used for the security checks like:

`securityManager.checkPermission(new FunctionalDevicePermission(this, "remove"));` . The permissions constructed by this constructor must not be added to the `FunctionalDevicePermission` permission collections.

Parameters:
`device` - The permission device.
`actions` - A comma-separated list of `ACTION_DISABLE`, `ACTION_ENABLE` `ACTION_PROPERTY` and `ACTION_REMOVE`. Any combinations are allowed.

Method Detail

equals

```
public boolean equals(Object obj)
```

Two `FunctionalDevicePermission` instances are equal if:

- ≅ represents the same filter and actions
- ≅ represents the same device and actions

Overrides:
`equals` in class `BasicPermission`
Parameters:
`obj` - The object being compared for equality with this object.
Returns:
`true` if two permissions are equal, `false` otherwise.

---

hashCode

```
public int hashCode()
```

Returns the hash code value for this object.

Overrides:
`hashCode` in class `BasicPermission`
Returns:
Hash code value for this object.

---

getActions

```
public String getActions()
```

Returns the canonical string representation of the actions. Always returns present actions in the following order: `ACTION_DISABLE`, `ACTION_ENABLE` `ACTION_PROPERTY`, `ACTION_REMOVE`.

Overrides:
`getActions` in class `BasicPermission`
Returns:
The canonical string representation of the actions.

---

implies

```
public boolean implies(Permission p)
```

Determines if the specified permission is implied by this object. The method will throw an exception if the specified permission was not constructed by FunctionalDevicePermission(FunctionalDevice, String). Returns true if the specified permission is a FunctionalDevicePermission and this permission filter matches the specified permission device properties.

Overrides:
implies in class BasicPermission
Parameters:
p - The permission to be implied. It must be constructed by FunctionalDevicePermission(FunctionalDevice, String).
Returns:
true if the specified permission is implied by this permission, false otherwise.
Throws:
IllegalArgumentException - If the specified permission is not constructed by FunctionalDevicePermission(FunctionalDevice, String).

---

newPermissionCollection

```
public PermissionCollection newPermissionCollection()
```

Returns a new PermissionCollection suitable for storing FunctionalDevicePermission instances.

Overrides:
newPermissionCollection in class BasicPermission
Returns:
A new PermissionCollection instance.

Interface FunctionalGroup

org.osgi.service.functionaldevice

```
public interface FunctionalGroup
```

The functional group can unite similar functionality based on the group type. The grouping is optionally supported by the `FunctionalDevice`s. The `FunctionalGroup` instances are registered in the OSGi service registry. Note that `FunctionalGroup` services are registered after `DeviceFunction` services and before `FunctionalDevice` services. It's possible that `PROPERTY_DEVICE_UID` points to missing service at the moment of the registration. The reverse order is used when the services are unregistered. `FunctionalGroup` services are unregistered after `FunctionalDevice` and before `DeviceFunction` services.

Field Summary		Page
String	**PROPERTY_DESCRIPTION** The service property value contains the functional group description.	61
String	**PROPERTY_DEVICE_UID** The service property value contains the device unique identifier.	62
String	**PROPERTY_FUNCTION_UIDS** The service property value contains the function unique identifiers.	62
String	**PROPERTY_TYPE** The service property value contains the functional group type.	61
String	**PROPERTY_UID** The service property value contains the Device `FunctionalGroup` unique identifier.	61

Field Detail

PROPERTY_TYPE

```
public static final String PROPERTY_TYPE = "functional.group.type"
```

The service property value contains the functional group type. It's a mandatory property. The value type is `java.lang.String`.

PROPERTY_UID

```
public static final String PROPERTY_UID = "functional.group.UID"
```

The service property value contains the Device `FunctionalGroup` unique identifier. It's a mandatory property. The value type is `java.lang.String`. To simplify the unique identifier generation, the property value must follow the rule:

group UID ::= device-id ':' functional-group-id

group UID - device functional group unique identifier

device-id - the value of the `FunctionalDevice.PROPERTY_UID` `FunctionalDevice` service property

functional-group-id - device functional group identifier in the scope of the device

---

PROPERTY_DESCRIPTION

```
public static final String PROPERTY_DESCRIPTION = "functional.group.description"
```

The service property value contains the functional group description. It's an optional property. The value type is `java.lang.String`.

PROPERTY_DEVICE_UID

```
public static final String PROPERTY_DEVICE_UID = "functional.group.device.UID"
```

The service property value contains the device unique identifier. The functional group belongs to this device. It's an optional property. The value type is `java.lang.String`.

PROPERTY_FUNCTION_UIDS

```
public static final String PROPERTY_FUNCTION_UIDS = "functional.group.function.UIDs"
```

The service property value contains the function unique identifiers. Those functions belong to this functional group. It's an optional property. The value type is `java.lang.String[]`.

Package org.osgi.service.functionaldevice.functions

Functional Device Functions 1.0.

See:
[Description](#)

Interface Summary		Page
BinarySensor	`BinarySensor` Device Function provides binary sensor monitoring.	*64*
BinarySwitch	`BinarySwitch` Device Function provides a binary switch control.	*66*
LockUnlock	`LockUnlock` Device Function represents lock and unlock functionality.	*69*
Meter	`Meter` Device Function can measure metering information.	*71*
MultiLevelSensor	`MultiLevelSensor` Device Function provides multi-level sensor monitoring.	*74*
MultiLevelSwitch	`MultiLevelSwitch` Device Function provides multi-level switch control.	*76*
OnOff	`OnOff` Device Function represents turn on and off functionality.	*79*
OpenClose	`OpenClose` Device Function represents open and close functionality.	*81*

Package org.osgi.service.functionaldevice.functions Description

Functional Device Functions 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.functionaldevice.functions; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.functionaldevice.functions; version="[1.0,1.1)"
```

Interface BinarySensor

org.osgi.service.functionaldevice.functions

All Superinterfaces:
DeviceFunction

```
public interface BinarySensor
extends DeviceFunction
```

BinarySensor Device Function provides binary sensor monitoring. It reports its state when an important event is available. The state is accessible with getState() getter. There are no operations. The Device Function name is org.osgi.service.functionaldevice.functions.BinarySensor.

Field Summary		Page
String	**PROPERTY_STATE** Specifies the state property name.	64

Fields inherited from interface org.osgi.service.functionaldevice.DeviceFunction

META_INFO_OPERATION_ARG_OUT, META_INFO_OPERATION_ARGS_IN_PREFIX, META_INFO_OPERATION_DESCRIPTION, META_INFO_PROPERTY_ACCESS, META_INFO_PROPERTY_ACCESS_EVENTABLE, META_INFO_PROPERTY_ACCESS_READABLE, META_INFO_PROPERTY_ACCESS_WRITABLE, META_INFO_VARIABLE_DESCRIPTION, META_INFO_VARIABLE_MAX, META_INFO_VARIABLE_MIN, META_INFO_VARIABLE_RESOLUTION, META_INFO_VARIABLE_UNIT, META_INFO_VARIABLE_VALUES, PROPERTY_DESCRIPTION, PROPERTY_DEVICE_UID, PROPERTY_GROUP_UID, PROPERTY_OPERATION_NAMES, PROPERTY_PROPERTY_NAMES, PROPERTY_UID

Method Summary		Page
boolean	**getState**() Returns the state of the Binary Sensor.	64

Methods inherited from interface org.osgi.service.functionaldevice.DeviceFunction

getOperationMetaData, getPropertyMetaData

Field Detail

PROPERTY_STATE

```
public static final String PROPERTY_STATE = "state"
```

Specifies the state property name. The property can be read with getState() method.

Method Detail

getState

```
boolean getState()
          throws UnsupportedOperationException,
                 IllegalStateException,
                 FunctionalDeviceException
```

Returns the state of the Binary Sensor. It's a getter method for PROPERTY_STATE property.

Returns:
The state of the Binary Sensor.

Throws:

`UnsupportedOperationException` - If the operation is not supported.
`IllegalStateException` - If this device service object has already been unregistered.
`FunctionalDeviceException` - If an operation error is available.

Interface BinarySwitch

org.osgi.service.functionaldevice.functions

All Superinterfaces:
DeviceFunction, OnOff

```
public interface BinarySwitch
extends OnOff
```

BinarySwitch Device Function provides a binary switch control. It extends OnOff Device Function with a property state. The state is accessible with getState() getter. The state can be reversed with toggle() method. STATE_ON state can be reached with the inherited method OnOff.turnOn(). STATE_OFF state can be reached with the inherited method OnOff.turnOff(). The Device Function name is org.osgi.service.functionaldevice.functions.BinarySwitch.

Field Summary		Page
String	**OPERATION_TOGGLE** Specifies the toggle operation name.	67
String	**PROPERTY_STATE** Specifies the state property name.	67
boolean	**STATE_OFF** Specifies the off state property value.	67
boolean	**STATE_ON** Specifies the on state property value.	67

Fields inherited from interface org.osgi.service.functionaldevice.functions.OnOff
OPERATION_TURN_OFF, OPERATION_TURN_ON

Fields inherited from interface org.osgi.service.functionaldevice.DeviceFunction
META_INFO_OPERATION_ARG_OUT, META_INFO_OPERATION_ARGS_IN_PREFIX, META_INFO_OPERATION_DESCRIPTION, META_INFO_PROPERTY_ACCESS, META_INFO_PROPERTY_ACCESS_EVENTABLE, META_INFO_PROPERTY_ACCESS_READABLE, META_INFO_PROPERTY_ACCESS_WRITABLE, META_INFO_VARIABLE_DESCRIPTION, META_INFO_VARIABLE_MAX, META_INFO_VARIABLE_MIN, META_INFO_VARIABLE_RESOLUTION, META_INFO_VARIABLE_UNIT, META_INFO_VARIABLE_VALUES, PROPERTY_DESCRIPTION, PROPERTY_DEVICE_UID, PROPERTY_GROUP_UID, PROPERTY_OPERATION_NAMES, PROPERTY_PROPERTY_NAMES, PROPERTY_UID

Method Summary		Page
boolean	**getState**() Returns the state of the binary switch.	67
void	**toggle**() Reverses the current state of the binary switch.	67

Methods inherited from interface org.osgi.service.functionaldevice.functions.OnOff
turnOff, turnOn

Methods inherited from interface org.osgi.service.functionaldevice.DeviceFunction
getOperationMetaData, getPropertyMetaData

Field Detail

OPERATION_TOGGLE

```
public static final String OPERATION_TOGGLE = "toggle"
```

Specifies the toggle operation name. The operation can be executed with `toggle()` method.

PROPERTY_STATE

```
public static final String PROPERTY_STATE = "state"
```

Specifies the state property name. The property can be read with `getState()` method.

STATE_ON

```
public static final boolean STATE_ON = true
```

Specifies the on state property value.

STATE_OFF

```
public static final boolean STATE_OFF = false
```

Specifies the off state property value.

Method Detail

toggle

```
void toggle()
     throws UnsupportedOperationException,
            IllegalStateException,
            FunctionalDeviceException
```

Reverses the current state of the binary switch. If the current state is `STATE_ON`, it'll be reversed to `STATE_OFF`. If the current state is `STATE_OFF`, it'll be reversed to `STATE_ON`.

Throws:

`UnsupportedOperationException` - If the operation is not supported.
`IllegalStateException` - If this device service object has already been unregistered.
`FunctionalDeviceException` - If an operation error is available.

---

getState

```
boolean getState()
          throws UnsupportedOperationException,
                 IllegalStateException,
                 FunctionalDeviceException
```

Returns the state of the binary switch. It's a getter method for `PROPERTY_STATE` property.

Returns:

The state of the binary switch.

Throws:

`UnsupportedOperationException` - If the operation is not supported.

`IllegalStateException` - If this device service object has already been unregistered.
`FunctionalDeviceException` - If an operation error is available.

Interface LockUnlock

org.osgi.service.functionaldevice.functions

All Superinterfaces:
DeviceFunction

```
public interface LockUnlock
extends DeviceFunction
```

LockUnlock Device Function represents lock and unlock functionality. The function doesn't provide an access to properties, there are only operations. The Device Function name is org.osgi.service.functionaldevice.functions.LockUnlock.

Field Summary		Page
String	**OPERATION_LOCK** Specifies the lock operation name.	69
String	**OPERATION_UNLOCK** Specifies the unlock operation name.	69

Fields inherited from interface org.osgi.service.functionaldevice.DeviceFunction

META_INFO_OPERATION_ARG_OUT, META_INFO_OPERATION_ARGS_IN_PREFIX, META_INFO_OPERATION_DESCRIPTION, META_INFO_PROPERTY_ACCESS, META_INFO_PROPERTY_ACCESS_EVENTABLE, META_INFO_PROPERTY_ACCESS_READABLE, META_INFO_PROPERTY_ACCESS_WRITABLE, META_INFO_VARIABLE_DESCRIPTION, META_INFO_VARIABLE_MAX, META_INFO_VARIABLE_MIN, META_INFO_VARIABLE_RESOLUTION, META_INFO_VARIABLE_UNIT, META_INFO_VARIABLE_VALUES, PROPERTY_DESCRIPTION, PROPERTY_DEVICE_UID, PROPERTY_GROUP_UID, PROPERTY_OPERATION_NAMES, PROPERTY_PROPERTY_NAMES, PROPERTY_UID

Method Summary		Page
void	**lock**() Lock Device Function operation.	70
void	**unlock**() Unlock Device Function operation.	70

Methods inherited from interface org.osgi.service.functionaldevice.DeviceFunction

getOperationMetaData, getPropertyMetaData

Field Detail

OPERATION_LOCK

```
public static final String OPERATION_LOCK = "lock"
```

Specifies the lock operation name. The operation can be executed with lock() method.

OPERATION_UNLOCK

```
public static final String OPERATION_UNLOCK = "unlock"
```

Specifies the unlock operation name. The operation can be executed with unlock() method.

Method Detail

lock

```
void lock()
          throws UnsupportedOperationException,
                 IllegalStateException,
                 FunctionalDeviceException
```

Lock Device Function operation. The operation name is OPERATION_LOCK.

Throws:

UnsupportedOperationException - If the operation is not supported.
IllegalStateException - If this device service object has already been unregistered.
FunctionalDeviceException - If an operation error is available.

---

unlock

```
void unlock()
     throws UnsupportedOperationException,
            IllegalStateException,
            FunctionalDeviceException
```

Unlock Device Function operation. The operation name is OPERATION_UNLOCK.

Throws:

UnsupportedOperationException - If the operation is not supported.
IllegalStateException - If this device service object has already been unregistered.
FunctionalDeviceException - If an operation error is available.

Interface Meter

org.osgi.service.functionaldevice.functions

All Superinterfaces:
DeviceFunction

```
public interface Meter
extends DeviceFunction
```

`Meter` Device Function can measure metering information. The function provides two properties and one operation:

- PROPERTY_CURRENT
- - property accessible with `getCurrent()` getter;
- PROPERTY_TOTAL
- - property accessible with `getTotal()` getter;
- OPERATION_RESET_TOTAL
- - operation can be executed with `resetTotal()`.

The Device Function name is `org.osgi.service.functionaldevice.functions.Meter`.

Field Summary	Page
String **OPERATION_RESET_TOTAL** Specifies the reset total operation name.	71
String **PROPERTY_CURRENT** Specifies the current info property name.	72
String **PROPERTY_TOTAL** Specifies the total info property name.	72

Fields inherited from interface org.osgi.service.functionaldevice.DeviceFunction

META_INFO_OPERATION_ARG_OUT, META_INFO_OPERATION_ARGS_IN_PREFIX, META_INFO_OPERATION_DESCRIPTION, META_INFO_PROPERTY_ACCESS, META_INFO_PROPERTY_ACCESS_EVENTABLE, META_INFO_PROPERTY_ACCESS_READABLE, META_INFO_PROPERTY_ACCESS_WRITABLE, META_INFO_VARIABLE_DESCRIPTION, META_INFO_VARIABLE_MAX, META_INFO_VARIABLE_MIN, META_INFO_VARIABLE_RESOLUTION, META_INFO_VARIABLE_UNIT, META_INFO_VARIABLE_VALUES, PROPERTY_DESCRIPTION, PROPERTY_DEVICE_UID, PROPERTY_GROUP_UID, PROPERTY_OPERATION_NAMES, PROPERTY_PROPERTY_NAMES, PROPERTY_UID

Method Summary	Page
double **getCurrent**() Returns the current metering info.	72
double **getTotal**() Returns the total metering info.	72
void **resetTotal**() Resets the total metering info.	72

Methods inherited from interface org.osgi.service.functionaldevice.DeviceFunction

getOperationMetaData, getPropertyMetaData

Field Detail

OPERATION_RESET_TOTAL

```
public static final String OPERATION_RESET_TOTAL = "resetTotal"
```

Specifies the reset total operation name. The operation can be executed with resetTotal() method.

PROPERTY_CURRENT

```
public static final String PROPERTY_CURRENT = "current"
```

Specifies the current info property name. The property can be read with getCurrent() method.

PROPERTY_TOTAL

```
public static final String PROPERTY_TOTAL = "total"
```

Specifies the total info property name. The property can be read with getTotal() method.

Method Detail

getCurrent

```
double getCurrent()
          throws UnsupportedOperationException,
                 IllegalStateException,
                 FunctionalDeviceException
```

Returns the current metering info. It's a getter method for PROPERTY_CURRENT property.

Returns:
The current metering info.
Throws:
`UnsupportedOperationException` - If the operation is not supported.
`IllegalStateException` - If this device service object has already been unregistered.
`FunctionalDeviceException` - If an operation error is available.

---

getTotal

```
double getTotal()
        throws UnsupportedOperationException,
               IllegalStateException,
               FunctionalDeviceException
```

Returns the total metering info. It's a getter method for PROPERTY_TOTAL property.

Returns:
The total metering info.
Throws:
`UnsupportedOperationException` - If the operation is not supported.
`IllegalStateException` - If this device service object has already been unregistered.
`FunctionalDeviceException` - If an operation error is available.

---

resetTotal

```
void resetTotal()
        throws UnsupportedOperationException,
               IllegalStateException,
               FunctionalDeviceException
```

Resets the total metering info.

Throws:

`UnsupportedOperationException` - If the operation is not supported.
`IllegalStateException` - If this device service object has already been unregistered.
`FunctionalDeviceException` - If an operation error is available.

Interface MultiLevelSensor

org.osgi.service.functionaldevice.functions

All Superinterfaces:
DeviceFunction

```
public interface MultiLevelSensor
extends DeviceFunction
```

`MultiLevelSensor` Device Function provides multi-level sensor monitoring. It reports its state when an important event is available. The state is accessible with `getState()` getter. There are no operations. The Device Function name is `org.osgi.service.functionaldevice.functions.MultiLevelSensor`.

Field Summary		Page
String	**PROPERTY_STATE** Specifies the state property name.	74

Fields inherited from interface org.osgi.service.functionaldevice.DeviceFunction

META_INFO_OPERATION_ARG_OUT, META_INFO_OPERATION_ARGS_IN_PREFIX, META_INFO_OPERATION_DESCRIPTION, META_INFO_PROPERTY_ACCESS, META_INFO_PROPERTY_ACCESS_EVENTABLE, META_INFO_PROPERTY_ACCESS_READABLE, META_INFO_PROPERTY_ACCESS_WRITABLE, META_INFO_VARIABLE_DESCRIPTION, META_INFO_VARIABLE_MAX, META_INFO_VARIABLE_MIN, META_INFO_VARIABLE_RESOLUTION, META_INFO_VARIABLE_UNIT, META_INFO_VARIABLE_VALUES, PROPERTY_DESCRIPTION, PROPERTY_DEVICE_UID, PROPERTY_GROUP_UID, PROPERTY_OPERATION_NAMES, PROPERTY_PROPERTY_NAMES, PROPERTY_UID

Method Summary		Page
double	**getState**() Returns the state of the Multi Level Sensor.	74

Methods inherited from interface org.osgi.service.functionaldevice.DeviceFunction

getOperationMetaData, getPropertyMetaData

Field Detail

PROPERTY_STATE

```
public static final String PROPERTY_STATE = "state"
```

Specifies the state property name. The property can be read with `getState()` method.

Method Detail

getState

```
double getState()
         throws UnsupportedOperationException,
                IllegalStateException,
                FunctionalDeviceException
```

Returns the state of the Multi Level Sensor. It's a getter method for `PROPERTY_STATE` property.

Returns:
The state of the Multi Level Sensor.

Throws:

`UnsupportedOperationException` - If the operation is not supported.
`IllegalStateException` - If this device service object has already been unregistered.
`FunctionalDeviceException` - If an operation error is available.

Interface MultiLevelSwitch

org.osgi.service.functionaldevice.functions

All Superinterfaces:
DeviceFunction

```
public interface MultiLevelSwitch
extends DeviceFunction
```

MultiLevelSwitch Device Function provides multi-level switch control. It has a level, can increase or decrease the level with a given step and can set the level to a specific value. The level is accessible with getLevel() getter and can be set with setLevel(int) setter. The step is accessible with getStep() getter. stepUp() and stepDown() can increase and decrease the level with a given step. The Device Function name is org.osgi.service.functionaldevice.functions.MultiLevelSwitch.

Field Summary		Page
String	**OPERATION_STEP_DOWN** Specifies the step down operation name.	77
String	**OPERATION_STEP_UP** Specifies the step up operation name.	77
String	**PROPERTY_LEVEL** Specifies the level property name.	77
String	**PROPERTY_STEP** Specifies the step property name.	77

Fields inherited from interface org.osgi.service.functionaldevice.DeviceFunction

META_INFO_OPERATION_ARG_OUT, META_INFO_OPERATION_ARGS_IN_PREFIX, META_INFO_OPERATION_DESCRIPTION, META_INFO_PROPERTY_ACCESS, META_INFO_PROPERTY_ACCESS_EVENTABLE, META_INFO_PROPERTY_ACCESS_READABLE, META_INFO_PROPERTY_ACCESS_WRITABLE, META_INFO_VARIABLE_DESCRIPTION, META_INFO_VARIABLE_MAX, META_INFO_VARIABLE_MIN, META_INFO_VARIABLE_RESOLUTION, META_INFO_VARIABLE_UNIT, META_INFO_VARIABLE_VALUES, PROPERTY_DESCRIPTION, PROPERTY_DEVICE_UID, PROPERTY_GROUP_UID, PROPERTY_OPERATION_NAMES, PROPERTY_PROPERTY_NAMES, PROPERTY_UID

Method Summary		Page
int	**getLevel**() Returns the Multi Level Switch level.	77
int	**getStep**() Returns the step of the Multi Level Switch.	78
void	**setLevel**(int level) Sets the level of the Multi Level Switch.	77
void	**stepDown**() Moves the current level of the switch one step down.	78
void	**stepUp**() Moves the current level of the switch one step up.	78

Methods inherited from interface org.osgi.service.functionaldevice.DeviceFunction

getOperationMetaData, getPropertyMetaData

Field Detail

OPERATION_STEP_DOWN

```
public static final String OPERATION_STEP_DOWN = "stepDown"
```

Specifies the step down operation name. The operation can be executed with `stepDown()` method.

OPERATION_STEP_UP

```
public static final String OPERATION_STEP_UP = "stepUp"
```

Specifies the step up operation name. The operation can be executed with `stepUp()` method.

PROPERTY_LEVEL

```
public static final String PROPERTY_LEVEL = "level"
```

Specifies the level property name. The property can be read with `getLevel()` getter and can be set with `setLevel(int)` setter.

PROPERTY_STEP

```
public static final String PROPERTY_STEP = "step"
```

Specifies the step property name. The property can be read with `getStep()` getter.

Method Detail

getLevel

```
int getLevel()
      throws UnsupportedOperationException,
             IllegalStateException,
             FunctionalDeviceException
```

Returns the Multi Level Switch level. It's a getter method for `PROPERTY_LEVEL` property.

Returns:
The level of the Multi Level Switch.

Throws:
`UnsupportedOperationException` - If the operation is not supported.
`IllegalStateException` - If this device service object has already been unregistered.
`FunctionalDeviceException` - If an operation error is available.

---

setLevel

```
void setLevel(int level)
       throws UnsupportedOperationException,
              IllegalStateException,
              FunctionalDeviceException
```

Sets the level of the Multi Level Switch. It's a setter method for `PROPERTY_LEVEL` property.

Parameters:
`level` - The new level of the Multi Level Switch.

Throws:
UnsupportedOperationException - If the operation is not supported.
IllegalStateException - If this device service object has already been unregistered.
FunctionalDeviceException - If an operation error is available.

---

getStep

```
int getStep()
     throws UnsupportedOperationException,
            IllegalStateException,
            FunctionalDeviceException
```

Returns the step of the Multi Level Switch. It's a getter method for PROPERTY_STEP property. The step is used by stepUp() and stepDown() method to increase and decrease the current level.

Returns:
The step of the Multi Level Switch.
Throws:
UnsupportedOperationException - If the operation is not supported.
IllegalStateException - If this device service object has already been unregistered.
FunctionalDeviceException - If an operation error is available.

---

stepDown

```
void stepDown()
       throws UnsupportedOperationException,
              IllegalStateException,
              FunctionalDeviceException
```

Moves the current level of the switch one step down.

Throws:
UnsupportedOperationException - If the operation is not supported.
IllegalStateException - If this device service object has already been unregistered.
FunctionalDeviceException - If an operation error is available.

---

stepUp

```
void stepUp()
     throws UnsupportedOperationException,
            IllegalStateException,
            FunctionalDeviceException
```

Moves the current level of the switch one step up.

Throws:
UnsupportedOperationException - If the operation is not supported.
IllegalStateException - If this device service object has already been unregistered.
FunctionalDeviceException - If an operation error is available.

Interface OnOff

org.osgi.service.functionaldevice.functions

All Superinterfaces:
DeviceFunction

All Known Subinterfaces:
BinarySwitch

```
public interface OnOff
extends DeviceFunction
```

OnOff Device Function represents turn on and off functionality. The function doesn't provide an access to properties, there are only operations. The Device Function name is org.osgi.service.functionaldevice.functions.OnOff.

Field Summary	Page
String **OPERATION_TURN_OFF** Specifies the turn off operation name.	79
String **OPERATION_TURN_ON** Specifies the turn on operation name.	79

Fields inherited from interface org.osgi.service.functionaldevice.DeviceFunction
META_INFO_OPERATION_ARG_OUT, META_INFO_OPERATION_ARGS_IN_PREFIX, META_INFO_OPERATION_DESCRIPTION, META_INFO_PROPERTY_ACCESS, META_INFO_PROPERTY_ACCESS_EVENTABLE, META_INFO_PROPERTY_ACCESS_READABLE, META_INFO_PROPERTY_ACCESS_WRITABLE, META_INFO_VARIABLE_DESCRIPTION, META_INFO_VARIABLE_MAX, META_INFO_VARIABLE_MIN, META_INFO_VARIABLE_RESOLUTION, META_INFO_VARIABLE_UNIT, META_INFO_VARIABLE_VALUES, PROPERTY_DESCRIPTION, PROPERTY_DEVICE_UID, PROPERTY_GROUP_UID, PROPERTY_OPERATION_NAMES, PROPERTY_PROPERTY_NAMES, PROPERTY_UID

Method Summary	Page
void **turnOff**() Turn off Device Function operation.	80
void **turnOn**() Turn on Device Function operation.	80

Methods inherited from interface org.osgi.service.functionaldevice.DeviceFunction
getOperationMetaData, getPropertyMetaData

Field Detail

OPERATION_TURN_ON

```
public static final String OPERATION_TURN_ON = "turnOn"
```

Specifies the turn on operation name. The operation can be executed with turnOn() method.

OPERATION_TURN_OFF

```
public static final String OPERATION_TURN_OFF = "turnOff"
```

Specifies the turn off operation name. The operation can be executed with `turnOff()` method.

Method Detail

turnOn

```
void turnOn()
     throws UnsupportedOperationException,
            IllegalStateException,
            FunctionalDeviceException
```

Turn on Device Function operation. The operation name is `OPERATION_TURN_ON`.

Throws:

`UnsupportedOperationException` - If the operation is not supported.
`IllegalStateException` - If this device service object has already been unregistered.
`FunctionalDeviceException` - If an operation error is available.

---

turnOff

```
void turnOff()
      throws UnsupportedOperationException,
             IllegalStateException,
             FunctionalDeviceException
```

Turn off Device Function operation. The operation name is `OPERATION_TURN_OFF`.

Throws:

`UnsupportedOperationException` - If the operation is not supported.
`IllegalStateException` - If this device service object has already been unregistered.
`FunctionalDeviceException` - If an operation error is available.

Interface OpenClose

org.osgi.service.functionaldevice.functions

All Superinterfaces:
DeviceFunction

```
public interface OpenClose
extends DeviceFunction
```

OpenClose Device Function represents open and close functionality. The function doesn't provide an access to properties, there are only operations. The Device Function name is org.osgi.service.functionaldevice.functions.OpenClose.

Field Summary		Page
String	**OPERATION_CLOSE** Specifies the close operation name.	81
String	**OPERATION_OPEN** Specifies the open operation name.	81

Fields inherited from interface org.osgi.service.functionaldevice.DeviceFunction
META_INFO_OPERATION_ARG_OUT, META_INFO_OPERATION_ARGS_IN_PREFIX, META_INFO_OPERATION_DESCRIPTION, META_INFO_PROPERTY_ACCESS, META_INFO_PROPERTY_ACCESS_EVENTABLE, META_INFO_PROPERTY_ACCESS_READABLE, META_INFO_PROPERTY_ACCESS_WRITABLE, META_INFO_VARIABLE_DESCRIPTION, META_INFO_VARIABLE_MAX, META_INFO_VARIABLE_MIN, META_INFO_VARIABLE_RESOLUTION, META_INFO_VARIABLE_UNIT, META_INFO_VARIABLE_VALUES, PROPERTY_DESCRIPTION, PROPERTY_DEVICE_UID, PROPERTY_GROUP_UID, PROPERTY_OPERATION_NAMES, PROPERTY_PROPERTY_NAMES, PROPERTY_UID

Method Summary		Page
void	**close**() Close Device Function operation.	82
void	**open**() Open Device Function operation.	82

Methods inherited from interface org.osgi.service.functionaldevice.DeviceFunction
getOperationMetaData, getPropertyMetaData

Field Detail

OPERATION_OPEN

```
public static final String OPERATION_OPEN = "open"
```

Specifies the open operation name. The operation can be executed with open() method.

OPERATION_CLOSE

```
public static final String OPERATION_CLOSE = "close"
```

Specifies the close operation name. The operation can be executed with close() method.

Method Detail

open

```
void open()
          throws UnsupportedOperationException,
                 IllegalStateException,
                 FunctionalDeviceException
```

Open Device Function operation. The operation name is OPERATION_OPEN.

Throws:

UnsupportedOperationException - If the operation is not supported.
IllegalStateException - If this device service object has already been unregistered.
FunctionalDeviceException - If an operation error is available.

---

close

```
void close()
    throws UnsupportedOperationException,
           IllegalStateException,
           FunctionalDeviceException
```

Close Device Function operation. The operation name is OPERATION_CLOSE.

Throws:

UnsupportedOperationException - If the operation is not supported.
IllegalStateException - If this device service object has already been unregistered.
FunctionalDeviceException - If an operation error is available.

8 Considered Alternatives

8.1 Use Configuration Admin to update the Device service properties

OSGi service properties are used to represent the Device service properties. The properties can be updated with the help of `org.osgi.framework.ServiceRegistration.setProperties(Dictionary)` method. The service registration is intended for a private usage and should not be shared between the bundles.

The current design provides set methods, which can be used when an external application wants to modify the Device service properties. It's simple and a part of Device interface. We have to define a new permission check, because there is no such protection to `org.osgi.framework.ServiceRegistration.setProperties` method.

Considered alternative was about property update based on configuration update in the Configuration Admin service. The Device service properties can be updated when the corresponding configuration properties are updated. The disadvantages here are:

- Device properties duplication – they are stored in the device configuration and in the Device service properties.
- Possible performance issue when a lot of devices are used.

8.2 DeviceAdmin interface availability

DeviceAdmin service was removed from the current RFC document. That management functionality can be provided by a different specification document. That considered alternative is kept for completeness.

DeviceAdmin service can simplify the device service registration. It hides the implementation details i.e. realize program to an interface rather than to an implementation.

The considered alternative is not to use that interface and to register the Device service implementation to the OSGi service registry. Here are two code snippets, which demonstrates positives and negatives:

1. Without DeviceAdmin

```
Map ipCameraProps = new HashMap(3, 1F);
ipCameraProps.put("IP.Camera.Address", "192.168.0.21");
ipCameraProps.put("IP.Camera.Username", "test");
ipCameraProps.put("IP.Camera.Password", "test");

//WARNING - an access to implementation class, which should be bundle private
IPCameraDeviceImpl ipCameraImpl = new IPCameraDeviceImpl(ipCameraProps);
ipCameraImpl.register(bundleContext);
// play the video stream...
```

```
// remove the device
ipCameraImpl.unregister();
```

That snippet demonstrate program to implementation rather than an interface, which break basic OOP rule.

2. With DeviceAdmin

```
Map ipCameraProps = new HashMap(3, 1F);
ipCameraProps.put("IP.Camera.Address", "192.168.0.21");
ipCameraProps.put("IP.Camera.Username", "test");
ipCameraProps.put("IP.Camera.Password", "test");

DeviceAdmin ipCameraDeviceAdmin = getIPCameraDeviceAdmin();
Device ipCamera = ipCameraDeviceAdmin.add(ipCameraProps);
// play the device video stream
// remove the device
ipCamera.remove();
```

It demonstrate program to interface rather than an implementation, which is the correct approach.

8.3 Access helper methods removal of FunctionalDevice

org.osgi.service.functionaldevice.FunctionalDevice.getChildren(), org.osgi.service.functionaldevice.FunctionalDevice.getParent() and org.osgi.service.functionaldevice.FunctionalDevice.getReferences() were removed, because they provided access to the FunctionalDevice services outside the OSGi service registry. It can be problematic in various scenarios like:

- The service Find Hook can be ignored.
- No service unget is possible for such shared service instances.
- The dependency tools based on the service registry cannot track such sharings.

9 Security Considerations

9.1 Functional Device Permission

A bundle's authority to perform specific privileged administrative operations on the devices. The actions for this permission are:

Action	Method
ACTION_REMOVE	FunctionalDevice.remove()

ACTION_ENABLE	FunctionalDevice.enable()
ACTION_DISABLE	FunctionalDevice.disable()
ACTION_PROPERTY	FunctionalDevice.setProperty(String, Object) FunctionalDevice.setProperties(String[], Object[])

The name of the permission is a filter based. For more details about filter based permissions, see OSGi Core Specification, Filter Based Permissions. The filter provides an access to all device service properties. The service property names are case insensitive. The filter attribute names are processed in a case insensitive manner. For example, the operator can give a bundle the permission to only manage devices of vendor “acme”:

```
org.osgi.services.functionaldevice.FunctionalDevicePermission(“abstract.device.hardware.vendor=acme”, …)
```

The permission actions allows the operator to assign only the necessary permissions to the bundle. For example, the management bundle can have permission to remove all registered devices:

```
org.osgi.services.functionaldevice.FunctionalDevicePermission(“*”, “remove”)
```

The code that needs to check the Functional Device Permission must always use the constructor that takes the device as a parameter `FunctionalDevicePermission(FunctionalDevice, String)` with a single action. For example, the implementation of `org.osgi.services.functionaldevice.FunctionalDevice.remove()` method must check that the caller has an access to the operation:

```
public class DeviceImpl implements FunctionalDevice {
   public void start() {
      securityManager.checkPermission(new FunctionalDevicePermission(this, “remove”));
   }
}
```

9.2 Required Permissions

The Functional Device implementation must check the caller for the appropriate Functional Device Permission before execution of the real operation actions like remove, enable etc. Once the Functional Device Permission is checked against the caller the implementation will proceed with the actual operation. The operation can require a number of other permissions to complete. The implementation must isolate the caller from such permission checks by use of proper privileged blocks.

10 Document Support

10.1 References

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

[2]. Software Requirements & Specifications. Michael Jackson. ISBN 0-201-87712-0

[3]. JavaBeans Spec, http://www.oracle.com/technetwork/java/javase/documentation/spec-136004.html

[4]. Unicode Standard Annex #15, Unicode Normalization Forms

10.2 Author's Address

Name	Evgeni Grigorov
Company	ProSyst Software
Address	Aachenerstr. 222, 50935 Cologne, Germany
Voice	+49 221 6604 501
e-mail	e.grigorov@prosyst.com

10.3 Acronyms and Abbreviations

Item	Description
Device Abstraction Layer	Unifies the work with devices provided by different protocols.
Device Abstraction API	Unified API for management of devices provided by different protocols.
Device Abstraction Adapter	Examples for such adapters are ZigBee Adapter, Z-Wave Adapter etc. Provides support for a particular device protocol to Device Abstraction Layer. The adapter integrates the protocol specific driver devices.

10.4 End of Document