Device Abstraction Layer

Draft

134 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

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 10.1.

`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
7th draft	Sept 09 2013	Basic device function set is updated. Rename FunctionalDevice to Device. Rename FunctionalDeviceException to DeviceException. Rename FunctionalDevicePermission to DevicePermission. Relax the relation between the device and device function. DeviceExcpetion extends IOException. Functional group is removed. Renamed device function metadata properties. Evgeni Grigorov, ProSyst Software, e.grigorov@prosyst.com
8th draft	Jan 16 2014	Service property names are renamed form PROPERTY_<name> to SERVICE_<name>. Status disabled is removed, because it's applicable to small set of devices like peripherals. Remove the public methods to update the device properties. They should be initially configured. Updated permissions, because of updated device management operations. Overview diagram is added. Diagram with all device statuses is added. The package is renamed. Common device function data structure is introduced. Property and operation metadata structures are introduced. Device function type is added. There is a new interface with base set of device function types. There is a new interface with SI unit symbols. 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.

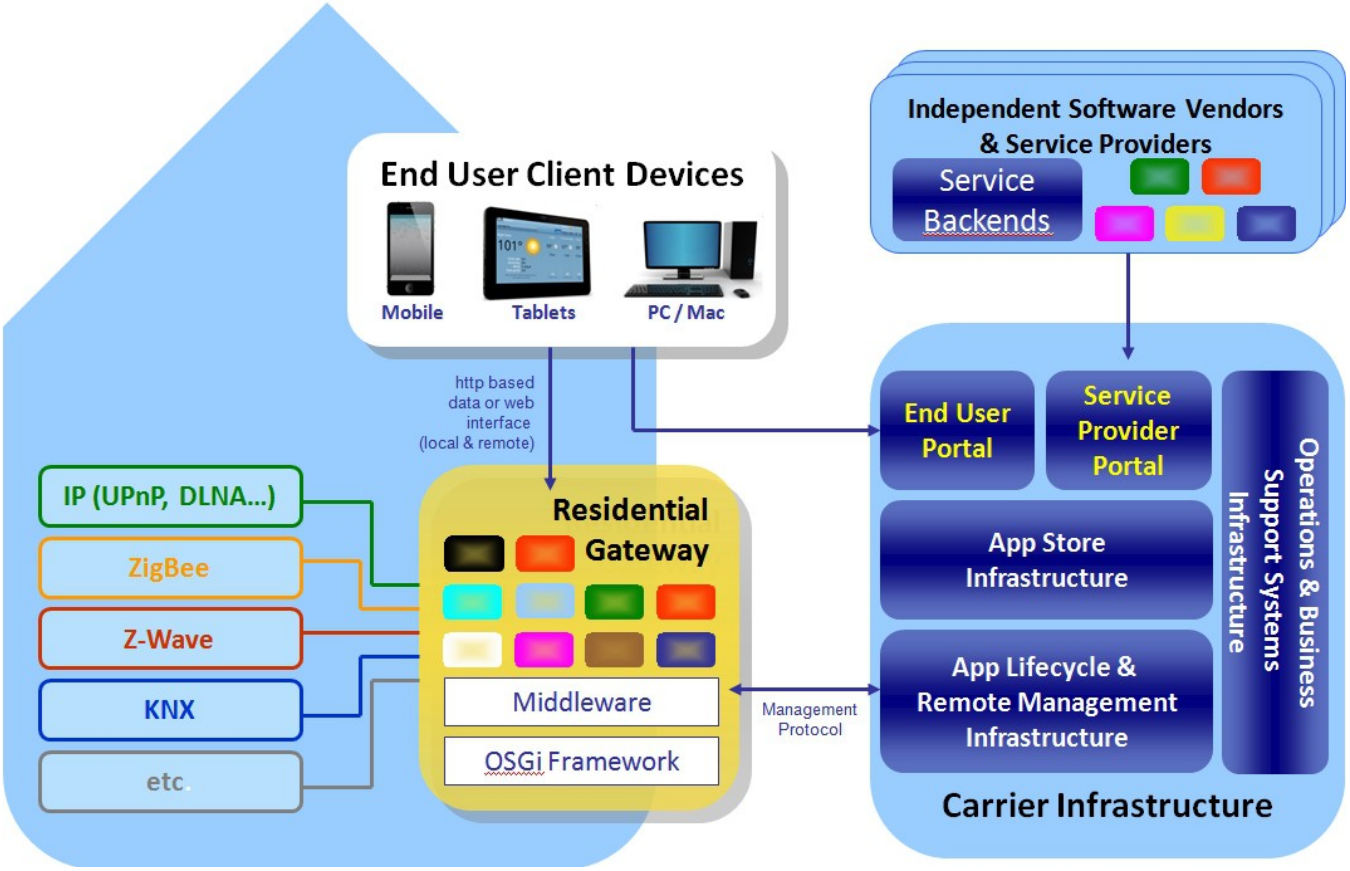

Illustration 1

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:

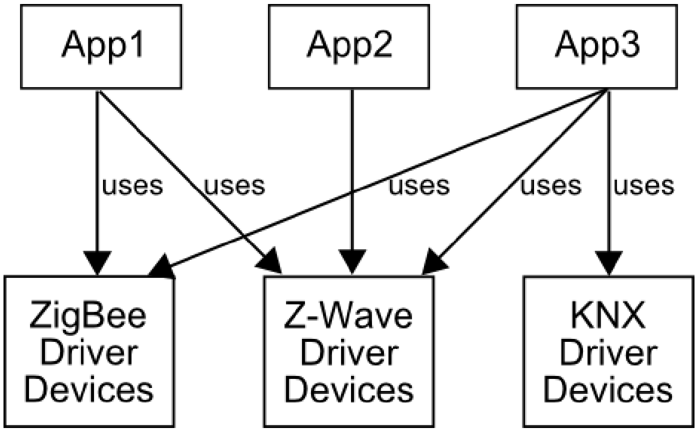

Illustration 2

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.

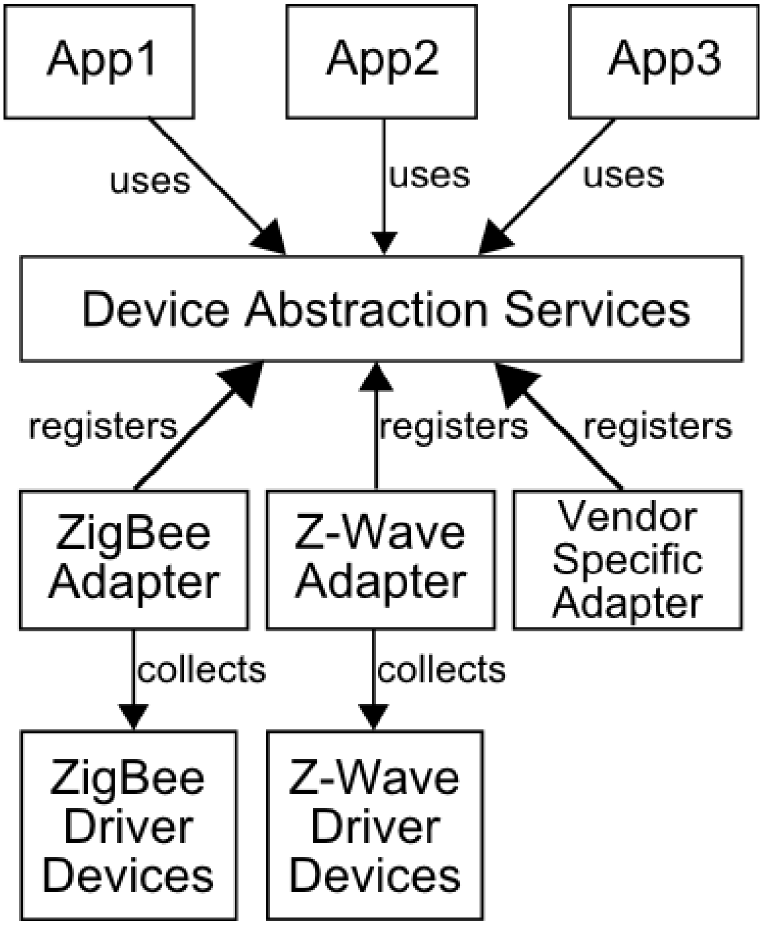

Illustration 3

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

5.1 Introduction

Remote device control provides opportunity to save energy, to provide better security, to save your time during daily tasks and many more. The devices can play different roles in their networks as events reporters, controllers etc. That dynamic behavior is well mappable to the dynamic OSGi service registry. There is a registration of `Device` service. It realizes basic set of management operations and provides rich set of properties. The applications are allowed to track the device status, to read descriptive information and to follow the device relations. A set of functions can belong to the device. They represents the device operations and related properties in an atomic way. The device functions can be found in 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 `Device` service and there is a `DeviceFunction` service to turn on and turn off the light.

5.1.1 Entities

- Device – represents the device in the OSGi service registry. It's described with a set of service properties and provides basic management operations.
- DeviceFunction – atomic device functional entity. The device can support a few functions like switch with a sensor. The function provides a set of properties and operations.
- DeviceFunctionEvent – asynchronous event. It's sent through EventAdmin service and notifies for Device Function property change.
- DeviceFunctionData – data structure carries DeviceFunction property value with additional metadata.
- PropertyMetadata and OperationMetadata – contains metadata about the DeviceFunction properties and operations.

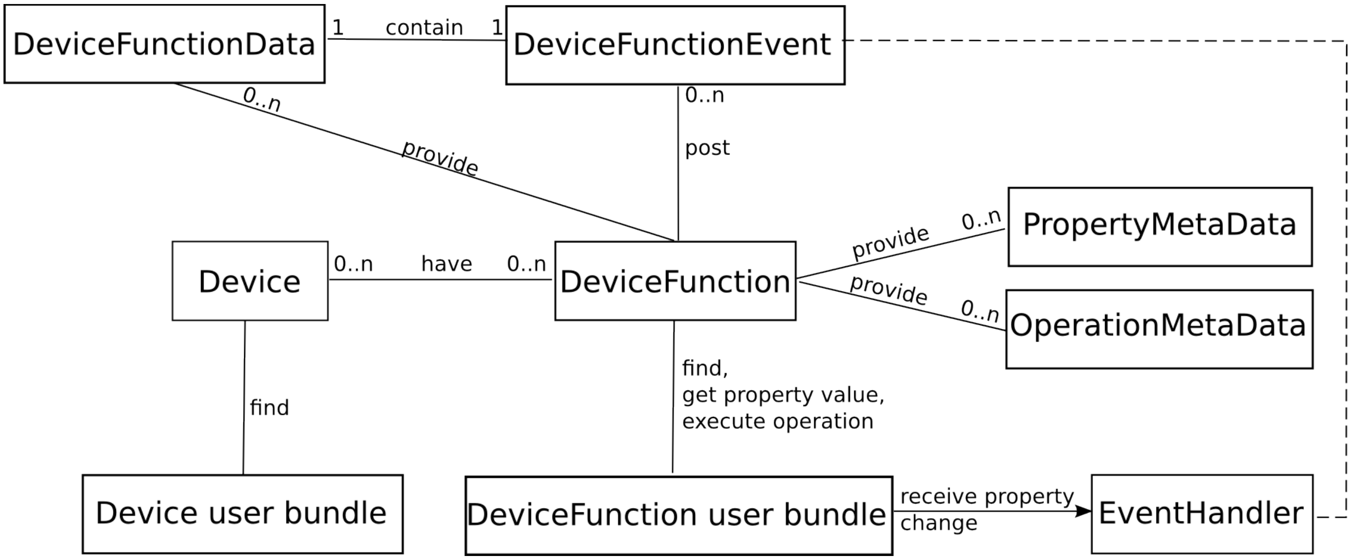

Illustration 4: Device Abstraction Layer Overview

5.2 Device Access Category

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

5.3 Device Service

`Device` interface is dedicated for a common access to the 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 `Device` services and different relations between them. `Device` service can represent pure software unit. For example, it can simulate the real device work. There are basic management operations for remove, property access and property update. New protocol devices can be supported with a registration of new `Device` services.

If the underlying protocol and the implementation allow, the Device services must be registered again after the OSGi framework reboot. The service properties must be restored, the supported device functions must be provided and 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 5:

- Device.SERVICE_UID – Specifies the device 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:

  UID ::= driver-name ':' device-id

  UID - device unique identifier

  driver-name - the value of the Device.SERVICE_DRIVER service property

  device-id - device unique identifier in the scope of the driver
- Device.SERVICE_REFERENCE_UIDS – Specifies the reference device unique identifiers. It's an optional property. 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.
- Device.SERVICE_DRIVER – Specifies the device driver name. For example, ZigBee, Z-Wave, Bluetooth etc. It's a mandatory property. The value type is java.lang.String.
- Device.SERVICE_NAME – Specifies the device name. It's an optional property. The value type is java.lang.String. The property value can be set with Device.setName(String) method.
- Device.SERVICE_STATUS – Specifies the current device status. It's a mandatory property. The value type java.lang.Integer. The possible values are:
  - Device.STATUS_REMOVED – Indicates that the device is removed from the network. That status must be set as the last device status and after that the device service can be unregistered from the service registry. The status is available for stale device services too. All transitions to and from this status are described in Transitions to STATUS_REMOVED section.
  - Device.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.
  - Device.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.
  - Device.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.
  - Device.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.
  - Device.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.

- Device.SERVICE_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 Device.STATUS_DETAIL_CONNECTING. Negative values indicate errors related to the current device status like Device.STATUS_DETAIL_DEVICE_BROKEN. The list with defined status details is:

    - Device.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.

    - Device.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.

    - Device.STATUS_DETAIL_REMOVING – The reason for the current device status is that the device is leaving the network. It indicates the reason with positive value 3. The device status must be STATUS_PROCESSING.

    - Device.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.

    - Device.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.

    - Device.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.

    - Device.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.

    - Device.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.

    - Device.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.

    - Device.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_REMOVING	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

- `Device.SERVICE_HARDWARE_VENDOR` – Specifies the device hardware vendor. It's an optional property. The value type is `java.lang.String`.
- `Device.SERVICE_HARDWARE_VERSION` – Specifies the device hardware version. It's an optional property. The value type is `java.lang.String`.
- `Device.SERVICE_FIRMWARE_VENDOR` – Specifies the device firmware vendor. It's an optional property. The value type is `java.lang.String`.
- `Device.SERVICE_FIRMWARE_VERSION` – Specifies the device firmware version. It's an optional property. The value type is `java.lang.String`.
- `Device.SERVICE_TYPES` – Specified the device types. It's an optional property. The value type is `java.lang.String[]`.
- `Device.SERVICE_MODEL` – Specifies the device model. It's an optional property. The value type is `java.lang.String`.
- `Device.SERVICE_SERIAL_NUMBER` – Specifies the device serial number. 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.Device` interface. The next code snippet prints the online devices.

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

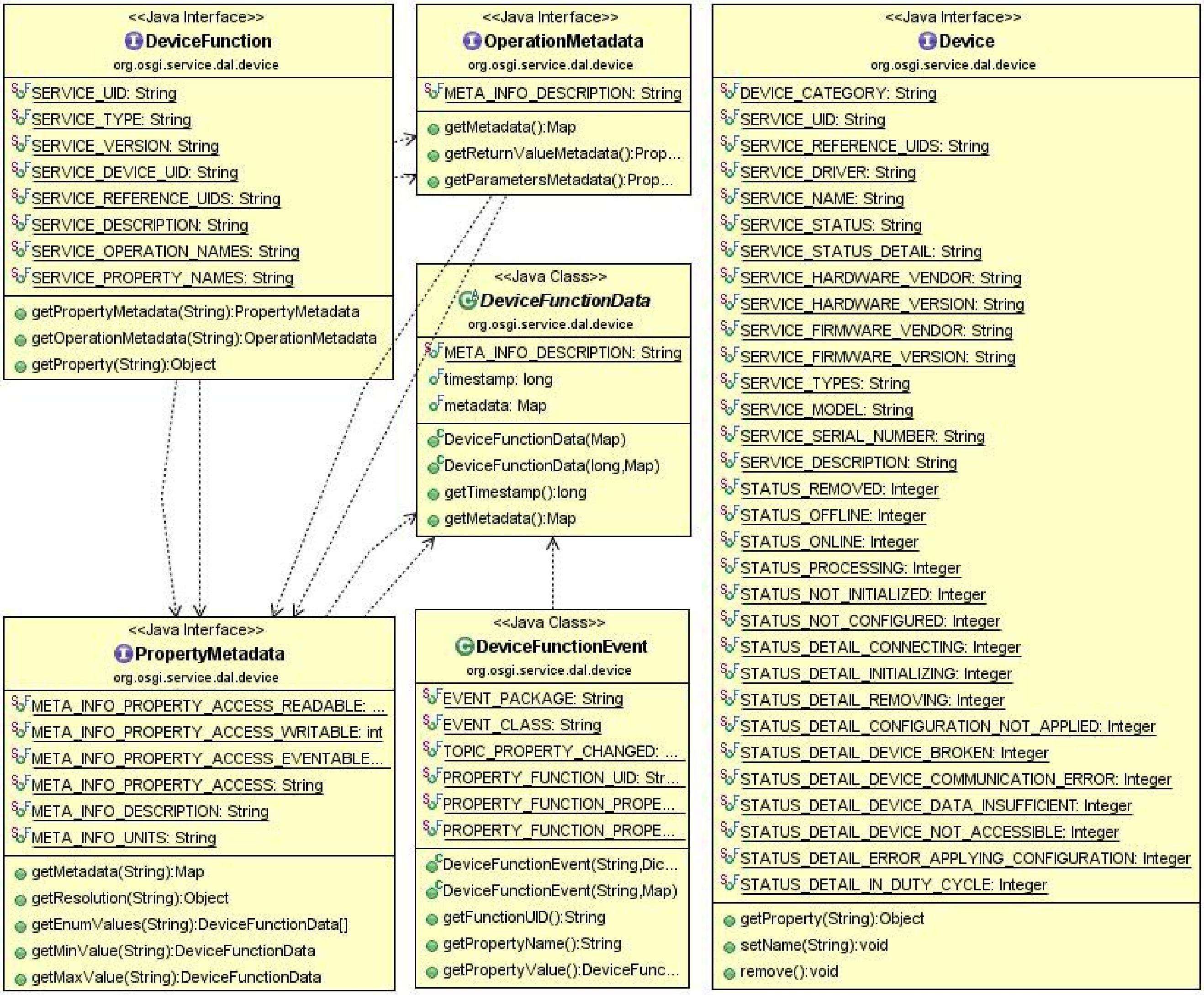
<<Java Interface>>
DeviceFunction
org.osgi.service.dal.device
SERVICE_UID: String
SERVICE_TYPE: String
SERVICE_VERSION: String
SERVICE_DEVICE_UID: String
SERVICE_REFERENCE_UIDS: String
SERVICE_DESCRIPTION: String
SERVICE_OPERATION_NAMES: String
SERVICE_PROPERTY_NAMES: String
getPropertyMetadata(String):PropertyMetadata
getOperationMetadata(String):OperationMetadata
getProperty(String):Object
<<Java Interface>>
OperationMetadata
org.osgi.service.dal.device
META_INFO_DESCRIPTION: String
getMetadata():Map
getReturnValueMetadata():Prop...
getParametersMetadata():Prop...
<<Java Class>>
DeviceFunctionData
org.osgi.service.dal.device
META_INFO_DESCRIPTION: String
timestamp: long
metadata: Map
DeviceFunctionData(Map)
DeviceFunctionData(long,Map)
getTimestamp():long
getMetadata():Map
<<Java Interface>>
Device
org.osgi.service.dal.device
DEVICE_CATEGORY: String
SERVICE_UID: String
SERVICE_REFERENCE_UIDS: String
SERVICE_DRIVER: String
SERVICE_NAME: String
SERVICE_STATUS: String
SERVICE_STATUS_DETAIL: String
SERVICE_HARDWARE_VENDOR: String
SERVICE_HARDWARE_VERSION: String
SERVICE_FIRMWARE_VENDOR: String
SERVICE_FIRMWARE_VERSION: String
SERVICE_TYPES: String
SERVICE_MODEL: String
SERVICE_SERIAL_NUMBER: String
SERVICE_DESCRIPTION: String
STATUS_REMOVED: Integer
STATUS_OFFLINE: Integer
STATUS_ONLINE: Integer
STATUS_PROCESSING: Integer
STATUS_NOT_INITIALIZED: Integer
STATUS_NOT_CONFIGURED: Integer
STATUS_DETAIL_CONNECTING: Integer
STATUS_DETAIL_INITIALIZING: Integer
STATUS_DETAIL_REMOVING: Integer
STATUS_DETAIL_CONFIGURATION_NOT_APPLIED: Integer
STATUS_DETAIL_DEVICE_BROKEN: Integer
STATUS_DETAIL_DEVICE_COMMUNICATION_ERROR: Integer
STATUS_DETAIL_DEVICE_DATA_INSUFFICIENT: Integer
STATUS_DETAIL_DEVICE_NOT_ACCESSIBLE: Integer
STATUS_DETAIL_ERROR_APPLYING_CONFIGURATION: Integer
STATUS_DETAIL_IN_DUTY_CYCLE: Integer
getProperty(String):Object
setName(String):void
remove():void
<<Java Interface>>
PropertyMetadata
org.osgi.service.dal.device
META_INFO_PROPERTY_ACCESS_READABLE: ...
META_INFO_PROPERTY_ACCESS_WRITABLE: int
META_INFO_PROPERTY_ACCESS_EVENTABLE...
META_INFO_PROPERTY_ACCESS: String
META_INFO_DESCRIPTION: String
META_INFO_UNITS: String
getMetadata(String):Map
getResolution(String):Object
getEnumValues(String):DeviceFunctionData[]
getMinValue(String):DeviceFunctionData
getMaxValue(String):DeviceFunctionData
<<Java Class>>
DeviceFunctionEvent
org.osgi.service.dal.device
EVENT_PACKAGE: String
EVENT_CLASS: String
TOPIC_PROPERTY_CHANGED: ...
PROPERTY_FUNCTION_UID: Str...
PROPERTY_FUNCTION_PROPE...
PROPERTY_FUNCTION_PROPE...
DeviceFunctionEvent(String,Dic...
DeviceFunctionEvent(String,Map)
getFunctionUID():String
getPropertyName():String
getPropertyValue():DeviceFunc...

Illustration 5

Applications need to have an access to the device properties. For convenience there are two helper methods:

- `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.
- `setName(String name)` – Sets the device name. The method must synchronously update the `Device.SERVICE_NAME` service property of this device. The new name must be persistently stored. It'll set after framework restart. `null` name will clean up the current device name.

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

5.3.1 Reference Device Services

Device 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 Device and ZigBee network controller Device. The network controller device can have a reference to the physical USB device as it's depicted on Illustration 6.

The related service property is Device.SERVICE_REFERENCE_UIDS.

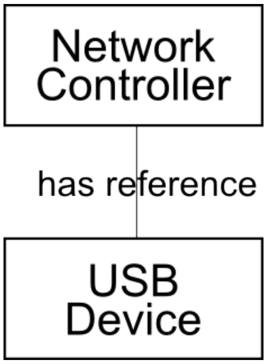

Illustration 6

5.3.2 Device Service Registration

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

5.3.3 Device Service Unregistration

OSGi service registry is only about the read-only access for 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 Device 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. Device services are unregistered first before DeviceFunction services.

5.4 Device Status Transitions

The 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, visualized in Illustration 7 and described in detail in the next sections. The entry device status is always STATUS_PROCESSING. When the device info is processed, the device can go to another status. The last possible device status is STATUS_REMOVED. The status must be set when the device is removed from the network. After that status, the device service will be unregistered.

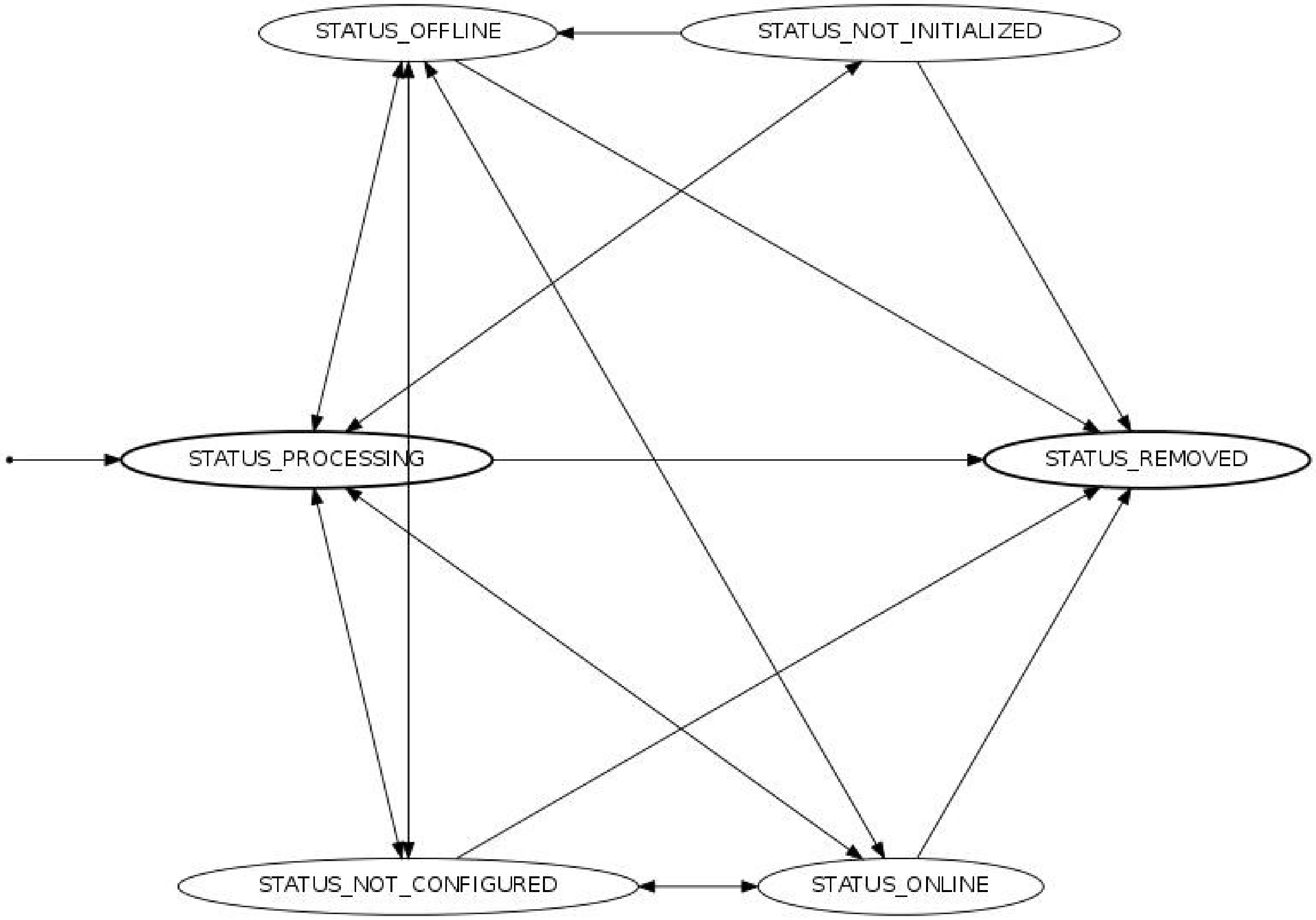

Illustration 7

From \ To Status	PROCESSING	ONLINE	OFFLINE	NOT_INITIALIZED	NOT_CONFIGURED	REMOVED
PROCESSING	-	Initial device data has been read.	Device is not accessible.	Initial device data is partially read.	Device has a pending configuration.	Device is removed.
ONLINE	Device data is processing.	-	Device is not accessible.	-	Device has a new pending configuration.	Device is removed.
OFFLINE	Device data is processing.	Device data has been read.	-	-	Device has a pending configuration.	Device is removed.
NOT_INITIALIZED	Device data is processing.	-	Device is not accessible.	-	-	Device is removed.
NOT_CONFIGURED	Device data is processing.	Device pending configuration is satisfied.	Device is not accessible.	-	-	Device is removed.
REMOVED	-	-	-	-	-	-

Table 2

5.4.1 Transitions to STATUS_REMOVED

The device can go to `Device.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 `Device` service, their status will be set to `STATUS_REMOVED`.

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

5.4.2 Transitions to and from STATUS_OFFLINE

The `STATUS_OFFLINE` indicates that the 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` – device is removed. The status can be set as a result of `Device.remove()` method call.
- From `STATUS_OFFLINE` to `STATUS_PROCESSING` – device data is processing.
- From `STATUS_OFFLINE` to `STATUS_NOT_CONFIGURED` – device has a pending configuration.
- From `STATUS_OFFLINE` to `STATUS_ONLINE` – 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 device is removed, the service will be unregistered from the service registry.
- To STATUS_OFFLINE from STATUS_PROCESSING – device is not accessible any more while device data is processing.
- To STATUS_OFFLINE from STATUS_NOT_CONFIGURED – Not configured device is not accessible any more.
- To STATUS_OFFLINE from STATUS_ONLINE – Online device is not accessible any more.
- To STATUS_OFFLINE from STATUS_NOT_INITIALIZED – Not initialized device is not accessible any more.

The possible transitions are summarized on Illustration 8.

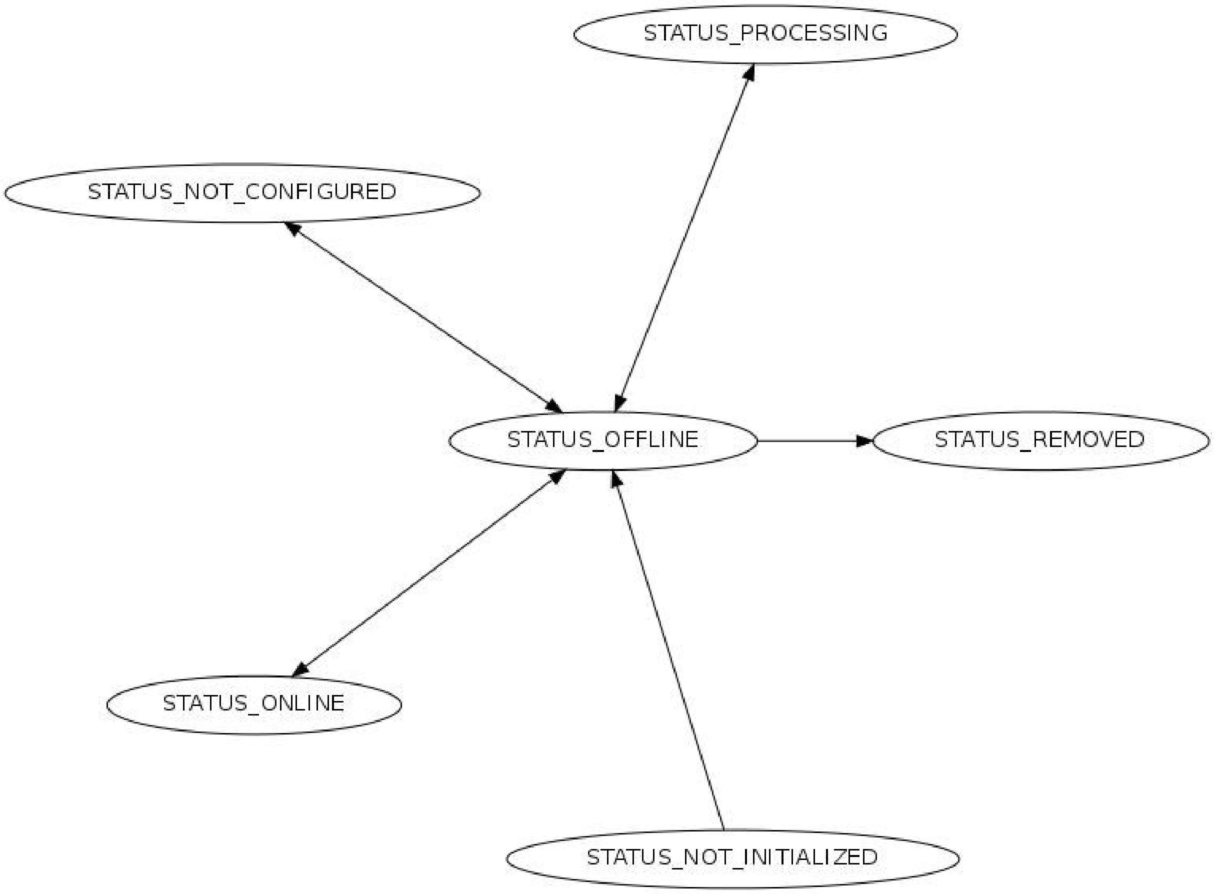

Illustration 8

5.4.3 Transitions to and from STATUS_ONLINE

The STATUS_ONLINE indicates that the 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 – device is removed. The status can be set as a result of Device.remove() method call.
- From STATUS_ONLINE to STATUS_PROCESSING – device data is processing.

- From STATUS_ONLINE to STATUS_NOT_CONFIGURED – device has a pending configuration.
- From STATUS_ONLINE to STATUS_OFFLINE – Online 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 device is removed, the service will be unregistered from the service registry.
- To STATUS_ONLINE from STATUS_PROCESSING – Initial device data has been read. The device is available for operations.
- To STATUS_ONLINE from STATUS_NOT_CONFIGURED – The device pending configuration is satisfied.
- To STATUS_ONLINE from STATUS_OFFLINE – 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 9.

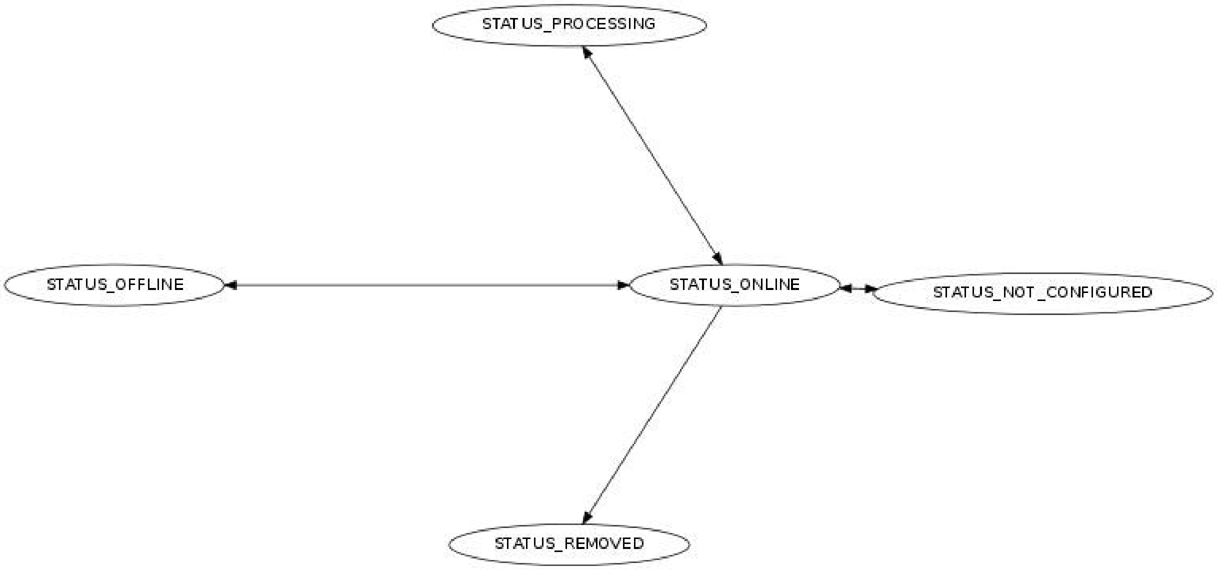

Illustration 9

5.4.4 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 – device is removed. The status can be set as a result of Device.remove() method call.
- From STATUS_PROCESSING to STATUS_ONLINE – Initial device data has been read. The device is available for operations.
- From STATUS_PROCESSING to STATUS_NOT_CONFIGURED – device has a pending configuration.

- From STATUS_PROCESSING to STATUS_OFFLINE – Online device is not accessible any more.
- From STATUS_PROCESSING to STATUS_NOT_INITIALIZED – device initial data is partially read.
- To STATUS_PROCESSING from STATUS_REMOVED – That transition is not possible. If device is removed, the service will be unregistered from the service registry.
- To STATUS_PROCESSING from STATUS_ONLINE – device is busy with an operation.
- To STATUS_PROCESSING from STATUS_NOT_CONFIGURED – The device pending configuration is satisfied and the device is busy with an operation.
- To STATUS_PROCESSING from STATUS_OFFLINE – device is busy with an operation.
- To STATUS_PROCESSING from STATUS_NOT_INITIALIZED – device initial data is processing.

The possible transitions are summarized on Illustration 10.

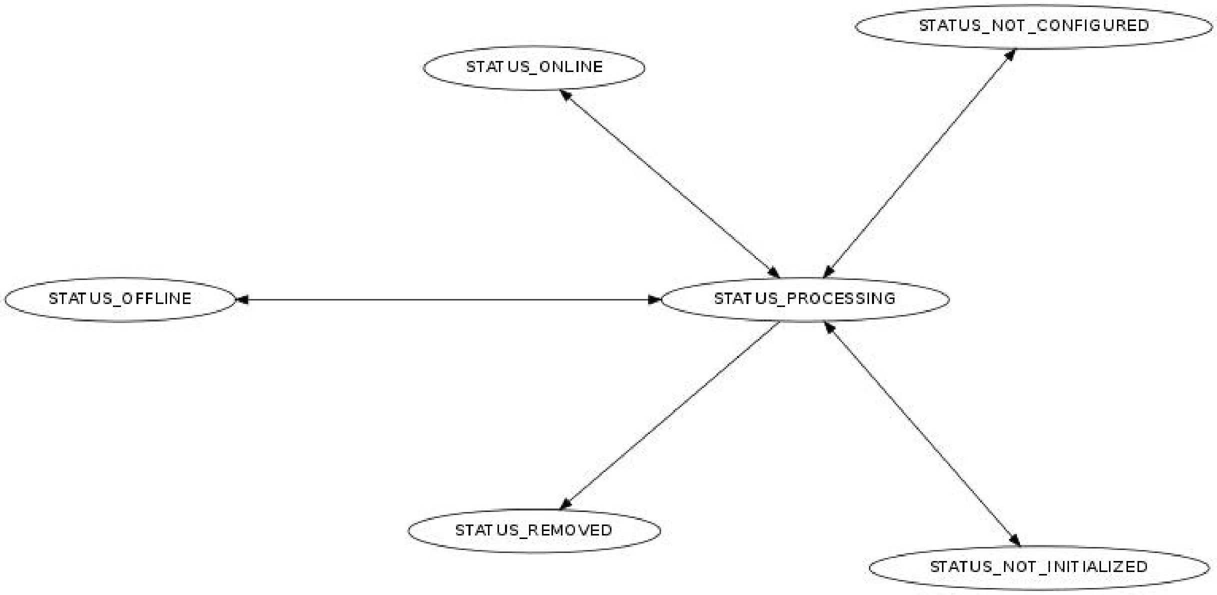

Illustration 10

5.4.5 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 – device is removed. The status can be set as a result of Device.remove() method call.
- From STATUS_NOT_INITIALIZED to STATUS_PROCESSING – device data is processing.
- From STATUS_NOT_INITIALIZED to STATUS_NOT_CONFIGURED – That transition is not possible. device requires some data processing.
- From STATUS_NOT_INITIALIZED to STATUS_OFFLINE – device is not accessible any more.
- From STATUS_NOT_INITIALIZED to STATUS_ONLINE – That transition is not possible. Device requires some data processing to become online.

- To STATUS_NOT_INITIALIZED from STATUS_REMOVED – That transition is not possible. If device is removed, the service will be unregistered from the service registry.
- To STATUS_NOT_INITIALIZED from STATUS_PROCESSING – device data is partially read.
- To STATUS_NOT_INITIALIZED from STATUS_NOT_CONFIGURED – That transition is not possible. When device pending configuration is satisfied, the device requires additional data processing.
- To STATUS_NOT_INITIALIZED from STATUS_OFFLINE – That transition is not possible. Device requires some data processing and then can become not initialized.
- To STATUS_NOT_INITIALIZED from STATUS_ONLINE – That transition is not possible. Online device is initialized.

The possible transitions are summarized on Illustration 11.

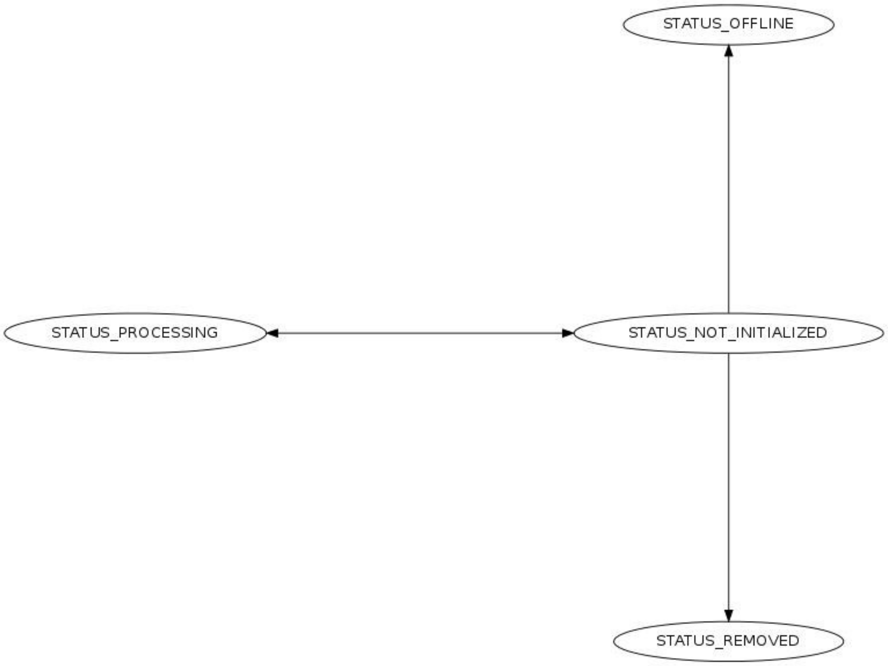

Illustration 11

5.4.6 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 – device is removed. The status can be set as a result of Device.remove() method call.
- From STATUS_NOT_CONFIGURED to STATUS_PROCESSING – device pending configuration is satisfied and some additional data processing is required.
- From STATUS_NOT_CONFIGURED to STATUS_ONLINE – device pending configuration is satisfied.
- From STATUS_NOT_CONFIGURED to STATUS_OFFLINE – device is not accessible any more.
- From STATUS_NOT_CONFIGURED to STATUS_NOT_INITIALIZED – That transition is not possible. When device pending configuration is satisfied, the device requires additional data processing.

- To STATUS_NOT_CONFIGURED from STATUS_REMOVED – That transition is not possible. If device is removed, the service will be unregistered from the service registry.
- To STATUS_NOT_CONFIGURED from STATUS_PROCESSING – Initial device data has been read but there is a pending configuration.
- To STATUS_NOT_CONFIGURED from STATUS_ONLINE – device has a pending configuration.
- To STATUS_NOT_CONFIGURED from STATUS_OFFLINE – 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. Device requires some data processing.

The possible transitions are summarized on Illustration 12.

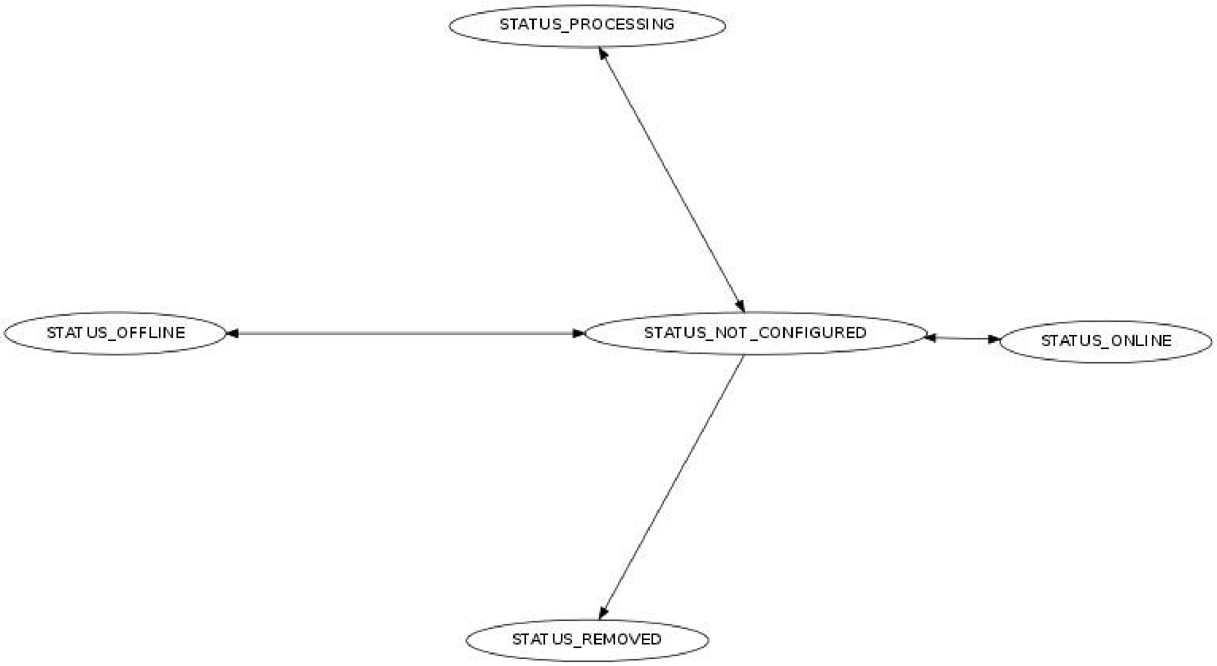

Illustration 12

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 DeviceFunction services. The DeviceFunction service can be registered in the service registry with those service properties:

- DeviceFunction.SERVICE_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 Device.SERVICE_UID Device service property

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

- DeviceFunction.SERVICE_TYPE – mandatory service property. The service property value contains the device function type. For example, the sensor function can have different types like temperature or pressure etc. It's an optional property. The value type is java.lang.String.

  Organizations that want to use device function types that do not clash with OSGi Alliance defined types should prefix their types in own namespace.

- DeviceFunction.SERVICE_VERSION – optional service property. The service property value contains the device function version. That version can point to specific implementation version and vary in the different vendor implementations. The value type is java.lang.String.

- DeviceFunction.SERVICE_DEVICE_UID – optional service property. The property value is the device identifier. The device function belongs to this device. The value type is java.lang.String.

- DeviceFunction.SERVICE_REFERENCE_UIDS – optional service property. The service property value contains the reference device function unique identifiers. The value type is java.lang.String[]. It can be used to represent different relationships between the device functions.

- DeviceFunction.SERVICE_DESCRIPTION – optional service property. The property value is the device function description. The value type is java.lang.String.
- DeviceFunction.SERVICE_OPERATION_NAMES – optional service property. The property value is the device function operation names. 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.
- DeviceFunction.SERVICE_PROPERTY_NAMES – optional service property. The property value is the device function property names. The value type is java.lang.String[]. It's not possible to exist two or more Device Function properties with the same name.

The DeviceFunction services are registered before the Device service. It's possible that DeviceFunction.SERVICE_DEVICE_UID points to missing services at the moment of the registration. The reverse order is used when the services are unregistered. Device service is unregistered before the DeviceFunction services.

DeviceFunction service must be registered only under concrete device function classes. It's not allowed to register DeviceFunction service under classes, which are not concrete device functions. For example, those registrations are not allowed:

- context.registerService(new String[] {ManagedService.class.getName(), BinaryControl.class.getName()}, this, regProps); - ManagedService interface is not a device function interface;

- context.registerService(new String[] {DeviceFunction.class.getName(), BinaryControl.class.getName()}, this, regProps); - DeviceFunction interface is not concrete device function.

That one is valid context.registerService(new String[] {Meter.class.getName(), BinaryControl.class.getName()}, this, regProps);. Meter and BinaryControl are concrete device function interfaces. That rule helps to the applications to find all supported device function classes. Otherwise the DeviceFunction 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 unique identifier, type, version, description, link to the Device service and information about the reference device functions. DeviceFunction interface must be the base interface for all functions. If the device provider defines custom functions, all of them 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 callback.
- 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 callback.
- 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. 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.dal.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.SERVICE_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 or to override the property accessor methods. The operations are regular java methods. That implies that they have zero or more arguments and zero or one return value. The operation arguments and return value must follow the general type rules.

The operations can be optionally described with a set of meta data properties. Metadata is accessible with `DeviceFunction.getOperationMetadata(String)` method. The result provides metadata about the operation, operation arguments and result value. Operation arguments and result value are using the same metadata as the Device Function properties. The full details are defined in the next section.

5.5.3 Device Function Properties

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

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

- Getter methods must be available for all properties with `PropertyMetadata.META_INFO_PROPERTY_ACCESS_READABLE` access.
- Getter method must return a subclass of DeviceFunctionData.
- Setter methods must be available for all properties with `PropertyMetadata.META_INFO_PROPERTY_ACCESS_WRITABLE` access.
- Setter method must use `DeviceFunctionData` wrapped type. For example, there is `MyFunctionData` with timestamp, unit and `BigDecimal` value. The setter must accept as an argument the value of type `BigDecimal`.
- It's possible to have a second setter method, which accepts the value as a first argument and the unit as a second argument.
- No methods are required for properties with `PropertyMetadata.META_INFO_PROPERTY_ACCESS_EVENTABLE` access.

The accessor method names must be defined according JavaBeans specification [3].

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 `PropertyMetadata` with:

- Minimum value – available through `PropertyMetadata.getMinValue(String)`. The minimum value can be different for the different units.
- Maximum value – available through `PropertyMetadata.getMaxValue(String)`. The maximum value can be different for the different units.
- Enumeration of values – available through `PropertyMetadata.getEnumValues(String)`. The array of the possible values is sorted in increasing order according to the given unit.
- Resolution – available through `PropertyMetadata.getResolution(String)`. For example, if the range is [0, 100], the resolution can be 10. That's the different between two values in series. The resolution type depends on the property type. If the property is using data bean like `org.osgi.service.dal.functions.data.LevelData`, the resolution will the `BigDecimal`.
- Property access – available as a value in `PropertyMetadata.getMetadata(String)` result map. It's a bitmap of `java.lang.Integer` type and doesn't depend on the given unit. The access is available only for the Device Function properties and it's missing for the operation arguments and result metadata. The bitmap can be any combination of:
  - `PropertyMetadata.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.
  - `PropertyMetadata.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.

- ◦ PropertyMetadata.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.

- Unit - available as a value in PropertyMetadata.getMetadata() result map. The value contains the property supported units. The property value type is java.lang.String[]. Each unit must follow those rules:

  - ◦ The International System of Units must be used where it's applicable. For example, kg for kilogram and km for kilometre.
  - ◦ If the unit name matches to an Unicode symbol name, the Unicode symbol must be used. For example, the degree unit matches to the Unicode degree sign (\u00B0).
  - ◦ If the unit name doesn't match to an Unicode symbol, the unit symbol must be built by Unicode Basic Latin block of characters, superscript and subscript characters. For example, watt per square metre steradian is built by W/(m\u00B2 sr), where \u00B2 is Unicode superscript two.

  If those rules cannot be applied to the unit symbol, custom rules are allowed.

  A set of predefined unit symbols are available in Units interface.

- Description – available as a value in PropertyMetadata.getMetadata() result map. The property value type is java.lang.String and specifies an user readable description. It doesn't depend on the given unit.
- Vendor custom properties – available as a value in PropertyMetadata.getMetadata() result map and can depend on the given unit.

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:

- DeviceFunctionEvent.PROPERTY_FUNCTION_UID – the event source function unique identifier.
- DeviceFunctionEvent.PROPERTY_FUNCTION_PROPERTY_NAME – the property name.
- DeviceFunctionEvent.PROPERTY_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 {

     dal.function.UID=acme.function

     dal.function.property.name=”state”

     dal.function.property.value=ACMEFuntionData(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. They are about the control, monitoring and metering information.

5.6.1 BooleanControl Device Function

`BooleanControl` device function provides a binary control support. The property eventing must follow the definition in Device Function Property Event. The full function definition is available in the next table.

BooleanControl	
Name	**Description**
Operations	
reverse	Reverses the `BooleanControl` state. If the current state represents `true` value, it'll be reversed to `false`. If the current state represents `false` value, it'll be reversed to `true`.
setTrue	Sets the `BooleanControl` state to `true` value.
setFalse	Sets the `BooleanControl` state to `false` value.
Properties	
data	Contains the current state of `BooleanControl`. The property access can be: readable, writable and eventable.
Types	
light, door, window, power and custom types as they are defined in `org.osgi.service.dal.functions.Types`.	

`BooleanData` data structure is used to provide information about the function state. That data object contains the boolean value, the value collecting time and additional metadata. The immutable `BooleanData.value` field is accessible with `BooleanData.getValue()` getter.

The function class diagram is depicted on Illustration 13. The next code snippet sets to `true` all `BooleanControl` functions.

```
final ServiceReference[] booleanControlSRefs = context.getServiceReferences(
      BooleanControl.class.getName(), null);
if (null == booleanControlSRefs) {
  return; // no such services
}
for (int i = 0; i < booleanControlSRefs.length; i++) {
  final BooleanControl booleanControl = (BooleanControl) context.getService(
      binaryControlSRefs[i]);
  if (null != booleanControl) {
    booleanControl.setTrue();
  }
}
```

5.6.2 BooleanSensor Device Function

BooleanSensor device function provides binary sensor monitoring. It reports its state when an important event is available. There are no operations. The property eventing must follow the definition in Device Function Property Event. The full function definition is available in the next table.

BooleanSensor	
Name	**Description**
Properties	
data	Contains the current state of BooleanSensor. The property access can be: readable and eventable.
Types	
light, gas, smoke, door, window, power, rain, contact, fire, occupancy, water, motion and custom types as they are defined in org.osgi.service.dal.functions.Types.	

BooleanSensor and BooleanControl are using the same BooleanData data structure to provide information about the state. For more details see the definition in BooleanControl Device Function. The function class diagram is depicted on Illustration 13.

5.6.3 MultiLevelControl Device Function

MultiLevelControl device function provides multi-level control support. The property eventing must follow the definition in Device Function Property Event. The full function definition is available in the next table.

MultiLevelControl	
Name	**Description**
Properties	
data	Contains the current state of MultiLevelControl. The property access can be: readable, writable and eventable.
Types	
light, temperature, flow, pressure, humidity, gas, smoke, door, window, liquid, power, noisiness and custom types as they are defined in org.osgi.service.dal.functions.Types.	

LevelData data structure is used to provide information about the function level. That data object contains the BigDecimal value and the value unit. The measurement unit is used as it's defined in Device Function Properties. The immutable LevelData.unit field is accessible with LevelData.getUnit() getter. The immutable LevelData.level field is accessible with LevelData.getLevel() getter.

The function class diagram is depicted on Illustration 13.

5.6.4 MultiLevelSensor Device Function

MultiLevelSensor device function provides multi-level sensor monitoring. It reports its state when an important event is available. There are no operations. The property eventing must follow the definition in Device Function Property Event. The full function definition is available in the next table.

MultiLevelSensor

Name	Description
Properties	
data	Contains the current state of `MultiLevelSensor`. The property access can be: readable and eventable.
Types	
light, temperature, flow, pressure, humidity, gas, smoke, door, window, liquid, power, noisiness, rain and custom types as they are defined in `org.osgi.service.dal.functions.Types`.	

`MultiLevelSensor` and `MultiLevelControl` are using the same `LevelData` data structure to provide information about the level. For more details see the definition in MultiLevelControl Device Function. The function class diagram is depicted on Illustration 13.

5.6.5 Meter Device Function

`Meter` device function can measure metering information.

Meter	
Name	**Description**
Operations	
resetTotal	Resets the total metering info.
Properties	
total	Contains the total consumption. It has been measured since the last call of resetTotal or device initial run. The property access is readable.
current	Contains the current consumption. The property is readable.
Service Properties	
meter.flow	Contains the metering flow. Currently, it can be “in” and “out”.
Types	
pressure, gas, power, water, heat, cold and custom types as they are defined in `org.osgi.service.dal.functions.Types`.	

`Meter` device function is using the same `LevelData` data structure as `MultiLevelSensor` and `MultiLevelControl` to provide metering information. For more details see the definition in MultiLevelControl Device Function. The property eventing must follow the definition in Device Function Property Event. The function class diagram is depicted on Illustration 13.

5.6.6 Alarm Device Function

`Alarm` device function provides alarm sensor support. There is only one eventable property and no operations. The property eventing must follow the definition in Device Function Property Event.

Alarm	
Name	**Description**
Properties	

alarm	Specifies the alarm property name. The property is eventable.

`AlarmData` data structure is used to provide information about the available alarm. That data object contains the alarm type and severity.

The function class diagram is depicted on Illustration 13.

5.6.7 Keypad Device Function

`Keypad` device function provides support for keypad control. A keypad typically consists of one or more keys/buttons, which can be discerned. Different types of key presses like short and long press can typically also be detected. There is only one eventable property and no operations. The property eventing must follow the definition in Device Function Property Event.

Keypad	
Name	**Description**
Properties	
key	Specifies a property name for a key from the keypad. The property is eventable.

`KeypadData` data structure is used to provide information when a change with some key from device keypad has occurred. That data object contains the event type, key code and key name. Currently, there are a few predefined event types:

- `EVENT_TYPE_PRESSED` – used for a key pressed;
- `EVENT_TYPE_PRESSED_LONG` – used for a long key pressed;
- `EVENT_TYPE_PRESSED_DOUBLE` – used for a double key pressed;
- `EVENT_TYPE_PRESSED_DOUBLE_LONG` – used for a double and long key pressed;
- `EVENT_TYPE_RELEASED` – used for a key released.

The function class diagram is depicted on Illustration 13.

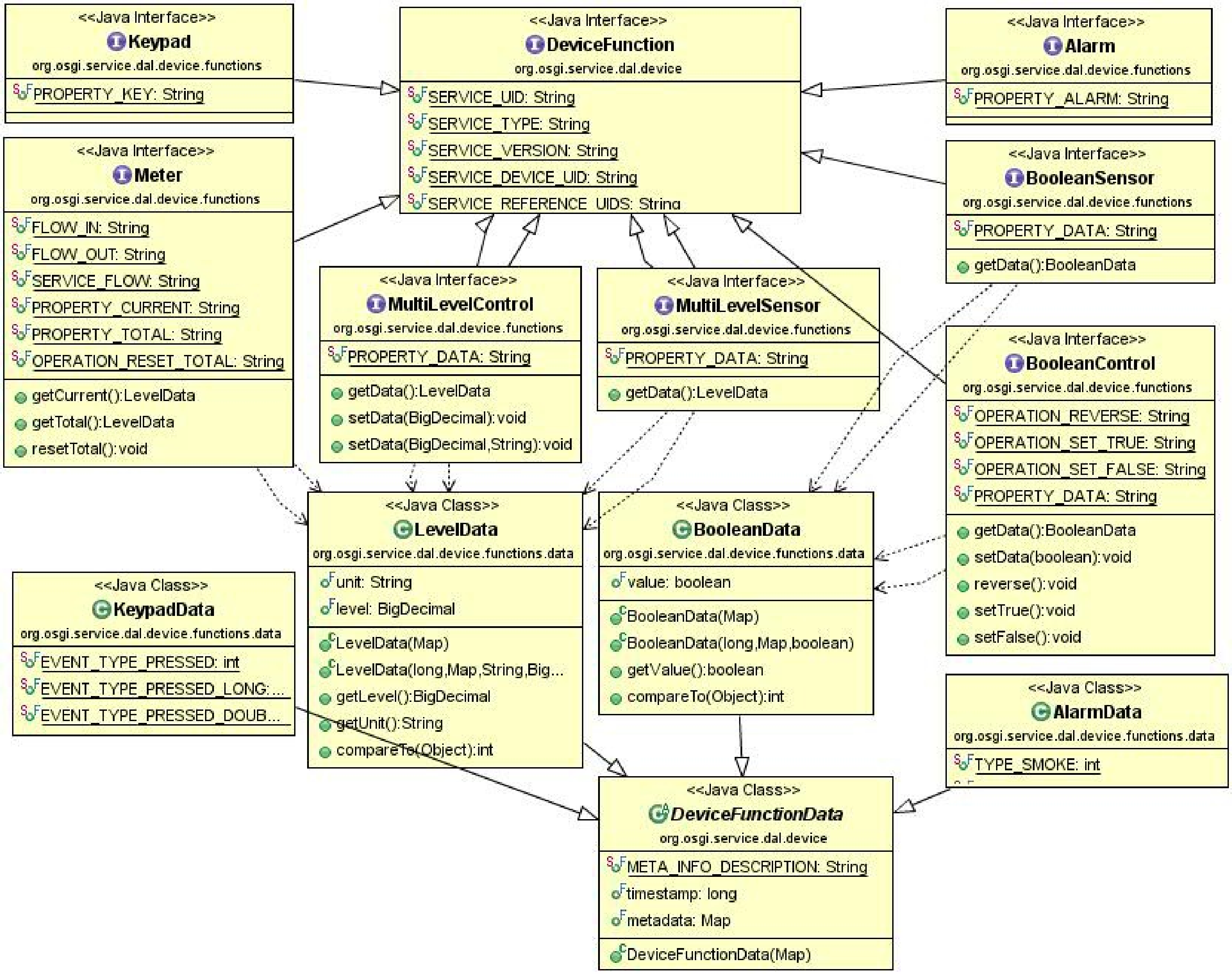

Illustration 13

6 Data Transfer Objects

TODO: Do we need those objects?

7 Javadoc

OSGi Javadoc

1/16/14 4:11 PM

Package Summary		Page
org.osgi.service.dal	Device Package Version 1.0.	36
org.osgi.service.dal.functions	Device Functions 1.0.	91
org.osgi.service.dal.functions.data	Device Function Data 1.0.	116

Package org.osgi.service.dal

Device Package Version 1.0.

See:

Description

Interface Summary		Page
Device	Represents the device in the OSGi service registry.	37
DeviceFunction	Device Function service provides specific device operations and properties.	48
OperationMetadata	Contains metadata about Device Function operation.	64
PropertyMetadata	Contains metadata about Device Function property or Device Function operation parameter.	66
Units	Contains the most of the International System of Units unit symbols.	70

Class Summary		Page
DeviceFunctionData	Abstract `DeviceFunction` data wrapper.	53
DeviceFunctionEvent	Asynchronous event, which marks a Device Function property value modification.	56
DevicePermission	A bundle's authority to perform specific privileged administrative operations on the devices.	60

Exception Summary		Page
DeviceException	`DeviceException` is a special `IOException`, which is thrown to indicate that there is a device operation fail.	45

Package org.osgi.service.dal Description

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.dal; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

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

Interface Device

org.osgi.service.dal

```
public interface Device
```

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

Field Summary		Page
`String`	**DEVICE_CATEGORY** Constant for the value of the `org.osgi.service.Types.Constants.DEVICE_CATEGORY` service property.	38
`String`	**SERVICE_DESCRIPTION** The service property value contains the device description.	41
`String`	**SERVICE_DRIVER** The service property value contains the device driver name.	39
`String`	**SERVICE_FIRMWARE_VENDOR** The service property value contains the device firmware vendor.	40
`String`	**SERVICE_FIRMWARE_VERSION** The service property value contains the device firmware version.	40
`String`	**SERVICE_HARDWARE_VENDOR** The service property value contains the device hardware vendor.	40
`String`	**SERVICE_HARDWARE_VERSION** The service property value contains the device hardware version.	40
`String`	**SERVICE_MODEL** The service property value contains the device model.	41
`String`	**SERVICE_NAME** The service property value contains the device name.	39
`String`	**SERVICE_REFERENCE_UIDS** The service property value contains the reference device unique identifiers.	39
`String`	**SERVICE_SERIAL_NUMBER** The service property value contains the device serial number.	41
`String`	**SERVICE_STATUS** The service property value contains the device status.	39
`String`	**SERVICE_STATUS_DETAIL** The service property value contains the device status detail.	40
`String`	**SERVICE_TYPES** The service property value contains the device types like DVD, TV etc.	40
`String`	**SERVICE_UID** The service property value contains the device unique identifier.	39
`Integer`	**STATUS_DETAIL_CONFIGURATION_NOT_APPLIED** Device status detail indicates that the device configuration is not applied.	42
`Integer`	**STATUS_DETAIL_CONNECTING** Device status detail indicates that the device is currently connecting to the network.	42
`Integer`	**STATUS_DETAIL_DEVICE_BROKEN** Device status detail indicates that the device is broken.	42

Integer	**STATUS_DETAIL_DEVICE_COMMUNICATION_ERROR** Device status detail indicates that the device communication is problematic.	43
Integer	**STATUS_DETAIL_DEVICE_DATA_INSUFFICIENT** Device status detail indicates that the device doesn't provide enough information and cannot be determined.	43
Integer	**STATUS_DETAIL_DEVICE_NOT_ACCESSIBLE** Device status detail indicates that the device is not accessible and further communication is not possible.	43
Integer	**STATUS_DETAIL_ERROR_APPLYING_CONFIGURATION** Device status detail indicates that the device cannot be configured.	43
Integer	**STATUS_DETAIL_IN_DUTY_CYCLE** Device status detail indicates that the device is in duty cycle.	43
Integer	**STATUS_DETAIL_INITIALIZING** Device status detail indicates that the device is currently in process of initialization.	42
Integer	**STATUS_DETAIL_REMOVING** Device status detail indicates that the device is leaving the network.	42
Integer	**STATUS_NOT_CONFIGURED** Device status indicates that the device is currently not configured.	42
Integer	**STATUS_NOT_INITIALIZED** Device status indicates that the device is currently not initialized.	42
Integer	**STATUS_OFFLINE** Device status indicates that the device is currently not available for operations.	41
Integer	**STATUS_ONLINE** Device status indicates that the device is currently available for operations.	41
Integer	**STATUS_PROCESSING** Device status indicates that the device is currently busy with an operation.	41
Integer	**STATUS_REMOVED** Device status indicates that the device is removed from the network.	41

Method Summary		*Page*
Object	**getProperty**(String propName) Returns the current value of the specified property.	43
void	**remove**() Removes this device.	44
void	**setName**(String name) Sets the device name.	44

Field Detail

DEVICE_CATEGORY

`public static final String` **`DEVICE_CATEGORY`** `= "DAL"`

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

See Also:

`org.osgi.service.Types.Constants.DEVICE_CATEGORY`

SERVICE_UID

```
public static final String SERVICE_UID = "dal.device.UID"
```

The service property value contains the device 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:

UID ::= driver-name ':' device-id

UID - device unique identifier

driver-name - the value of the `SERVICE_DRIVER` service property

device-id - device unique identifier in the scope of the driver

SERVICE_REFERENCE_UIDS

```
public static final String SERVICE_REFERENCE_UIDS = "dal.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[]`. It can be used to represent different relationships between the devices. For example, the ZigBee controller can have a reference to the USB dongle.

SERVICE_DRIVER

```
public static final String SERVICE_DRIVER = "dal.device.driver"
```

The service property value contains the device driver name. For example, ZigBee, Z-Wave, Bluetooth etc. It's a mandatory property. The value type is `java.lang.String`.

SERVICE_NAME

```
public static final String SERVICE_NAME = "dal.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 set with `setName(String)` method.

SERVICE_STATUS

```
public static final String SERVICE_STATUS = "dal.device.status"
```

The service property value contains the device status. It's a mandatory property. The value type is `java.lang.Integer`. The possible values are:

- `STATUS_ONLINE`
- `STATUS_OFFLINE`
- `STATUS_REMOVED`
- `STATUS_PROCESSING`
- `STATUS_NOT_INITIALIZED`
- `STATUS_NOT_CONFIGURED`

SERVICE_STATUS_DETAIL

`public static final String **SERVICE_STATUS_DETAIL** = "dal.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`. 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`.

SERVICE_HARDWARE_VENDOR

`public static final String **SERVICE_HARDWARE_VENDOR** = "dal.device.hardware.vendor"`

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

SERVICE_HARDWARE_VERSION

`public static final String **SERVICE_HARDWARE_VERSION** = "dal.device.hardware.version"`

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

SERVICE_FIRMWARE_VENDOR

`public static final String **SERVICE_FIRMWARE_VENDOR** = "dal.device.firmware.vendor"`

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

SERVICE_FIRMWARE_VERSION

`public static final String **SERVICE_FIRMWARE_VERSION** = "dal.device.firmware.version"`

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

SERVICE_TYPES

`public static final String **SERVICE_TYPES** = "dal.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[]`.

SERVICE_MODEL

`public static final String` **`SERVICE_MODEL`** `= "dal.device.model"`

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

SERVICE_SERIAL_NUMBER

`public static final String` **`SERVICE_SERIAL_NUMBER`** `= "dal.device.serial.number"`

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

SERVICE_DESCRIPTION

`public static final String` **`SERVICE_DESCRIPTION`** `= "dal.device.description"`

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

STATUS_REMOVED

`public static final Integer` **`STATUS_REMOVED`**

Device status indicates that the device is removed from the network. That status must be set as the last device status and after that the device service can be unregistered from the service registry. It can be used as a value of `SERVICE_STATUS` service property.

STATUS_OFFLINE

`public static final Integer` **`STATUS_OFFLINE`**

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

STATUS_ONLINE

`public static final Integer` **`STATUS_ONLINE`**

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

STATUS_PROCESSING

`public static final Integer` **`STATUS_PROCESSING`**

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

STATUS_NOT_INITIALIZED

```
public static final Integer STATUS_NOT_INITIALIZED
```

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 SERVICE_STATUS service property.

STATUS_NOT_CONFIGURED

```
public static final Integer STATUS_NOT_CONFIGURED
```

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 SERVICE_STATUS service property.

STATUS_DETAIL_CONNECTING

```
public static final Integer STATUS_DETAIL_CONNECTING
```

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

STATUS_DETAIL_INITIALIZING

```
public static final Integer STATUS_DETAIL_INITIALIZING
```

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

STATUS_DETAIL_REMOVING

```
public static final Integer STATUS_DETAIL_REMOVING
```

Device status detail indicates that the device is leaving the network. It can be used as a value of SERVICE_STATUS_DETAIL service property. The device status must be STATUS_PROCESSING.

STATUS_DETAIL_CONFIGURATION_NOT_APPLIED

```
public static final Integer STATUS_DETAIL_CONFIGURATION_NOT_APPLIED
```

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

STATUS_DETAIL_DEVICE_BROKEN

```
public static final Integer STATUS_DETAIL_DEVICE_BROKEN
```

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

STATUS_DETAIL_DEVICE_COMMUNICATION_ERROR

```
public static final Integer STATUS_DETAIL_DEVICE_COMMUNICATION_ERROR
```

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

STATUS_DETAIL_DEVICE_DATA_INSUFFICIENT

```
public static final Integer STATUS_DETAIL_DEVICE_DATA_INSUFFICIENT
```

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

STATUS_DETAIL_DEVICE_NOT_ACCESSIBLE

```
public static final Integer STATUS_DETAIL_DEVICE_NOT_ACCESSIBLE
```

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

STATUS_DETAIL_ERROR_APPLYING_CONFIGURATION

```
public static final Integer STATUS_DETAIL_ERROR_APPLYING_CONFIGURATION
```

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

STATUS_DETAIL_IN_DUTY_CYCLE

```
public static final Integer STATUS_DETAIL_IN_DUTY_CYCLE
```

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

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.

---

setName

```
void setName(String name)
      throws DeviceException,
             UnsupportedOperationException,
             SecurityException,
             IllegalStateException
```

Sets the device name. The method must synchronously update the `SERVICE_NAME` service property of this device. The new name must be persistently stored. It'll set after framework restart. `null` name will clean up the current device name.

Parameters:
`name` - The new device name or `null` to clean up the name.

Throws:
`DeviceException` - 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, DevicePermission.ACTION_SET_NAME]` and the Java Runtime Environment supports permissions.
`IllegalStateException` - If this device service object has already been unregistered.

---

remove

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

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

Throws:
`DeviceException` - 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, DevicePermission.ACTION_REMOVE]` and the Java Runtime Environment supports permissions.
`IllegalStateException` - If this device service object has already been unregistered.

Class DeviceException

org.osgi.service.dal

```
java.lang.Object
  └java.lang.Throwable
      └java.lang.Exception
          └java.io.IOException
              └org.osgi.service.dal.DeviceException
```

All Implemented Interfaces:

Serializable

```
public class DeviceException
extends IOException
```

`DeviceException` is a special `IOException`, which is thrown to indicate that there is a device operation fail. The error reason can be located with `getCode()` method. The cause is available with `getCause()`.

Field Summary		Page
static int	**CODE_COMMUNICATION_ERROR** An exception code indicates that there is an error in the communication.	46
static int	**CODE_NO_DATA** An exception code indicates that the requested value is currently not available.	46
static int	**CODE_NOT_INITIALIZED** An exception code indicates that the device is not initialized.	46
static int	**CODE_TIMEOUT** An exception code indicates that there is expired timeout without any processing.	46
static int	**CODE_UNKNOWN** An exception code indicates that the error is unknown.	45

Constructor Summary	Page
DeviceException()	46

Method Summary		Page
Throwable	**getCause**() Returns the cause for this exception or `null` if the cause is missing.	46
int	**getCode**() Returns the exception error code.	46

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 there is expired timeout without any processing.

CODE_NOT_INITIALIZED

`public static final int` **`CODE_NOT_INITIALIZED`** `= 4`

An exception code indicates that the device is not initialized. The device status is `Device.STATUS_NOT_INITIALIZED` or `Device.STATUS_PROCESSING`.

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

DeviceException

`public` **`DeviceException`**`()`

Method Detail

getCode

`public int` **`getCode`**`()`

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

Returns:

An exception code.

---

getCause

`public Throwable` **`getCause`**`()`

Returns the cause for this exception 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.

Interface DeviceFunction

org.osgi.service.dal

All Known Subinterfaces:

Alarm, BooleanControl, BooleanSensor, Keypad, Meter, MultiLevelControl, MultiLevelSensor

---

`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:

- `SERVICE_UID` - mandatory service property. The property value contains the device function unique identifier.
- `SERVICE_DEVICE_UID` - optional service property. The property value is the Functional Device identifiers. The Device Function belongs to those devices.
- `SERVICE_REFERENCE_UIDS` - optional service property. The property value contains the reference device function unique identifiers.
- `SERVICE_TYPE` - mandatory service property. The property value is the function type.
- `SERVICE_VERSION` - optional service property. The property value contains the function version.
- `SERVICE_DESCRIPTION` - optional service property. The property value is the device function description.
- `SERVICE_OPERATION_NAMES` - optional service property. The property value is the Device Function operation names.
- `SERVICE_PROPERTY_NAMES` - optional service property. The property value is the Device Function property names.

The `DeviceFunction` services are registered before the `Device` services. It's possible that `SERVICE_DEVICE_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 `Device` 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(), BooleanControl.class.getName()}, this, regProps);` - `ManagedService` interface is not a Device Function interface;
- `context.registerService(new String[] {DeviceFunction.class.getName(), BooleanControl.class.getName()}, this, regProps);` - `DeviceFunction` interface is not concrete Device Function interface.

That one is a valid registration: `context.registerService(new String[] {Meter.class.getName(), BooleanControl.class.getName()}, this, regProps);`. `Meter` and `BooleanControl` 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 `PropertyMetadata.META_INFO_PROPERTY_ACCESS_READABLE` access.
- Getter method must return a subclass of `DeviceFunctionData`.
- Setter methods must be available for all properties with `PropertyMetadata.META_INFO_PROPERTY_ACCESS_WRITABLE` access.

- Setter method must use DeviceFunctionData wrapped type. For example, there is MyFunctionData with timestamp, unit and BigDecimal value. The setter must accept as an argument the value of type BigDecimal.
- It's possible to have a second setter method, which accepts the value as a first argument and the unit as a second argument.
- No methods are required for properties with PropertyMetadata.META_INFO_PROPERTY_ACCESS_EVENTABLE access.

The accessor method names 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 parameters and zero or one return value.

Operation arguments and Device Function properties are restricted by the same set of rules. 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.Maps. 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 metadata is accessible with getPropertyMetadata(String). The operations metadata is accessible with getOperationMetadata(String). 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 callback.
- 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 callback.
- 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	**SERVICE_DESCRIPTION** The service property value contains the device function description.	*51*
String	**SERVICE_DEVICE_UID** The service property value contains the device unique identifier.	*51*
String	**SERVICE_OPERATION_NAMES** The service property value contains the device function operation names.	*51*
String	**SERVICE_PROPERTY_NAMES** The service property value contains the device function property names.	*51*
String	**SERVICE_REFERENCE_UIDS** The service property value contains the reference device function unique identifiers.	*51*
String	**SERVICE_TYPE** The service property value contains the device function type.	*50*

String	**SERVICE_UID** The service property value contains the device function unique identifier.	50
String	**SERVICE_VERSION** The service property value contains the device function version.	50

Method Summary		Page
OperationMetadata	**getOperationMetadata**(String operationName) Provides metadata about the Device Function operation.	52
Object	**getProperty**(String propName) Returns the current value of the specified property.	52
PropertyMetadata	**getPropertyMetadata**(String propertyName) Provides metadata about the Device Function property specified with the name argument.	51

Field Detail

SERVICE_UID

```
public static final String SERVICE_UID = "dal.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 `Device.SERVICE_UID` Functional Device service property

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

SERVICE_TYPE

```
public static final String SERVICE_TYPE = "dal.function.type"
```

The service property value contains the device function type. It's an optional property. For example, the sensor function can have different types like temperature or pressure etc. The value type is `java.lang.String`.

Organizations that want to use device function types that do not clash with OSGi Alliance defined types should prefix their types in own namespace.

SERVICE_VERSION

```
public static final String SERVICE_VERSION = "dal.function.version"
```

The service property value contains the device function version. That version can point to specific implementation version and vary in the different vendor implementations. It's an optional property. The value type is `java.lang.String`.

SERVICE_DEVICE_UID

```
public static final String SERVICE_DEVICE_UID = "dal.function.device.UID"
```

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

SERVICE_REFERENCE_UIDS

```
public static final String SERVICE_REFERENCE_UIDS = "dal.function.reference.UIDs"
```

The service property value contains the reference device function unique identifiers. It's an optional property. The value type is `java.lang.String[]`. It can be used to represent different relationships between the device functions.

SERVICE_DESCRIPTION

```
public static final String SERVICE_DESCRIPTION = "dal.function.description"
```

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

SERVICE_OPERATION_NAMES

```
public static final String SERVICE_OPERATION_NAMES = "dal.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.

SERVICE_PROPERTY_NAMES

```
public static final String SERVICE_PROPERTY_NAMES = "dal.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

```
PropertyMetadata getPropertyMetadata(String propertyName)
                              throws IllegalArgumentException
```

Provides metadata about the Device Function property specified with the name argument.

This method must continue to return the property metadata after the Device Function service has been unregistered.

Parameters:

`propertyName` - The function property name, which metadata is requested.

Returns:
The property metadata for the given property name. `null` if the property metadata is not supported.
Throws:
`IllegalArgumentException` - If the function property with the specified name is not supported.

---

getOperationMetadata

```
OperationMetadata getOperationMetadata(String operationName)
                                throws IllegalArgumentException
```

Provides metadata about the Device Function operation.

This method must continue to return the operation metadata after the Device Function service has been unregistered.

Parameters:
`operationName` - The function operation name, which metadata is requested.
Returns:
The operation metadata for the given operation name. `null` if the operation metadata is not supported.
Throws:
`IllegalArgumentException` - If the function operation with the specified name is not supported.

---

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 function.

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

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

Class DeviceFunctionData

org.osgi.service.dal

```
java.lang.Object
  └org.osgi.service.dal.DeviceFunctionData
```

All Implemented Interfaces:

Comparable

Direct Known Subclasses:

AlarmData, BooleanData, KeypadData, LevelData

```
abstract public class DeviceFunctionData
extends Object
implements Comparable
```

Abstract `DeviceFunction` data wrapper. A subclass must be used for an access to the property values by all Device Functions. It takes care about the timestamp and additional metadata. The subclasses are responsible to provide concrete value and unit if required.

The subclass is responsible to provide correct implementation of `Comparable.compareTo(Object)` method.

Field Summary		Page
static String	**META_INFO_DESCRIPTION** Metadata key, which value represents the data description.	*53*
Map	**metadata** Contains `DeviceFunctionData` metadata.	*54*
long	**timestamp** Contains `DeviceFunctionData` timestamp.	*54*

Constructor Summary	Page
DeviceFunctionData(Map fields) Constructs new `DeviceFunctionData` instance with the specified field values.	*54*
DeviceFunctionData(long timestamp, Map metadata) Constructs new `DeviceFunctionData` instance with the specified arguments.	*54*

Method Summary		Page
Map	**getMetadata**() Returns `DeviceFunctionData` metadata.	*55*
long	**getTimestamp**() Returns `DeviceFunctionData` timestamp.	*54*

Field Detail

META_INFO_DESCRIPTION

```
public static final String META_INFO_DESCRIPTION = "description"
```

Metadata key, which value represents the data description. The property value type is `java.lang.String`.

timestamp

```
public final long timestamp
```

Contains `DeviceFunctionData` timestamp. The timestamp is the difference between the value collecting time and midnight, January 1, 1970 UTC. It's measured in milliseconds. The device driver is responsible to generate that value when the value is received from the device. `Long.MIN_VALUE` value means no timestamp.

metadata

```
public final Map metadata
```

Contains `DeviceFunctionData` metadata. It's dynamic metadata related only to this specific value. Possible keys:

- `META_INFO_DESCRIPTION`
- custom key

Constructor Detail

DeviceFunctionData

```
public DeviceFunctionData(Map fields)
```

Constructs new `DeviceFunctionData` instance with the specified field values. The map keys must match to the field names. The map values will be assigned to the appropriate class fields. For example, the maps can be: {"timestamp"=Long(1384440775495)}. That map will initialize the "timestamp" field with 1384440775495.

Parameters:

`fields` - Contains the new `DeviceFunctionData` instance field values.

DeviceFunctionData

```
public DeviceFunctionData(long timestamp,
                          Map metadata)
```

Constructs new `DeviceFunctionData` instance with the specified arguments.

Parameters:

`timestamp` - The data timestamp.
`metadata` - The data metadata.

Method Detail

getTimestamp

```
public long getTimestamp()
```

Returns `DeviceFunctionData` timestamp. The timestamp is the difference between the value collecting time and midnight, January 1, 1970 UTC. It's measured in milliseconds. The device driver is responsible to generate that value when the value is received from the device. `Long.MIN_VALUE` value means no timestamp.

Returns:

`DeviceFunctionData` timestamp.

---

getMetadata

```
public Map getMetadata()
```

Returns `DeviceFunctionData` metadata. It's dynamic metadata related only to this specific value. Possible keys:

- META_INFO_DESCRIPTION
- custom key

Returns:

`DeviceFunctionData` metadata or `null` is there is no metadata.

Class DeviceFunctionEvent

org.osgi.service.dal

```
java.lang.Object
  └org.osgi.service.event.Event
      └org.osgi.service.dal.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:

- PROPERTY_FUNCTION_UID - the event source function unique identifier.
- PROPERTY_FUNCTION_PROPERTY_NAME - the property name.
- PROPERTY_FUNCTION_PROPERTY_VALUE - the property value. The property value type must be a subclass of DeviceFunctionData.

Field Summary		Page
static String	**EVENT_CLASS** Represents the event class.	57
static String	**EVENT_PACKAGE** Represents the event package.	57
static String	**PROPERTY_FUNCTION_PROPERTY_NAME** Represents an event property key for the Device Function property name.	57
static String	**PROPERTY_FUNCTION_PROPERTY_VALUE** Represents an event property key for the Device Function property value.	57
static String	**PROPERTY_FUNCTION_UID** Represents an event property key for Device Function UID.	57
static String	**TOPIC_PROPERTY_CHANGED** Represents the event topic for the Device Function property changed.	57

Constructor Summary	Page
DeviceFunctionEvent(String topic, Dictionary properties) Constructs a new event with the specified topic and properties.	58
DeviceFunctionEvent(String topic, Map properties) Constructs a new event with the specified topic and properties.	58

Method Summary		Page
String	**getFunctionUID**() Returns the property value change source function identifier.	58
String	**getPropertyName**() Returns the property name.	58
DeviceFunctionData	**getPropertyValue**() Returns the property value.	58

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_FUNCTION_UID

`public static final String` **`PROPERTY_FUNCTION_UID`** `= "dal.function.UID"`

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

PROPERTY_FUNCTION_PROPERTY_NAME

`public static final String` **`PROPERTY_FUNCTION_PROPERTY_NAME`** `= "dal.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_FUNCTION_PROPERTY_VALUE

`public static final String` **`PROPERTY_FUNCTION_PROPERTY_VALUE`** `= "dal.function.property.value"`

Represents an event property key for the Device Function property value. The property value type is a subclass of `DeviceFunctionData`. 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

getFunctionUID

```
public String getFunctionUID()
```

Returns the property value change source function identifier. The value is same as the value of `PROPERTY_FUNCTION_UID` 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_FUNCTION_PROPERTY_NAME`.

Returns:

The property name.

getPropertyValue

```
public DeviceFunctionData getPropertyValue()
```

Returns the property value. The value is same as the value of `PROPERTY_FUNCTION_PROPERTY_VALUE`.

Returns:

The property value.

Class DevicePermission

org.osgi.service.dal

```
java.lang.Object
  └java.security.Permission
      └java.security.BasicPermission
          └org.osgi.service.dal.DevicePermission
```

All Implemented Interfaces:

Guard, Serializable

```
final public class DevicePermission
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	Device.remove()
ACTION_SET_NAME	Device.setName(String)

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_REMOVE** A permission action to remove the device.	61
static String	**ACTION_SET_NAME** A permission action to modify the device name.	61

Constructor Summary	Page
DevicePermission(String filter, String actions) Creates a new `FunctionalDevicePermission` with the given filter and actions.	61
DevicePermission(Device device, String actions) Creates a new `FunctionalDevicePermission` with the given device and actions.	61

Method Summary		Page
boolean	**equals**(Object obj) Two `FunctionalDevicePermission` instances are equal if: $\frac{35}{17}$ represents the same filter and actions $\frac{35}{17}$ represents the same device and actions	62
String	**getActions**() Returns the canonical string representation of the actions.	62
int	**hashCode**() Returns the hash code value for this object.	62
boolean	**implies**(Permission p) Determines if the specified permission is implied by this object.	62

Permission Collection	**newPermissionCollection**() Returns a new `PermissionCollection` suitable for storing `FunctionalDevicePermission` instances.	*63*

Field Detail

ACTION_SET_NAME

```
public static final String ACTION_SET_NAME = "setName"
```

A permission action to modify the device name.

ACTION_REMOVE

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

A permission action to remove the device.

Constructor Detail

DevicePermission

```
public DevicePermission(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: (dal.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_SET_NAME` and `ACTION_REMOVE`. Any combinations are allowed.

Throws:

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

DevicePermission

```
public DevicePermission(Device 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_SET_NAME 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_SET_NAME, 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 DevicePermission(Device, 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 `DevicePermission(Device, String)`.

Returns:

`true` if the specified permission is implied by this permission, `false` otherwise.

Throws:

`IllegalArgumentException` - If the specified permission is not constructed by `DevicePermission(Device, 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 OperationMetadata

org.osgi.service.dal

`public interface` **`OperationMetadata`**

Contains metadata about Device Function operation.

See Also:

`DeviceFunction`, `PropertyMetadata`

Field Summary		***Pag e***
`String`	**`META_INFO_DESCRIPTION`** Metadata key, which value represents the operation description.	*64*

Method Summary		***Pag e***
`Map`	**`getMetadata`**`()` Returns metadata about the Device Function operation.	*64*
`PropertyMetadata[]`	**`getParametersMetadata`**`()` Returns metadata about the operation parameters or `null` if no such medatadata is available.	*65*
`PropertyMetadata`	**`getReturnValueMetadata`**`()` Returns metadata about the operation return value or `null` if no such metadata is available.	*65*

Field Detail

META_INFO_DESCRIPTION

`public static final String` **`META_INFO_DESCRIPTION`** `= "description"`

Metadata key, which value represents the operation description. The property value type is `java.lang.String`.

Method Detail

getMetadata

`Map` **`getMetadata`**`()`

Returns metadata about the Device Function operation. The keys of the `java.util.Map` result must be of `java.lang.String` type. Possible keys:

- `META_INFO_DESCRIPTION`
- custom key

Returns:

The operation metadata or `null` if no such metadata is available.

---

getReturnValueMetadata

`PropertyMetadata getReturnValueMetadata()`

Returns metadata about the operation return value or `null` if no such metadata is available.

Returns:
Operation return value metadata.

---

getParametersMetadata

`PropertyMetadata[] getParametersMetadata()`

Returns metadata about the operation parameters or `null` if no such medatadata is available.

Returns:
Operation parameters medata.

Interface PropertyMetadata

org.osgi.service.dal

```
public interface PropertyMetadata
```

Contains metadata about Device Function property or Device Function operation parameter. The access to the Device Function properties is a bitmap value of `META_INFO_PROPERTY_ACCESS` metadata 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. `DeviceFunctionEvent`s are sent on property change.

See Also:

`DeviceFunction`, `PropertyMetadata`

Field Summary		***Page***
String	**META_INFO_DESCRIPTION** Metadata key, which value represents the property description.	*67*
String	**META_INFO_PROPERTY_ACCESS** Metadata key, which value represents the access to the Device Function property.	*67*
int	**META_INFO_PROPERTY_ACCESS_EVENTABLE** Marks the eventable Device Function properties.	*67*
int	**META_INFO_PROPERTY_ACCESS_READABLE** Marks the readable Device Function properties.	*67*
int	**META_INFO_PROPERTY_ACCESS_WRITABLE** Marks the writable Device Function properties.	*67*
String	**META_INFO_UNITS** Metadata key, which value represents the property supported units.	*68*

Method Summary		***Page***
DeviceFunctionData[]	**getEnumValues**(String unit) Returns the property possible values according to the specified unit.	*69*
DeviceFunctionData	**getMaxValue**(String unit) Returns the property maximum value according to the specified unit.	*69*
Map	**getMetadata**(String unit) Returns metadata about the Device Function property or operation parameter.	*68*
DeviceFunctionData	**getMinValue**(String unit) Returns the property minimum value according to the specified unit.	*69*
Object	**getResolution**(String unit) Returns the resolution value of specific range.	*68*

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.

See Also:
DeviceFunction

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 methods.

See Also:
DeviceFunction

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.

See Also:
DeviceFunction

META_INFO_PROPERTY_ACCESS

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

Metadata 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.

The property access is available only for Device Function properties and it's missing for the operation parameters.

META_INFO_DESCRIPTION

```
public static final String META_INFO_DESCRIPTION = "description"
```

Metadata key, which value represents the property description. The property value type is `java.lang.String`.

META_INFO_UNITS

```
public static final String META_INFO_UNITS = "units"
```

Metadata key, which value represents the property supported units. The property value type is `java.lang.String[]`. Each unit must follow those rules:

- The International System of Units must be used where it's applicable. For example, kg for kilogram and km for kilometre.
- If the unit name matches to an Unicode symbol name, the Unicode symbol must be used. For example, the degree unit matches to the Unicode degree sign (°).
- If the unit name doesn't match to an Unicode symbol, the unit symbol must be built by Unicode Basic Latin block of characters, superscript and subscript characters. For example, watt per square metre steradian is built by W/(m^2 sr), where ^{2} is Unicode superscript two.

If those rules cannot be applied to the unit symbol, custom rules are allowed. A set of predefined unit symbols are available in `Units` interface.

Method Detail

getMetadata

```
Map getMetadata(String unit)
```

Returns metadata about the Device Function property or operation parameter. The keys of the `java.util.Map` result must be of `java.lang.String` type. Possible keys:

- `META_INFO_DESCRIPTION` - doesn't depend on the given unit.
- `META_INFO_PROPERTY_ACCESS` - available only for Device Function property and missing for Device FUnction operation parameters. It doesn't depend on the given unit.
- `META_INFO_UNITS` - doesn't depend on the given unit.
- custom key - can depend on the unit.

Parameters:

`unit` - The unit to align the metadata if it's applicable. It can be null, which means that the default unit will be used.

Returns:

The property metadata or `null` if no such metadata is available.

---

getResolution

```
Object getResolution(String unit)
              throws IllegalArgumentException
```

Returns the resolution value of specific range. For example, if the range is [0, 100], the resolution can be 10. That's the different between two values in series. The resolution type depends on the property type. If the property is using data bean like `LevelData`, the resolution will the `BigDecimal`.

Parameters:

`unit` - The unit to align the resolution, can be `null`.

Returns:

The resolution according to the specified unit or `null` if no resolution is supported.

Throws:

`IllegalArgumentException` - If the unit is not supported.

---

getEnumValues

```
DeviceFunctionData[] getEnumValues(String unit)
                            throws IllegalArgumentException
```

Returns the property possible values according to the specified unit. If the unit is `null`, the values set is aligned to the default unit. If there is no such set of supported values, `null` is returned. The values must be sorted in increasing order.

Parameters:

`unit` - The unit to align the supported values, can be `null`.

Returns:

The supported values according to the specified unit or `null` if no such values are supported. The values must be sorted in increasing order.

Throws:

`IllegalArgumentException` - If the unit is not supported.

---

getMinValue

```
DeviceFunctionData getMinValue(String unit)
                        throws IllegalArgumentException
```

Returns the property minimum value according to the specified unit. If the unit is `null`, the minimum value is aligned to the default unit. If there is no minimum value, `null` is returned.

Parameters:

`unit` - The unit to align the minimum value, can be `null` .

Returns:

The minimum value according to the specified unit or `null` if no minimum value is supported.

Throws:

`IllegalArgumentException` - If the unit is not supported.

---

getMaxValue

```
DeviceFunctionData getMaxValue(String unit)
                        throws IllegalArgumentException
```

Returns the property maximum value according to the specified unit. If the unit is `null`, the maximum value is aligned to the default unit. If there is no maximum value, `null` is returned.

Parameters:

`unit` - The unit to align the maximum value, can be `null` .

Returns:

The maximum value according to the specified unit or `null` if no maximum value is supported.

Throws:

`IllegalArgumentException` - If the unit is not supported.

Interface Units

org.osgi.service.dal

```
public interface Units
```

Contains the most of the International System of Units unit symbols. The constant name represents the unit name. The constant value represents the unit symbol as it's defined in `PropertyMetadata.META_INFO_UNITS`.

Field Summary		Page
`String`	**AMPERE** Unit of electric current defined by the International System of Units (SI).	75
`String`	**AMPERE_PER_METRE** Unit of magnetic field strength.	77
`String`	**AMPERE_PER_SQUARE_METRE** Unit of current density.	76
`String`	**ANGSTROM** Unit of length.	85
`String`	**BAR** Unit of pressure.	85
`String`	**BARN** Unit of area.	85
`String`	**BECQUEREL** Unit of activity referred to a radionuclide.	80
`String`	**BEL** Unit of logarithmic ratio quantities.	86
`String`	**CANDELA** Unit of luminous intensity defined by the International System of Units (SI).	75
`String`	**CANDELA_PER_SQUARE_METRE** Unit of luminance.	77
`String`	**COULOMB** Unit of electronic charge, amount of electricity.	78
`String`	**COULOMB_PER_CUBIC_METRE** Unit of electric charge density.	82
`String`	**COULOMB_PER_KILOGRAM** Unit of exposure (x- and gamma-rays).	83
`String`	**COULOMB_PER_SQUARE_METRE** Unit of surface charge density, electric flux density, electric displacement.	82
`String`	**CUBIC_METRE** Unit of volume.	75
`String`	**CUBIC_METRE_PER_KILOGRAM** Unit of specific volume.	76
`String`	**DAY** Unit of time.	84
`String`	**DECIBEL** Unit of logarithmic ratio quantities.	86
`String`	**DEGREE** Unit of plane angle.	84

String	DEGREE_CELSIUS Unit of Celsius temperature.	79
String	DYNE Unit of force.	86
String	ERG Unit of energy.	86
String	FARAD Unit of capacitance.	78
String	FARAD_PER_METRE Unit of permittivity.	82
String	GAL Unit of acceleration.	87
String	GAUSS Unit of magnetic flux density.	87
String	GRAY Unit of absorbed dose, specific energy (imparted), kerma.	80
String	GRAY_PER_SECOND Unit of absorbed dose rate.	83
String	HECTARE Unit of area.	84
String	HENRY Unit of inductance.	79
String	HENRY_PER_METRE Unit of permeability.	82
String	HERTZ Unit of frequency.	77
String	HOUR Unit of time.	84
String	JOULE Unit of energy, work, amount of electricity.	78
String	JOULE_PER_CUBIC_METRE Unit of energy density.	82
String	JOULE_PER_KELVIN Unit of heat capacity, entropy.	81
String	JOULE_PER_KILOGRAM Unit of specific energy.	81
String	JOULE_PER_KILOGRAM_KELVIN Unit of specific heat capacity, specific entropy.	81
String	JOULE_PER_MOLE Unit of molar energy.	82
String	JOULE_PER_MOLE_KELVIN Unit of molar entropy, molar heat capacity.	83
String	KATAL Unit of catalytic activity.	80
String	KATAL_PER_CUBIC_METRE Unit of catalytic activity concentration.	83
String	KELVIN Unit of thermodynamic temperature defined by the International System of Units (SI).	75
String	KILOGRAM Unit of mass defined by the International System of Units (SI).	74

String	**KILOGRAM_PER_CUBIC_METRE** Unit of density, mass density, mass concentration.	76
String	**KILOGRAM_PER_SQUARE_METRE** Unit of surface density.	76
String	**KNOT** Unit of speed.	85
String	**LITRE** Unit of volume.	84
String	**LUMEN** Unit of luminous flux.	79
String	**LUX** Unit of illuminance.	79
String	**MAXWELL** Unit of magnetic flux.	87
String	**METRE** Unit of length defined by the International System of Units (SI).	74
String	**METRE_PER_SECOND** Unit of speed, velocity.	76
String	**METRE_PER_SECOND_SQUARED** Unit of acceleration.	76
String	**MILLIMETRE_OF_MERCURY** Unit of pressure.	85
String	**MOLE** Unit of amount of substance defined by the International System of Units (SI).	75
String	**MOLE_PER_CUBIC_METRE** Unit of amount concentration, concentration.	77
String	**NAUTICAL_MILE** Unit of distance.	85
String	**NEPER** Unit of logarithmic ratio quantities.	85
String	**NEWTON** Unit of force.	77
String	**NEWTON_METRE** Unit of moment of force.	80
String	**NEWTON_PER_METRE** Unit of surface tension.	80
String	**OERSTED** Unit of magnetic field.	87
String	**OHM** Unit of electric resistance.	78
String	**PASCAL** Unit of pressure, stress.	78
String	**PASCAL_SECOND** Unit of dynamic viscosity.	80
String	**PHOT** Unit of illuminance.	86
String	**PLANE_ANGLE_MINUTE** Unit of plane angle.	84
String	**PLANE_ANGLE_SECOND** Unit of plane angle.	84

String	POISE Unit of dynamic viscosity.	86
String	PREFIX_ATTO Adopted prefix symbol to form the symbols of the decimal submultiples of SI units.	89
String	PREFIX_CENTI Adopted prefix symbol to form the symbols of the decimal submultiples of SI units.	88
String	PREFIX_DECA Adopted prefix symbol to form the symbols of the decimal multiples of SI units.	87
String	PREFIX_DECI Adopted prefix symbol to form the symbols of the decimal submultiples of SI units.	88
String	PREFIX_EXA Adopted prefix symbol to form the symbols of the decimal multiples of SI units.	88
String	PREFIX_FEMTO Adopted prefix symbol to form the symbols of the decimal submultiples of SI units.	89
String	PREFIX_GIGA Adopted prefix symbol to form the symbols of the decimal multiples of SI units.	88
String	PREFIX_HECTO Adopted prefix symbol to form the symbols of the decimal multiples of SI units.	87
String	PREFIX_KILO Adopted prefix symbol to form the symbols of the decimal multiples of SI units.	87
String	PREFIX_MEGA Adopted prefix symbol to form the symbols of the decimal multiples of SI units.	88
String	PREFIX_MICRO Adopted prefix symbol to form the symbols of the decimal submultiples of SI units.	89
String	PREFIX_MILLI Adopted prefix symbol to form the symbols of the decimal submultiples of SI units.	89
String	PREFIX_NANO Adopted prefix symbol to form the symbols of the decimal submultiples of SI units.	89
String	PREFIX_PICO Adopted prefix symbol to form the symbols of the decimal submultiples of SI units.	89
String	PREFIX_YOCTO Adopted prefix symbol to form the symbols of the decimal submultiples of SI units.	90
String	PREFIX_YOTTA Adopted prefix symbol to form the symbols of the decimal multiples of SI units.	88
String	PREFIX_ZEPTO Adopted prefix symbol to form the symbols of the decimal submultiples of SI units.	89
String	PREFIX_ZETTA Adopted prefix symbol to form the symbols of the decimal multiples of SI units.	88
String	RADIAN Unit of plane angle.	77
String	RADIAN_PER_SECOND Unit of angular velocity.	81
String	RADIAN_PER_SECOND_SQUARED Unit of angular acceleration.	81
String	RECIPROCAL_METRE Unit of wavenumber.	76
String	SECOND Unit of time defined by the International System of Units (SI).	75
String	SIEMENS Unit of electric conductance.	79

String	**SIEVERT** Unit of dose equivalent, ambient dose equivalent, directional dose equivalent, personal dose equivalent.	80
String	**SQUARE_METRE** Unit of area.	75
String	**STERADIAN** Unit of solid angle.	77
String	**STILB** Unit of luminance.	86
String	**STOKES** Unit of kinematic viscosity.	86
String	**TESLA** Unit of magnetic flux density.	79
String	**TIME_MINUTE** Unit of time.	83
String	**TONNE** Unit of mass.	85
String	**VOLT** Unit of electric potential difference, electromotive force.	78
String	**VOLT_PER_METRE** Unit of electric field strength.	82
String	**WATT** Unit of power, radiant flux.	78
String	**WATT_PER_METRE_KELVIN** Unit of thermal conductivity.	81
String	**WATT_PER_SQUARE_METRE** Unit of heat flux density, irradiance.	81
String	**WATT_PER_SQUARE_METRE_STERADIAN** Unit of radiance.	83
String	**WATT_PER_STERADIAN** Unit of radiant intensity.	83
String	**WEBER** Unit of magnetic flux.	79

Field Detail

METRE

```
public static final String METRE = "m"
```

Unit of length defined by the International System of Units (SI). It's one of be base units called metre.

KILOGRAM

```
public static final String KILOGRAM = "kg"
```

Unit of mass defined by the International System of Units (SI). It's one of be base units called kilogram.

SECOND

```
public static final String SECOND = "s"
```

Unit of time defined by the International System of Units (SI). It's one of be base units called second.

AMPERE

```
public static final String AMPERE = "A"
```

Unit of electric current defined by the International System of Units (SI). It's one of be base units called ampere.

KELVIN

```
public static final String KELVIN = "\u212a"
```

Unit of thermodynamic temperature defined by the International System of Units (SI). It's one of be base units called kelvin.

MOLE

```
public static final String MOLE = "mol"
```

Unit of amount of substance defined by the International System of Units (SI). It's one of be base units called mole.

CANDELA

```
public static final String CANDELA = "cd"
```

Unit of luminous intensity defined by the International System of Units (SI). It's one of be base units called candela.

SQUARE_METRE

```
public static final String SQUARE_METRE = "m\u00b2"
```

Unit of area. It's one of coherent derived units in the SI expressed in terms of base units. The unit is called square metre.

CUBIC_METRE

```
public static final String CUBIC_METRE = "m\u00b3"
```

Unit of volume. It's one of coherent derived units in the SI expressed in terms of base units. The unit is called cubic metre.

METRE_PER_SECOND

```
public static final String METRE_PER_SECOND = "m/s"
```

Unit of speed, velocity. It's one of coherent derived units in the SI expressed in terms of base units. The unit is called metre per second.

METRE_PER_SECOND_SQUARED

```
public static final String METRE_PER_SECOND_SQUARED = "m/s\u00b2"
```

Unit of acceleration. It's one of coherent derived units in the SI expressed in terms of base units. The unit is called metre per second squared.

RECIPROCAL_METRE

```
public static final String RECIPROCAL_METRE = "m\u207b\u00b9"
```

Unit of wavenumber. It's one of coherent derived units in the SI expressed in terms of base units. The unit is called reciprocal metre.

KILOGRAM_PER_CUBIC_METRE

```
public static final String KILOGRAM_PER_CUBIC_METRE = "kg/m\u00b3"
```

Unit of density, mass density, mass concentration. It's one of coherent derived units in the SI expressed in terms of base units. The unit is called kilogram per cubic metre.

KILOGRAM_PER_SQUARE_METRE

```
public static final String KILOGRAM_PER_SQUARE_METRE = "kg/m\u00b2"
```

Unit of surface density. It's one of coherent derived units in the SI expressed in terms of base units. The unit is called kilogram per square metre.

CUBIC_METRE_PER_KILOGRAM

```
public static final String CUBIC_METRE_PER_KILOGRAM = "m\u00b3/kg"
```

Unit of specific volume. It's one of coherent derived units in the SI expressed in terms of base units. The unit is called cubic metre per kilogram.

AMPERE_PER_SQUARE_METRE

```
public static final String AMPERE_PER_SQUARE_METRE = "A/m\u00b2"
```

Unit of current density. It's one of coherent derived units in the SI expressed in terms of base units. The unit is called ampere per square metre.

AMPERE_PER_METRE

```
public static final String AMPERE_PER_METRE = "A/m"
```

Unit of magnetic field strength. It's one of coherent derived units in the SI expressed in terms of base units. The unit is called ampere per metre.

MOLE_PER_CUBIC_METRE

```
public static final String MOLE_PER_CUBIC_METRE = "mol/m\u00b3"
```

Unit of amount concentration, concentration. It's one of coherent derived units in the SI expressed in terms of base units. The unit is called mole per cubic metre.

CANDELA_PER_SQUARE_METRE

```
public static final String CANDELA_PER_SQUARE_METRE = "cd/m\u00b2"
```

Unit of luminance. It's one of coherent derived units in the SI expressed in terms of base units. The unit is called candela per square metre.

RADIAN

```
public static final String RADIAN = "rad"
```

Unit of plane angle. It's one of the coherent derived units in the SI with special names and symbols. The unit is called radian.

STERADIAN

```
public static final String STERADIAN = "sr"
```

Unit of solid angle. It's one of the coherent derived units in the SI with special names and symbols. The unit is called steradian.

HERTZ

```
public static final String HERTZ = "Hz"
```

Unit of frequency. It's one of the coherent derived units in the SI with special names and symbols. The unit is called hertz.

NEWTON

```
public static final String NEWTON = "N"
```

Unit of force. It's one of the coherent derived units in the SI with special names and symbols. The unit is called newton.

PASCAL

```
public static final String PASCAL = "Pa"
```

Unit of pressure, stress. It's one of the coherent derived units in the SI with special names and symbols. The unit is called pascal.

JOULE

```
public static final String JOULE = "J"
```

Unit of energy, work, amount of electricity. It's one of the coherent derived units in the SI with special names and symbols. The unit is called joule.

WATT

```
public static final String WATT = "W"
```

Unit of power, radiant flux. It's one of the coherent derived units in the SI with special names and symbols. The unit is called watt.

COULOMB

```
public static final String COULOMB = "C"
```

Unit of electronic charge, amount of electricity. It's one of the coherent derived units in the SI with special names and symbols. The unit is called coulomb.

VOLT

```
public static final String VOLT = "V"
```

Unit of electric potential difference, electromotive force. It's one of the coherent derived units in the SI with special names and symbols. The unit is called volt.

FARAD

```
public static final String FARAD = "F"
```

Unit of capacitance. It's one of the coherent derived units in the SI with special names and symbols. The unit is called farad.

OHM

```
public static final String OHM = "\u2126"
```

Unit of electric resistance. It's one of the coherent derived units in the SI with special names and symbols. The unit is called ohm.

SIEMENS

```
public static final String SIEMENS = "S"
```

Unit of electric conductance. It's one of the coherent derived units in the SI with special names and symbols. The unit is called siemens.

WEBER

```
public static final String WEBER = "Wb"
```

Unit of magnetic flux. It's one of the coherent derived units in the SI with special names and symbols. The unit is called weber.

TESLA

```
public static final String TESLA = "T"
```

Unit of magnetic flux density. It's one of the coherent derived units in the SI with special names and symbols. The unit is called tesla.

HENRY

```
public static final String HENRY = "H"
```

Unit of inductance. It's one of the coherent derived units in the SI with special names and symbols. The unit is called henry.

DEGREE_CELSIUS

```
public static final String DEGREE_CELSIUS = "\u2103"
```

Unit of Celsius temperature. It's one of the coherent derived units in the SI with special names and symbols. The unit is called degree Celsius.

LUMEN

```
public static final String LUMEN = "lm"
```

Unit of luminous flux. It's one of the coherent derived units in the SI with special names and symbols. The unit is called lumen.

LUX

```
public static final String LUX = "lx"
```

Unit of illuminance. It's one of the coherent derived units in the SI with special names and symbols. The unit is called lux.

BECQUEREL

```
public static final String BECQUEREL = "Bq"
```

Unit of activity referred to a radionuclide. It's one of the coherent derived units in the SI with special names and symbols. The unit is called becquerel.

GRAY

```
public static final String GRAY = "Gy"
```

Unit of absorbed dose, specific energy (imparted), kerma. It's one of the coherent derived units in the SI with special names and symbols. The unit is called gray.

SIEVERT

```
public static final String SIEVERT = "Sv"
```

Unit of dose equivalent, ambient dose equivalent, directional dose equivalent, personal dose equivalent. It's one of the coherent derived units in the SI with special names and symbols. The unit is called sievert.

KATAL

```
public static final String KATAL = "kat"
```

Unit of catalytic activity. It's one of the coherent derived units in the SI with special names and symbols. The unit is called katal.

PASCAL_SECOND

```
public static final String PASCAL_SECOND = "Pa s"
```

Unit of dynamic viscosity. It's one of coherent derived units whose names and symbols include SI coherent derived units with special names and symbols. The unit is called pascal second.

NEWTON_METRE

```
public static final String NEWTON_METRE = "N m"
```

Unit of moment of force. It's one of coherent derived units whose names and symbols include SI coherent derived units with special names and symbols. The unit is called newton metre.

NEWTON_PER_METRE

```
public static final String NEWTON_PER_METRE = "N/m"
```

Unit of surface tension. It's one of coherent derived units whose names and symbols include SI coherent derived units with special names and symbols. The unit is called newton per metre.

RADIAN_PER_SECOND

```
public static final String RADIAN_PER_SECOND = "rad/s"
```

Unit of angular velocity. It's one of coherent derived units whose names and symbols include SI coherent derived units with special names and symbols. The unit is called radian per second.

RADIAN_PER_SECOND_SQUARED

```
public static final String RADIAN_PER_SECOND_SQUARED = "rad/s\u00b2"
```

Unit of angular acceleration. It's one of coherent derived units whose names and symbols include SI coherent derived units with special names and symbols. The unit is called radian per second squared.

WATT_PER_SQUARE_METRE

```
public static final String WATT_PER_SQUARE_METRE = "W/m\u00b2"
```

Unit of heat flux density, irradiance. It's one of coherent derived units whose names and symbols include SI coherent derived units with special names and symbols. The unit is called watt per square metre.

JOULE_PER_KELVIN

```
public static final String JOULE_PER_KELVIN = "J/K"
```

Unit of heat capacity, entropy. It's one of coherent derived units whose names and symbols include SI coherent derived units with special names and symbols. The unit is called joule per kelvin.

JOULE_PER_KILOGRAM_KELVIN

```
public static final String JOULE_PER_KILOGRAM_KELVIN = "J/(kg K)"
```

Unit of specific heat capacity, specific entropy. It's one of coherent derived units whose names and symbols include SI coherent derived units with special names and symbols. The unit is called joule per kilogram kelvin.

JOULE_PER_KILOGRAM

```
public static final String JOULE_PER_KILOGRAM = "J/kg"
```

Unit of specific energy. It's one of coherent derived units whose names and symbols include SI coherent derived units with special names and symbols. The unit is called joule per kilogram.

WATT_PER_METRE_KELVIN

```
public static final String WATT_PER_METRE_KELVIN = "W/(m K)"
```

Unit of thermal conductivity. It's one of coherent derived units whose names and symbols include SI coherent derived units with special names and symbols. The unit is called watt per metre kelvin.

JOULE_PER_CUBIC_METRE

```
public static final String JOULE_PER_CUBIC_METRE = "J/m\u00b3"
```

Unit of energy density. It's one of coherent derived units whose names and symbols include SI coherent derived units with special names and symbols. The unit is called joule per cubic metre.

VOLT_PER_METRE

```
public static final String VOLT_PER_METRE = "V/m"
```

Unit of electric field strength. It's one of coherent derived units whose names and symbols include SI coherent derived units with special names and symbols. The unit is called volt per metre.

COULOMB_PER_CUBIC_METRE

```
public static final String COULOMB_PER_CUBIC_METRE = "C/m\u00b3"
```

Unit of electric charge density. It's one of coherent derived units whose names and symbols include SI coherent derived units with special names and symbols. The unit is called coulomb per cubic metre.

COULOMB_PER_SQUARE_METRE

```
public static final String COULOMB_PER_SQUARE_METRE = "C/m\u00b2"
```

Unit of surface charge density, electric flux density, electric displacement. It's one of coherent derived units whose names and symbols include SI coherent derived units with special names and symbols. The unit is called coulomb per square metre.

FARAD_PER_METRE

```
public static final String FARAD_PER_METRE = "F/m"
```

Unit of permittivity. It's one of coherent derived units whose names and symbols include SI coherent derived units with special names and symbols. The unit is called farad per metre.

HENRY_PER_METRE

```
public static final String HENRY_PER_METRE = "H/m"
```

Unit of permeability. It's one of coherent derived units whose names and symbols include SI coherent derived units with special names and symbols. The unit is called henry per metre.

JOULE_PER_MOLE

```
public static final String JOULE_PER_MOLE = "J/mol"
```

Unit of molar energy. It's one of coherent derived units whose names and symbols include SI coherent derived units with special names and symbols. The unit is called joule per mole.

JOULE_PER_MOLE_KELVIN

```
public static final String JOULE_PER_MOLE_KELVIN = "J/(mol K)"
```

Unit of molar entropy, molar heat capacity. It's one of coherent derived units whose names and symbols include SI coherent derived units with special names and symbols. The unit is called joule per mole kelvin.

COULOMB_PER_KILOGRAM

```
public static final String COULOMB_PER_KILOGRAM = "C/kg"
```

Unit of exposure (x- and gamma-rays). It's one of coherent derived units whose names and symbols include SI coherent derived units with special names and symbols. The unit is called coulomb per kilogram.

GRAY_PER_SECOND

```
public static final String GRAY_PER_SECOND = "Gy/s"
```

Unit of absorbed dose rate. It's one of coherent derived units whose names and symbols include SI coherent derived units with special names and symbols. The unit is called gray per second.

WATT_PER_STERADIAN

```
public static final String WATT_PER_STERADIAN = "W/sr"
```

Unit of radiant intensity. It's one of coherent derived units whose names and symbols include SI coherent derived units with special names and symbols. The unit is called watt per steradian.

WATT_PER_SQUARE_METRE_STERADIAN

```
public static final String WATT_PER_SQUARE_METRE_STERADIAN = "W/(m\u00b2 sr)"
```

Unit of radiance. It's one of coherent derived units whose names and symbols include SI coherent derived units with special names and symbols. The unit is called watt per square metre steradian.

KATAL_PER_CUBIC_METRE

```
public static final String KATAL_PER_CUBIC_METRE = "kat/m\u00b3"
```

Unit of catalytic activity concentration. It's one of coherent derived units whose names and symbols include SI coherent derived units with special names and symbols. The unit is called katal per cubic metre.

TIME_MINUTE

```
public static final String TIME_MINUTE = "min"
```

Unit of time. It's one of non-SI units accepted for use with the International System of Units. The unit is called minute.

HOUR

```
public static final String HOUR = "h"
```

Unit of time. It's one of non-SI units accepted for use with the International System of Units. The unit is called hour.

DAY

```
public static final String DAY = "d"
```

Unit of time. It's one of non-SI units accepted for use with the International System of Units. The unit is called day.

DEGREE

```
public static final String DEGREE = "\u00b0"
```

Unit of plane angle. It's one of non-SI units accepted for use with the International System of Units. The unit is called degree.

PLANE_ANGLE_MINUTE

```
public static final String PLANE_ANGLE_MINUTE = "\u2032"
```

Unit of plane angle. It's one of non-SI units accepted for use with the International System of Units. The unit is called minute.

PLANE_ANGLE_SECOND

```
public static final String PLANE_ANGLE_SECOND = "\u2033"
```

Unit of plane angle. It's one of non-SI units accepted for use with the International System of Units. The unit is called second.

HECTARE

```
public static final String HECTARE = "ha"
```

Unit of area. It's one of non-SI units accepted for use with the International System of Units. The unit is called hectare.

LITRE

```
public static final String LITRE = "l"
```

Unit of volume. It's one of non-SI units accepted for use with the International System of Units. The unit is called litre. International System of Units accepts two symbols: lower-case l and capital L. That constant value is using the lower-case l.

TONNE

```
public static final String TONNE = "t"
```

Unit of mass. It's one of non-SI units accepted for use with the International System of Units. The unit is called tonne.

BAR

```
public static final String BAR = "bar"
```

Unit of pressure. It's one of other non-SI units. The unit is called bar.

MILLIMETRE_OF_MERCURY

```
public static final String MILLIMETRE_OF_MERCURY = "mmHg"
```

Unit of pressure. It's one of other non-SI units. The unit is called millimetre of mercury.

ANGSTROM

```
public static final String ANGSTROM = "\u212b"
```

Unit of length. It's one of other non-SI units. The unit is called angstrom.

NAUTICAL_MILE

```
public static final String NAUTICAL_MILE = "M"
```

Unit of distance. It's one of other non-SI units. The unit is called nautical mile.

BARN

```
public static final String BARN = "b"
```

Unit of area. It's one of other non-SI units. The unit is called barn.

KNOT

```
public static final String KNOT = "kn"
```

Unit of speed. It's one of other non-SI units. The unit is called knot.

NEPER

```
public static final String NEPER = "Np"
```

Unit of logarithmic ratio quantities. It's one of other non-SI units. The unit is called neper.

BEL

```
public static final String BEL = "B"
```

Unit of logarithmic ratio quantities. It's one of other non-SI units. The unit is called bel.

DECIBEL

```
public static final String DECIBEL = "dB"
```

Unit of logarithmic ratio quantities. It's one of other non-SI units. The unit is called decibel.

ERG

```
public static final String ERG = "erg"
```

Unit of energy. It's one of non-SI units associated with the CGS and the CGS-Gaussian system of units. The unit is called erg.

DYNE

```
public static final String DYNE = "dyn"
```

Unit of force. It's one of non-SI units associated with the CGS and the CGS-Gaussian system of units. The unit is called dyne.

POISE

```
public static final String POISE = "P"
```

Unit of dynamic viscosity. It's one of non-SI units associated with the CGS and the CGS-Gaussian system of units. The unit is called poise.

STOKES

```
public static final String STOKES = "St"
```

Unit of kinematic viscosity. It's one of non-SI units associated with the CGS and the CGS-Gaussian system of units. The unit is called stokes.

STILB

```
public static final String STILB = "sb"
```

Unit of luminance. It's one of non-SI units associated with the CGS and the CGS-Gaussian system of units. The unit is called stilb.

PHOT

```
public static final String PHOT = "ph"
```

Unit of illuminance. It's one of non-SI units associated with the CGS and the CGS-Gaussian system of units. The unit is called phot.

GAL

```
public static final String GAL = "Gal"
```

Unit of acceleration. It's one of non-SI units associated with the CGS and the CGS-Gaussian system of units. The unit is called gal.

MAXWELL

```
public static final String MAXWELL = "Mx"
```

Unit of magnetic flux. It's one of non-SI units associated with the CGS and the CGS-Gaussian system of units. The unit is called maxwell.

GAUSS

```
public static final String GAUSS = "G"
```

Unit of magnetic flux density. It's one of non-SI units associated with the CGS and the CGS-Gaussian system of units. The unit is called gauss.

OERSTED

```
public static final String OERSTED = "Oe"
```

Unit of magnetic field. It's one of non-SI units associated with the CGS and the CGS-Gaussian system of units. The unit is called oersted.

PREFIX_DECA

```
public static final String PREFIX_DECA = "da"
```

Adopted prefix symbol to form the symbols of the decimal multiples of SI units. It's called deca and represents the 1st power of ten.

PREFIX_HECTO

```
public static final String PREFIX_HECTO = "h"
```

Adopted prefix symbol to form the symbols of the decimal multiples of SI units. It's called hecto and represents the 2nd power of ten.

PREFIX_KILO

```
public static final String PREFIX_KILO = "k"
```

Adopted prefix symbol to form the symbols of the decimal multiples of SI units. It's called kilo and represents the 3rd power of ten.

PREFIX_MEGA

```
public static final String PREFIX_MEGA = "M"
```

Adopted prefix symbol to form the symbols of the decimal multiples of SI units. It's called mega and represents the 6th power of ten.

PREFIX_GIGA

```
public static final String PREFIX_GIGA = "G"
```

Adopted prefix symbol to form the symbols of the decimal multiples of SI units. It's called giga and represents the 9th power of ten.

PREFIX_EXA

```
public static final String PREFIX_EXA = "E"
```

Adopted prefix symbol to form the symbols of the decimal multiples of SI units. It's called exa and represents the 18th power of ten.

PREFIX_ZETTA

```
public static final String PREFIX_ZETTA = "Z"
```

Adopted prefix symbol to form the symbols of the decimal multiples of SI units. It's called zetta and represents the 21th power of ten.

PREFIX_YOTTA

```
public static final String PREFIX_YOTTA = "Y"
```

Adopted prefix symbol to form the symbols of the decimal multiples of SI units. It's called yotta and represents the 24th power of ten.

PREFIX_DECI

```
public static final String PREFIX_DECI = "d"
```

Adopted prefix symbol to form the symbols of the decimal submultiples of SI units. It's called deci and represents the 1st negative power of ten.

PREFIX_CENTI

```
public static final String PREFIX_CENTI = "c"
```

Adopted prefix symbol to form the symbols of the decimal submultiples of SI units. It's called centi and represents the 2nd negative power of ten.

PREFIX_MILLI

```
public static final String PREFIX_MILLI = "m"
```

Adopted prefix symbol to form the symbols of the decimal submultiples of SI units. It's called milli and represents the 3rd negative power of ten.

PREFIX_MICRO

```
public static final String PREFIX_MICRO = "\u00b5"
```

Adopted prefix symbol to form the symbols of the decimal submultiples of SI units. It's called micro and represents the 6th negative power of ten.

PREFIX_NANO

```
public static final String PREFIX_NANO = "n"
```

Adopted prefix symbol to form the symbols of the decimal submultiples of SI units. It's called nano and represents the 9th negative power of ten.

PREFIX_PICO

```
public static final String PREFIX_PICO = "p"
```

Adopted prefix symbol to form the symbols of the decimal submultiples of SI units. It's called pico and represents the 12th negative power of ten.

PREFIX_FEMTO

```
public static final String PREFIX_FEMTO = "f"
```

Adopted prefix symbol to form the symbols of the decimal submultiples of SI units. It's called femto and represents the 15th negative power of ten.

PREFIX_ATTO

```
public static final String PREFIX_ATTO = "a"
```

Adopted prefix symbol to form the symbols of the decimal submultiples of SI units. It's called atto and represents the 18th negative power of ten.

PREFIX_ZEPTO

```
public static final String PREFIX_ZEPTO = "z"
```

Adopted prefix symbol to form the symbols of the decimal submultiples of SI units. It's called zepto and represents the 21th negative power of ten.

PREFIX_YOCTO

```
public static final String PREFIX_YOCTO = "y"
```

Adopted prefix symbol to form the symbols of the decimal submultiples of SI units. It's called yocto and represents the 24th negative power of ten.

Package org.osgi.service.dal.functions

Device Functions 1.0.

See:
Description

Interface Summary		*Page*
Alarm	`Alarm` Device Function provides alarm sensor support.	*92*
BooleanControl	`BooleanControl` Device Function provides a boolean control support.	*93*
BooleanSensor	`BooleanSensor` Device Function provides boolean sensor monitoring.	*97*
Keypad	`Keypad` Device Function provides support for keypad control.	*99*
Meter	`Meter` Device Function can measure metering information.	*100*
MultiLevelControl	`MultiLevelControl` Device Function provides multi-level control support.	*103*
MultiLevelSensor	`MultiLevelSensor` Device Function provides multi-level sensor monitoring.	*106*
Types	Shares common constants for all device functions defined in this package.	*108*

Package org.osgi.service.dal.functions Description

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.dal.functions; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

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

Interface Alarm

org.osgi.service.dal.functions

All Superinterfaces:

DeviceFunction

```
public interface Alarm
extends DeviceFunction
```

`Alarm` Device Function provides alarm sensor support. There is only one eventable property and no operations.

See Also:

`AlarmData`

Field Summary		*Page*
`String`	**`PROPERTY_ALARM`** Specifies the alarm property name.	*92*

Fields inherited from interface org.osgi.service.dal.DeviceFunction
`SERVICE_DESCRIPTION, SERVICE_DEVICE_UID, SERVICE_OPERATION_NAMES, SERVICE_PROPERTY_NAMES, SERVICE_REFERENCE_UIDS, SERVICE_TYPE, SERVICE_UID, SERVICE_VERSION`

Methods inherited from interface org.osgi.service.dal.DeviceFunction
`getOperationMetadata, getProperty, getPropertyMetadata`

Field Detail

PROPERTY_ALARM

```
public static final String PROPERTY_ALARM = "alarm"
```

Specifies the alarm property name. The property is eventable.

See Also:

`AlarmData`

Interface BooleanControl

org.osgi.service.dal.functions

All Superinterfaces:

DeviceFunction

```
public interface BooleanControl
extends DeviceFunction
```

`BooleanControl` Device Function provides a boolean control support. The function state is accessible with `getData()` getter and `setData(boolean)` setter. The state can be reversed with `reverse()` method, can be set to `true` value with `setTrue()` method and can be set to `false` value with `setFalse()` method.

As an example, the function is easily mappable to ZigBee OnOff cluster and Z-Wave Binary Switch command class. The control type can be:

- `Types.TYPE_LIGHT`
- `Types.TYPE_DOOR`
- `Types.TYPE_WINDOW`
- `Types.TYPE_POWER`
- custom - vendor specific type

See Also:

`BooleanData`

Field Summary		Page
`String`	**`OPERATION_REVERSE`** Specifies the reverse operation name.	*94*
`String`	**`OPERATION_SET_FALSE`** Specifies the operation name, which sets the control state to `false` value.	*94*
`String`	**`OPERATION_SET_TRUE`** Specifies the operation name, which sets the control state to `true` value.	*94*
`String`	**`PROPERTY_DATA`** Specifies the state property name.	*94*

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

`SERVICE_DESCRIPTION, SERVICE_DEVICE_UID, SERVICE_OPERATION_NAMES, SERVICE_PROPERTY_NAMES, SERVICE_REFERENCE_UIDS, SERVICE_TYPE, SERVICE_UID, SERVICE_VERSION`

Method Summary		Page
`BooleanData`	**`getData`**`()` Returns the current state of `BooleanControl`.	*94*
`void`	**`reverse`**`()` Reverses the `BooleanControl` state.	*95*
`void`	**`setData`**`(boolean data)` Sets the `BooleanControl` state to the specified value.	*95*
`void`	**`setFalse`**`()` Sets the `BooleanControl` state to `false` value.	*96*
`void`	**`setTrue`**`()` Sets the `BooleanControl` state to `true` value.	*95*

Methods inherited from interface org.osgi.service.dal.DeviceFunction
`getOperationMetadata, getProperty, getPropertyMetadata`

Field Detail

OPERATION_REVERSE

```
public static final String OPERATION_REVERSE = "reverse"
```

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

OPERATION_SET_TRUE

```
public static final String OPERATION_SET_TRUE = "setTrue"
```

Specifies the operation name, which sets the control state to `true` value. The operation can be executed with `setTrue()` method.

OPERATION_SET_FALSE

```
public static final String OPERATION_SET_FALSE = "setFalse"
```

Specifies the operation name, which sets the control state to `false` value. The operation can be executed with `setFalse()` method.

PROPERTY_DATA

```
public static final String PROPERTY_DATA = "data"
```

Specifies the state property name. The property value is accessible with `getData()` method.

See Also:

`BooleanData`

Method Detail

getData

```
BooleanData getData()
             throws UnsupportedOperationException,
                    IllegalStateException,
                    DeviceException
```

Returns the current state of `BooleanControl`. It's a getter method for `PROPERTY_DATA` property.

Returns:

The current state of `BooleanControl`.

Throws:

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

See Also:
BooleanData, PROPERTY_DATA

setData

```
void setData(boolean data)
      throws UnsupportedOperationException,
             IllegalStateException,
             DeviceException
```

Sets the `BooleanControl` state to the specified value. It's setter method for `PROPERTY_DATA` property.

Parameters:
`data` - The new function value.

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

See Also:
PROPERTY_DATA

reverse

```
void reverse()
      throws UnsupportedOperationException,
             IllegalStateException,
             DeviceException
```

Reverses the `BooleanControl` state. If the current state represents `true` value, it'll be reversed to `false`. If the current state represents `false` value, it'll be reversed to `true`. The operation name is `OPERATION_REVERSE`.

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

setTrue

```
void setTrue()
      throws UnsupportedOperationException,
             IllegalStateException,
             DeviceException
```

Sets the `BooleanControl` state to `true` value. The operation name is `OPERATION_SET_TRUE`.

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

setFalse

```
void setFalse()
       throws UnsupportedOperationException,
              IllegalStateException,
              DeviceException
```

Sets the `BooleanControl` state to `false` value. The operation name is `OPERATION_SET_FALSE`.

Throws:

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

Interface BooleanSensor

org.osgi.service.dal.functions

All Superinterfaces:

DeviceFunction

```
public interface BooleanSensor
extends DeviceFunction
```

`BooleanSensor` Device Function provides boolean sensor monitoring. It reports its state when an important event is available. The state is accessible with `getData()` getter. There are no operations.

As an example, the function is easily mappable to ZigBee Occupancy Sensing cluster and Z-Wave Binary Sensor command class. The sensor type can be:

- `Types.TYPE_LIGHT`
- `Types.TYPE_GAS`
- `Types.TYPE_SMOKE`
- `Types.TYPE_DOOR`
- `Types.TYPE_WINDOW`
- `Types.TYPE_POWER`
- `Types.TYPE_RAIN`
- `Types.TYPE_CONTACT`
- `Types.TYPE_FIRE`
- `Types.TYPE_OCCUPANCY`
- `Types.TYPE_WATER`
- `Types.TYPE_MOTION`
- custom - vendor specific type

See Also:

`BooleanData`

Field Summary		***Page***
`String`	**`PROPERTY_DATA`** Specifies the state property name.	*98*

Fields inherited from interface org.osgi.service.dal.DeviceFunction
`SERVICE_DESCRIPTION`, `SERVICE_DEVICE_UID`, `SERVICE_OPERATION_NAMES`, `SERVICE_PROPERTY_NAMES`, `SERVICE_REFERENCE_UIDS`, `SERVICE_TYPE`, `SERVICE_UID`, `SERVICE_VERSION`

Method Summary		***Page***
`BooleanData`	**`getData`**`()` Returns the `BooleanSensor`current state.	*98*

Methods inherited from interface org.osgi.service.dal.DeviceFunction
`getOperationMetadata`, `getProperty`, `getPropertyMetadata`

Field Detail

PROPERTY_DATA

```
public static final String PROPERTY_DATA = "data"
```

Specifies the state property name. The property value is accessible with `getData()` getter.

Method Detail

getData

```
BooleanData getData()
            throws UnsupportedOperationException,
                   IllegalStateException,
                   DeviceException
```

Returns the `BooleanSensor`current state. It's a getter method for `PROPERTY_DATA` property.

Returns:

The `BooleanSensor` current state.

Throws:

`UnsupportedOperationException` - If the operation is not supported.

`IllegalStateException` - If this device function service object has already been unregistered.

`DeviceException` - If an operation error is available.

See Also:

`BooleanData`

Interface Keypad

org.osgi.service.dal.functions

All Superinterfaces:

DeviceFunction

```
public interface Keypad
extends DeviceFunction
```

`Keypad` Device Function provides support for keypad control. A keypad typically consists of one or more keys/buttons, which can be discerned. Different types of key presses like short and long press can typically also be detected. There is only one eventable property and no operations.

See Also:

`KeypadData`

Field Summary		Page
`String`	**`PROPERTY_KEY`** Specifies a property name for a key from the keypad.	*99*

Fields inherited from interface org.osgi.service.dal.DeviceFunction
`SERVICE_DESCRIPTION`, `SERVICE_DEVICE_UID`, `SERVICE_OPERATION_NAMES`, `SERVICE_PROPERTY_NAMES`, `SERVICE_REFERENCE_UIDS`, `SERVICE_TYPE`, `SERVICE_UID`, `SERVICE_VERSION`

Methods inherited from interface org.osgi.service.dal.DeviceFunction
`getOperationMetadata`, `getProperty`, `getPropertyMetadata`

Field Detail

PROPERTY_KEY

```
public static final String PROPERTY_KEY = "key"
```

Specifies a property name for a key from the keypad. The property is eventable.

See Also:

`KeypadData`

Interface Meter

org.osgi.service.dal.functions

All Superinterfaces:

DeviceFunction

```
public interface Meter
extends DeviceFunction
```

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

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

As an example, the function is easily mappable to ZigBee Simple Metering cluster and Z-Wave Meter command class. The sensor type can be:

- `Types.TYPE_PRESSURE`
- `Types.TYPE_GAS`
- `Types.TYPE_POWER`
- `Types.TYPE_WATER`
- `Types.TYPE_HEAT`
- `Types.TYPE_COLD`
- custom - vendor specific type

See Also:

`LevelData`

Field Summary

		Pag e
`String`	**`FLOW_IN`** Represents the metering consumption flow.	*101*
`String`	**`FLOW_OUT`** Represents the metering generation flow.	*101*
`String`	**`OPERATION_RESET_TOTAL`** Specifies the reset total operation name.	*102*
`String`	**`PROPERTY_CURRENT`** Specifies the current consumption property name.	*101*
`String`	**`PROPERTY_TOTAL`** Specifies the total consumption property name.	*101*
`String`	**`SERVICE_FLOW`** The service property value contains the metering flow.	*101*

Fields inherited from interface org.osgi.service.dal.DeviceFunction
`SERVICE_DESCRIPTION`, `SERVICE_DEVICE_UID`, `SERVICE_OPERATION_NAMES`, `SERVICE_PROPERTY_NAMES`, `SERVICE_REFERENCE_UIDS`, `SERVICE_TYPE`, `SERVICE_UID`, `SERVICE_VERSION`

Method Summary		Pag e
LevelData	**getCurrent**() Returns the current metering info.	102
LevelData	**getTotal**() Returns the total metering info.	102
void	**resetTotal**() Resets the total metering info.	102

Methods inherited from interface org.osgi.service.dal.DeviceFunction
getOperationMetadata, getProperty, getPropertyMetadata

Field Detail

FLOW_IN

```
public static final String FLOW_IN = "in"
```

Represents the metering consumption flow. It can be used as SERVICE_FLOW property value.

FLOW_OUT

```
public static final String FLOW_OUT = "out"
```

Represents the metering generation flow. It can be used as SERVICE_FLOW property value.

SERVICE_FLOW

```
public static final String SERVICE_FLOW = "meter.flow"
```

The service property value contains the metering flow. It's an optional property and available only if it's supported by the meter. The value type is `java.lang.String`. Possible property values:

- FLOW_IN
- FLOW_OUT

PROPERTY_CURRENT

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

Specifies the current consumption property name. The property can be read with getCurrent() getter.

PROPERTY_TOTAL

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

Specifies the total consumption property name. It has been measured since the last call of resetTotal() or device initial run. The property can be read with getTotal() getter.

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.

Method Detail

getCurrent

```
LevelData getCurrent()
              throws UnsupportedOperationException,
                     IllegalStateException,
                     DeviceException
```

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 function service object has already been unregistered.
`DeviceException` - If an operation error is available.

See Also:

`LevelData`

---

getTotal

```
LevelData getTotal()
            throws UnsupportedOperationException,
                   IllegalStateException,
                   DeviceException
```

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 function service object has already been unregistered.
`DeviceException` - If an operation error is available.

See Also:

`LevelData`

---

resetTotal

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

Resets the total metering info.

Throws:

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

Interface MultiLevelControl

org.osgi.service.dal.functions

All Superinterfaces:

DeviceFunction

```
public interface MultiLevelControl
extends DeviceFunction
```

`MultiLevelControl` Device Function provides multi-level control support. The function level is accessible with `getData()` getter, `setData(BigDecimal)` setter and `setData(BigDecimal, String)` setter.

As an example, the function is easily mappable to ZigBee Level Control and Z-Wave Multilevel Switch command class. The control type can be:

- `Types.TYPE_LIGHT`
- `Types.TYPE_TEMPERATURE`
- `Types.TYPE_FLOW`
- `Types.TYPE_PRESSURE`
- `Types.TYPE_HUMIDITY`
- `Types.TYPE_GAS`
- `Types.TYPE_SMOKE`
- `Types.TYPE_DOOR`
- `Types.TYPE_WINDOW`
- `Types.TYPE_LIQUID`
- `Types.TYPE_POWER`
- `Types.TYPE_NOISINESS`
- custom - vendor specific type

See Also:

`LevelData`

Field Summary		Page
`String`	**`PROPERTY_DATA`** Specifies the level property name.	*104*

Fields inherited from interface org.osgi.service.dal.DeviceFunction
`SERVICE_DESCRIPTION`, `SERVICE_DEVICE_UID`, `SERVICE_OPERATION_NAMES`, `SERVICE_PROPERTY_NAMES`, `SERVICE_REFERENCE_UIDS`, `SERVICE_TYPE`, `SERVICE_UID`, `SERVICE_VERSION`

Method Summary		Page
`LevelData`	**`getData`**`()` Returns `MultiLevelControl` level.	*104*
`void`	**`setData`**`(BigDecimal level)` Sets `MultiLevelControl` level to the specified value.	*104*
`void`	**`setData`**`(BigDecimal level, String unit)` Sets `MultiLevelControl` level and unit to the specified values.	*104*

Methods inherited from interface org.osgi.service.dal.DeviceFunction
`getOperationMetadata`, `getProperty`, `getPropertyMetadata`

Field Detail

PROPERTY_DATA

```
public static final String PROPERTY_DATA = "data"
```

Specifies the level property name. The property can be read with getData() getter and can be set with setData(BigDecimal) or setData(BigDecimal, String) setters.

Method Detail

getData

```
LevelData getData()
            throws UnsupportedOperationException,
                   IllegalStateException,
                   DeviceException
```

Returns MultiLevelControl level. It's a getter method for PROPERTY_DATA property.

Returns:
MultiLevelControl level.

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

See Also:
LevelData

---

setData

```
void setData(BigDecimal level)
      throws UnsupportedOperationException,
             IllegalStateException,
             DeviceException
```

Sets MultiLevelControl level to the specified value. It's a setter method for PROPERTY_DATA property.

Parameters:
level - The new control level.

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

---

setData

```
void setData(BigDecimal level,
             String unit)
      throws UnsupportedOperationException,
             IllegalStateException,
             DeviceException
```

Sets MultiLevelControl level and unit to the specified values. It's a setter method for PROPERTY_DATA property.

Parameters:

`level` - The new control level.

`unit` - The level unit.

Throws:

`UnsupportedOperationException` - If the operation is not supported.

`IllegalStateException` - If this device function service object has already been unregistered.

`DeviceException` - If an operation error is available.

Interface MultiLevelSensor

org.osgi.service.dal.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 `getData()` getter. There are no operations.

As an example, the function is easily mappable to ZigBee Illuminance Measurement, Temperature Measurement, Pressure Measurement, Flow Measurement and Relative Humidity Measurement cluster and Z-Wave Multilevel Sensor command class. The sensor type can be:

- `Types.TYPE_LIGHT`
- `Types.TYPE_TEMPERATURE`
- `Types.TYPE_FLOW`
- `Types.TYPE_PRESSURE`
- `Types.TYPE_HUMIDITY`
- `Types.TYPE_GAS`
- `Types.TYPE_SMOKE`
- `Types.TYPE_DOOR`
- `Types.TYPE_WINDOW`
- `Types.TYPE_LIQUID`
- `Types.TYPE_POWER`
- `Types.TYPE_NOISINESS`
- `Types.TYPE_RAIN`
- custom - vendor specific type

See Also:

`LevelData`

Field Summary		**Page**
`String`	**`PROPERTY_DATA`** Specifies the state property name.	*107*

Fields inherited from interface org.osgi.service.dal.DeviceFunction
`SERVICE_DESCRIPTION`, `SERVICE_DEVICE_UID`, `SERVICE_OPERATION_NAMES`, `SERVICE_PROPERTY_NAMES`, `SERVICE_REFERENCE_UIDS`, `SERVICE_TYPE`, `SERVICE_UID`, `SERVICE_VERSION`

Method Summary		**Page**
`LevelData`	**`getData`**`()` Returns the `MultiLevelSensor` current state.	*107*

Methods inherited from interface org.osgi.service.dal.DeviceFunction
`getOperationMetadata`, `getProperty`, `getPropertyMetadata`

Field Detail

PROPERTY_DATA

```
public static final String PROPERTY_DATA = "data"
```

Specifies the state property name. The property can be read with `getData()` getter.

See Also:
`LevelData`

Method Detail

getData

```
LevelData getData()
          throws UnsupportedOperationException,
                 IllegalStateException,
                 DeviceException
```

Returns the `MultiLevelSensor` current state. It's a getter method for `PROPERTY_DATA` property.

Returns:
The `MultiLevelSensor` current state.

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

See Also:
`LevelData`

Interface Types

org.osgi.service.dal.functions

```
public interface Types
```

Shares common constants for all device functions defined in this package. The defined device function types are mapped as follow:

- TYPE_LIGHT - MultiLevelControl, MultiLevelSensor, BooleanSensor and BooleanControl
- TYPE_TEMPERATURE - MultiLevelControl and MultiLevelSensor
- TYPE_FLOW - MultiLevelControl and MultiLevelSensor
- TYPE_PRESSURE - MultiLevelControl, MultiLevelSensor and Meter
- TYPE_HUMIDITY - MultiLevelControl and MultiLevelSensor
- TYPE_GAS - MultiLevelControl, MultiLevelSensor, BooleanSensor and Meter
- TYPE_SMOKE - MultiLevelControl, MultiLevelSensor and BooleanSensor
- TYPE_DOOR - MultiLevelControl, MultiLevelSensor, BooleanSensor and BooleanControl
- TYPE_WINDOW - MultiLevelControl, MultiLevelSensor, BooleanSensor and BooleanControl
- TYPE_LIQUID - MultiLevelControl and MultiLevelSensor
- TYPE_POWER - MultiLevelControl, MultiLevelSensor, BooleanSensor, BooleanControl and Meter
- TYPE_NOISINESS - MultiLevelControl and MultiLevelSensor
- TYPE_RAIN - MultiLevelSensor and BooleanSensor
- TYPE_CONTACT - BooleanSensor
- TYPE_FIRE - BooleanSensor
- TYPE_OCCUPANCY - BooleanSensor
- TYPE_WATER - BooleanSensor and Meter
- TYPE_MOTION - BooleanSensor
- TYPE_HEAT - Meter
- TYPE_COLD - Meter

Field Summary		Page
String	**TYPE_COLD** The device function type is applicable to: - Meter - indicates that the Meter measures thermal energy provided by a source. This type can be specified as a value of DeviceFunction.SERVICE_TYPE.	115
String	**TYPE_CONTACT** The device function type is applicable to: - BinarySensor - indicates that the BinarySensor can detect contact.	114
String	**TYPE_DOOR** The device function type is applicable to: - MultiLevelControl - indicates that the MultiLevelControl can control the door position.	112
String	**TYPE_FIRE** The device function type is applicable to: - BinarySensor - indicates that the BinarySensor can detect fire.	114

String	**TYPE_FLOW** The device function type is applicable to: • `MultiLevelControl` - indicates that the `MultiLevelControl` can control the flow level.	*111*
String	**TYPE_GAS** The device function type is applicable to: • `MultiLevelControl` - indicates that the `MultiLevelControl` can control the gas level.	*112*
String	**TYPE_HEAT** The device function type is applicable to: • `Meter` - indicates that the `Meter` measures thermal energy provided by a source. This type can be specified as a value of `DeviceFunction.SERVICE_TYPE`.	*115*
String	**TYPE_HUMIDITY** The device function type is applicable to: • `MultiLevelControl` - indicates that the `MultiLevelControl` can control the humidity level.	*111*
String	**TYPE_LIGHT** The device function type is applicable to: • `MultiLevelControl` - indicates that the `MultiLevelControl` can control light devices.	*110*
String	**TYPE_LIQUID** The device function type is applicable to: • `MultiLevelControl` - indicates that the `MultiLevelControl` can control the liquid level.	*113*
String	**TYPE_MOTION** The device function type is applicable to: • `BinarySensor` - indicates that the `BinarySensor` can detect motion.	*115*
String	**TYPE_NOISINESS** The device function type is applicable to: • `MultiLevelControl` - indicates that the `MultiLevelControl` can control the noise level.	*113*
String	**TYPE_OCCUPANCY** The device function type is applicable to: • `BinarySensor` - indicates that the `BinarySensor` can detect presence.	*114*

`String`	**TYPE_POWER** The device function type is applicable to: - `MultiLevelControl` - indicates that the `MultiLevelControl` can control the power level.	*113*
`String`	**TYPE_PRESSURE** The device function type is applicable to: - `MultiLevelControl` - indicates that the `MultiLevelControl` can control the pressure level.	*111*
`String`	**TYPE_RAIN** The device function type is applicable to: - `MultiLevelSensor` - indicates that the `MultiLevelSensor` can monitor the rain rate.	*114*
`String`	**TYPE_SMOKE** The device function type is applicable to: - `MultiLevelControl` - indicates that the `MultiLevelControl` can control the smoke level.	*112*
`String`	**TYPE_TEMPERATURE** The device function type is applicable to: - `MultiLevelControl` - indicates that the `MultiLevelControl` can control temperature devices.	*111*
`String`	**TYPE_WATER** The device function type is applicable to: - `BinarySensor` - indicates that the `BinarySensor` can detect water leak.	*115*
`String`	**TYPE_WINDOW** The device function type is applicable to: - `MultiLevelControl` - indicates that the `MultiLevelControl` can control the window position.	*113*

Field Detail

TYPE_LIGHT

```
public static final String TYPE_LIGHT = "light"
```

The device function type is applicable to:

- `MultiLevelControl` - indicates that the `MultiLevelControl` can control light devices. Usually, such devices are called dimmable. `MultiLevelControl` minimum value can switch off the device and `MultiLevelControl` maximum value can increase the device light to the maximum possible value.
- `MultiLevelSensor` - indicates that the sensor can monitor the light level.
- `BinarySensor` - indicates that the `BinarySensor` can detected light. `true` state means that there is light. `false` state means that there is no light.

- `BinaryControl` - indicates that there is a light device control. `true` state means that the light device will be turned on. `false` state means that the light device will be turned off.

This type can be specified as a value of `DeviceFunction.SERVICE_TYPE`.

TYPE_TEMPERATURE

```
public static final String TYPE_TEMPERATURE = "temperature"
```

The device function type is applicable to:

- `MultiLevelControl` - indicates that the `MultiLevelControl` can control temperature devices. For example, such device can be thermostat. `MultiLevelControl` minimum value is the lowest supported temperature. `MultiLevelControl` maximum value is the highest supported temperature.
- `MultiLevelSensor` - indicates that the sensor can monitor the temperature.

This type can be specified as a value of `DeviceFunction.SERVICE_TYPE`.

TYPE_FLOW

```
public static final String TYPE_FLOW = "flow"
```

The device function type is applicable to:

- `MultiLevelControl` - indicates that the `MultiLevelControl` can control the flow level. `MultiLevelControl` minimum value is the minimum supported flow level. `MultiLevelControl` maximum value is the maximum supported flow level.
- `MultiLevelSensor` - indicates that the sensor can monitor the flow level.

This type can be specified as a value of `DeviceFunction.SERVICE_TYPE`.

TYPE_PRESSURE

```
public static final String TYPE_PRESSURE = "pressure"
```

The device function type is applicable to:

- `MultiLevelControl` - indicates that the `MultiLevelControl` can control the pressure level. `MultiLevelControl` minimum value is the lowest supported pressure level. `MultiLevelControl` maximum value is the highest supported pressure level.
- `MultiLevelSensor` - indicates that the sensor can monitor the pressure level.
- `Meter` - Indicates that the `Meter` measures pressure.

This type can be specified as a value of `DeviceFunction.SERVICE_TYPE`.

TYPE_HUMIDITY

```
public static final String TYPE_HUMIDITY = "humidity"
```

The device function type is applicable to:

- `MultiLevelControl` - indicates that the `MultiLevelControl` can control the humidity level. It's typical functionality for HVAC (heating, ventilation, and air conditioning) devices.

`MultiLevelControl` minimum value is the lowest supported humidity level. `MultiLevelControl` maximum value is the highest supported humidity level.
- `MultiLevelSensor` - indicates that the sensor can monitor the humidity level.

This type can be specified as a value of `DeviceFunction.SERVICE_TYPE`.

TYPE_GAS

```
public static final String TYPE_GAS = "gas"
```

The device function type is applicable to:

- `MultiLevelControl` - indicates that the `MultiLevelControl` can control the gas level. `MultiLevelControl` minimum value is the lowest supported gas level. `MultiLevelControl` maximum value is the highest supported gas level.
- `MultiLevelSensor` - indicates that the sensor can monitor the gas level.
- `BinarySensor` - indicates that the `BinarySensor` supports gas detection. `true` state means there is gas. `false` state means that there is no gas.
- `Meter` - indicates that the `Meter` measures the gas consumption.

This type can be specified as a value of `DeviceFunction.SERVICE_TYPE`.

TYPE_SMOKE

```
public static final String TYPE_SMOKE = "smoke"
```

The device function type is applicable to:

- `MultiLevelControl` - indicates that the `MultiLevelControl` can control the smoke level. `MultiLevelControl` minimum value is the lowest supported smoke level. `MultiLevelControl` maximum value is the highest supported smoke level.
- `MultiLevelSensor` - indicates that the sensor can monitor the smoke level.
- `BinarySensor` - indicates that the `BinarySensor` can detect smoke. `true` state means that there is smoke. `false` state means that there is no rain.

This type can be specified as a value of `DeviceFunction.SERVICE_TYPE`.

TYPE_DOOR

```
public static final String TYPE_DOOR = "door"
```

The device function type is applicable to:

- `MultiLevelControl` - indicates that the `MultiLevelControl` can control the door position. `MultiLevelControl` minimum value can completely close the door. `MultiLevelControl` maximum value can open the door to the maximum allowed position.
- `MultiLevelSensor` - indicates that the sensor can monitor the door position.
- `BinarySensor` - indicates that the `BinarySensor` can detect the door state. `true` state means that the door is opened. `false` state means that the door is closed.
- `BinaryControl` - indicates that there is a door position control. `true` state means that the door will be opened. `false` state means that the the door will be closed.

This type can be specified as a value of `DeviceFunction.SERVICE_TYPE`.

TYPE_WINDOW

public static final String **TYPE_WINDOW** = "window"

The device function type is applicable to:

- MultiLevelControl - indicates that the MultiLevelControl can control the window position. MultiLevelControl minimum value can completely close the window. MultiLevelControl maximum value can open the window to the maximum allowed position.
- MultiLevelSensor - indicates that the sensor can monitor the window position.
- BinarySensor - indicates that the BinarySensor can window state. true state means that the window is opened. false state means that the window is closed.
- BinaryControl - indicates that there is a window position control. true state means that the window will be opened. false state means that the the window will be closed.

This type can be specified as a value of DeviceFunction.SERVICE_TYPE.

TYPE_LIQUID

public static final String **TYPE_LIQUID** = "liquid"

The device function type is applicable to:

- MultiLevelControl - indicates that the MultiLevelControl can control the liquid level. MultiLevelControl minimum value is the lowest supported liquid level. MultiLevelControl maximum value is the highest supported liquid level.
- MultiLevelSensor - indicates that the sensor can monitor the liquid level.

This type can be specified as a value of DeviceFunction.SERVICE_TYPE.

TYPE_POWER

public static final String **TYPE_POWER** = "power"

The device function type is applicable to:

- MultiLevelControl - indicates that the MultiLevelControl can control the power level. MultiLevelControl minimum value is the lowest supported power level. MultiLevelControl maximum value is the highest supported power level.
- MultiLevelSensor - indicates that the sensor can monitor the power level.
- BinarySensor - indicates that the BinarySensor can detect motion. true state means that there is power restore. false state means that there is power cut.
- BinaryControl - indicates that there is electricity control. true state means that the power will be restored. false state means that the power will be cut.
- Meter - indicates that the Meter measures the power consumption.

This type can be specified as a value of DeviceFunction.SERVICE_TYPE.

TYPE_NOISINESS

public static final String **TYPE_NOISINESS** = "noisiness"

The device function type is applicable to:

- `MultiLevelControl` - indicates that the `MultiLevelControl` can control the noise level. `MultiLevelControl` minimum value is the lowest supported noise level. `MultiLevelControl` maximum value is the highest supported noise level.
- `MultiLevelSensor` - indicates that the sensor can monitor the noise level.

This type can be specified as a value of `DeviceFunction.SERVICE_TYPE`.

TYPE_RAIN

`public static final String` **`TYPE_RAIN`** `= "rain"`

The device function type is applicable to:

- `MultiLevelSensor` - indicates that the `MultiLevelSensor` can monitor the rain rate. It's not applicable to `MultiLevelControl`.
- `BinarySensor` - indicates that the `BinarySensor` can detect rain. `true` state means that there is rain. `false` state means that there is no rain.

This type can be specified as a value of `DeviceFunction.SERVICE_TYPE`.

TYPE_CONTACT

`public static final String` **`TYPE_CONTACT`** `= "contact"`

The device function type is applicable to:

- `BinarySensor` - indicates that the `BinarySensor` can detect contact. `true` state means that there is contact. `false` state means that there is no contact.

This type can be specified as a value of `DeviceFunction.SERVICE_TYPE`.

TYPE_FIRE

`public static final String` **`TYPE_FIRE`** `= "fire"`

The device function type is applicable to:

- `BinarySensor` - indicates that the `BinarySensor` can detect fire. `true` state means that there is fire. `false` state means that there is no fire.

This type can be specified as a value of `DeviceFunction.SERVICE_TYPE`.

TYPE_OCCUPANCY

`public static final String` **`TYPE_OCCUPANCY`** `= "occupancy"`

The device function type is applicable to:

- `BinarySensor` - indicates that the `BinarySensor` can detect presence. `true` state means that someone is detected. `false` state means that nobody is detected.

This type can be specified as a value of `DeviceFunction.SERVICE_TYPE`.

TYPE_WATER

```
public static final String TYPE_WATER = "water"
```

The device function type is applicable to:

- BinarySensor - indicates that the BinarySensor can detect water leak. true state means that there is water leak. false state means that there is no water leak.
- Meter - indicates that the Meter measures water consumption.

This type can be specified as a value of DeviceFunction.SERVICE_TYPE.

TYPE_MOTION

```
public static final String TYPE_MOTION = "motion"
```

The device function type is applicable to:

- BinarySensor - indicates that the BinarySensor can detect motion. true state means that there is motion detection. false state means that there is no motion detection.

This type can be specified as a value of DeviceFunction.SERVICE_TYPE.

TYPE_HEAT

```
public static final String TYPE_HEAT = "heat"
```

The device function type is applicable to:

- Meter - indicates that the Meter measures thermal energy provided by a source.

This type can be specified as a value of DeviceFunction.SERVICE_TYPE.

TYPE_COLD

```
public static final String TYPE_COLD = "cold"
```

The device function type is applicable to:

- Meter - indicates that the Meter measures thermal energy provided by a source.

This type can be specified as a value of DeviceFunction.SERVICE_TYPE.

Package org.osgi.service.dal.functions.data

Device Function Data 1.0.

See:

Description

Class Summary		Page
AlarmData	Device Function alarm data.	117
BooleanData	Device Function boolean data wrapper.	122
KeypadData	Represents a keypad event data that is collected when a change with some key from device keypad has occurred.	124
LevelData	Device Function level data wrapper.	128

Package org.osgi.service.dal.functions.data Description

Device Function Data 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.dal.functions.data; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

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

Class AlarmData

org.osgi.service.dal.functions.data

```
java.lang.Object
  └ org.osgi.service.dal.DeviceFunctionData
      └ org.osgi.service.dal.functions.data.AlarmData
```

All Implemented Interfaces:

Comparable

```
public class AlarmData
extends DeviceFunctionData
```

Device Function alarm data. It cares about the alarm type, severity, timestamp and additional metadata. It doesn't support unit. The alarm type is mapped to `DeviceFunctionData` value.

See Also:

Alarm, DeviceFunctionData

Field Summary		Page
int	**severity** Represents the alarm severity.	120
static int	**SEVERITY_HIGH** The severity rating indicates that there is an alarm with high priority.	120
static int	**SEVERITY_LOW** The severity rating indicates that there is an alarm with lowest priority.	119
static int	**SEVERITY_MEDIUM** The severity rating indicates that there is an alarm with medium priority.	119
static int	**SEVERITY_NONE** The severity constant indicates that there is no severity rating for this alarm.	119
static int	**SEVERITY_URGENT** The severity rating indicates that there an urgent alarm.	120
int	**type** Represents the alarm type.	120
static int	**TYPE_COLD** The alarm type indicates that temperature is too low.	118
static int	**TYPE_GAS_CO** The alarm type indicates that carbon monoxide is detected.	119
static int	**TYPE_GAS_CO2** The alarm type indicates that carbon dioxide is detected.	118
static int	**TYPE_HEAT** The alarm type indicates that temperature is too high.	118
static int	**TYPE_HW_FAIL** The alarm type indicates that there is hardware failure.	119
static int	**TYPE_POWER_FAIL** The alarm type indicates a power cut.	119
static int	**TYPE_SMOKE** The alarm type indicates that smoke is detected.	118
static int	**TYPE_SW_FAIL** The alarm type indicates that there is software failure.	119

static int	**TYPE_WATER** The alarm type indicates that water leak is detected.	*119*

Fields inherited from class org.osgi.service.dal.DeviceFunctionData
META_INFO_DESCRIPTION, metadata, timestamp

Constructor Summary	*Page*
AlarmData(Map fields) Constructs new AlarmData instance with the specified field values.	*120*
AlarmData(long timestamp, Map metadata, int severity, int type) Constructs new AlarmData instance with the specified arguments.	*120*

Method Summary		*Page*
int	**compareTo**(Object o)	*121*
int	**getSeverity**() Returns the alarm severity.	*121*
int	**getType**() Returns the alarm type.	*121*

Methods inherited from class org.osgi.service.dal.DeviceFunctionData
getMetadata, getTimestamp

Field Detail

TYPE_SMOKE

```
public static final int TYPE_SMOKE = 1
```

The alarm type indicates that smoke is detected.

TYPE_HEAT

```
public static final int TYPE_HEAT = 2
```

The alarm type indicates that temperature is too high.

TYPE_COLD

```
public static final int TYPE_COLD = 3
```

The alarm type indicates that temperature is too low.

TYPE_GAS_CO2

```
public static final int TYPE_GAS_CO2 = 4
```

The alarm type indicates that carbon dioxide is detected.

TYPE_GAS_CO

```
public static final int TYPE_GAS_CO = 5
```

The alarm type indicates that carbon monoxide is detected.

TYPE_WATER

```
public static final int TYPE_WATER = 6
```

The alarm type indicates that water leak is detected.

TYPE_POWER_FAIL

```
public static final int TYPE_POWER_FAIL = 7
```

The alarm type indicates a power cut.

TYPE_HW_FAIL

```
public static final int TYPE_HW_FAIL = 8
```

The alarm type indicates that there is hardware failure.

TYPE_SW_FAIL

```
public static final int TYPE_SW_FAIL = 9
```

The alarm type indicates that there is software failure.

SEVERITY_NONE

```
public static final int SEVERITY_NONE = 0
```

The severity constant indicates that there is no severity rating for this alarm.

SEVERITY_LOW

```
public static final int SEVERITY_LOW = 1
```

The severity rating indicates that there is an alarm with lowest priority.

SEVERITY_MEDIUM

```
public static final int SEVERITY_MEDIUM = 2
```

The severity rating indicates that there is an alarm with medium priority. The severity priority is higher than `SEVERITY_LOW` and lower than `SEVERITY_HIGH`.

SEVERITY_HIGH

```
public static final int SEVERITY_HIGH = 3
```

The severity rating indicates that there is an alarm with high priority. The severity priority is higher than `SEVERITY_MEDIUM` and lower than `SEVERITY_URGENT`.

SEVERITY_URGENT

```
public static final int SEVERITY_URGENT = 4
```

The severity rating indicates that there an urgent alarm. That severity has highest priority.

severity

```
public final int severity
```

Represents the alarm severity. The field is accessible with `getSeverity()` getter. The vendor can define own alarm severity ratings with negative values.

type

```
public final int type
```

Represents the alarm type. The field is accessible with `getType()` getter. The vendor can define own alarm types with negative values.

Constructor Detail

AlarmData

```
public AlarmData(Map fields)
```

Constructs new `AlarmData` instance with the specified field values. The map keys must match to the field names. The map values will be assigned to the appropriate class fields. For example, the maps can be: {"severity"=Integer(1)...}. That map will initialize the "severity" field with 1.

Parameters:
`fields` - Contains the new `AlarmData` instance field values.

AlarmData

```
public AlarmData(long timestamp,
                 Map metadata,
                 int severity,
                 int type)
```

Constructs new `AlarmData` instance with the specified arguments.

Parameters:
`timestamp` - The alarm data timestamp.
`metadata` - The alarm data metadata.
`severity` - The alarm data severity.

`type` - The alarm data type.

Method Detail

getType

```
public int getType()
```

Returns the alarm type. The type can be one of the predefined:

- `TYPE_SMOKE`
- `TYPE_HEAT`
- `TYPE_COLD`
- `TYPE_GAS_CO`
- `TYPE_GAS_CO2`
- `TYPE_WATER`
- `TYPE_POWER_FAIL`
- `TYPE_HW_FAIL`
- `TYPE_SW_FAIL`

The vendor can define own alarm types with negative values.

Returns:
The alarm type.

---

getSeverity

```
public int getSeverity()
```

Returns the alarm severity.

Returns:
The alarm severity.

---

compareTo

```
public int compareTo(Object o)
```

Specified by:
`compareTo` in interface `Comparable`

Class BooleanData

org.osgi.service.dal.functions.data

```
java.lang.Object
  └org.osgi.service.dal.DeviceFunctionData
      └org.osgi.service.dal.functions.data.BooleanData
```

All Implemented Interfaces:

Comparable

```
public class BooleanData
extends DeviceFunctionData
```

Device Function boolean data wrapper. It can contain a boolean value, timestamp and additional metadata. It doesn't support measurement unit.

See Also:

BooleanControl, BooleanSensor, DeviceFunctionData

Field Summary		Page
boolean	**value** Represents the boolean value.	122

Fields inherited from class org.osgi.service.dal.DeviceFunctionData
META_INFO_DESCRIPTION, metadata, timestamp

Constructor Summary	Page
BooleanData(Map fields) Constructs new `BooleanData` instance with the specified field values.	123
BooleanData(long timestamp, Map metadata, boolean value) Constructs new `BooleanData` instance with the specified arguments.	123

Method Summary		Page
int	**compareTo**(Object o)	123
boolean	**getValue**() Returns `BooleanData` value.	123

Methods inherited from class org.osgi.service.dal.DeviceFunctionData
getMetadata, getTimestamp

Field Detail

value

```
public final boolean value
```

Represents the boolean value. The field is accessible with `getValue()` getter.

Constructor Detail

BooleanData

```
public BooleanData(Map fields)
```

Constructs new `BooleanData` instance with the specified field values. The map keys must match to the field names. The map values will be assigned to the appropriate class fields. For example, the maps can be: {"data"=Boolean(true)...}. That map will initialize the "data" field with `true`.

Parameters:
`fields` - Contains the new `BooleanData` instance field values.

BooleanData

```
public BooleanData(long timestamp,
                   Map metadata,
                   boolean value)
```

Constructs new `BooleanData` instance with the specified arguments.

Parameters:
`timestamp` - The boolean data timestamp.
`metadata` - The boolean data metadata.
`value` - The boolean value.

Method Detail

getValue

```
public boolean getValue()
```

Returns `BooleanData` value.

Returns:
`BooleanData` value.

---

compareTo

```
public int compareTo(Object o)
```

Specified by:
`compareTo` in interface `Comparable`

Class KeypadData

org.osgi.service.dal.functions.data

```
java.lang.Object
  └org.osgi.service.dal.DeviceFunctionData
      └org.osgi.service.dal.functions.data.KeypadData
```

All Implemented Interfaces:

Comparable

```
public class KeypadData
extends DeviceFunctionData
```

Represents a keypad event data that is collected when a change with some key from device keypad has occurred. The key code is mapped to `DeviceFunctionData` value.

See Also:

Keypad, DeviceFunctionData

Field Summary		Page
static int	**EVENT_TYPE_PRESSED** Represents a keypad event type for a key pressed.	125
static int	**EVENT_TYPE_PRESSED_DOUBLE** Represents a keypad event type for a double key pressed.	125
static int	**EVENT_TYPE_PRESSED_DOUBLE_LONG** Represents a keypad event type for a double and long key pressed.	125
static int	**EVENT_TYPE_PRESSED_LONG** Represents a keypad event type for a long key pressed.	125
static int	**EVENT_TYPE_RELEASED** Represents a keypad event type for a key released.	125
int	**eventType** Represents the keypad event type.	125
int	**keyCode** Represents the key code.	126
String	**keyName** Represents the key name, if it's available.	126

Fields inherited from class org.osgi.service.dal.DeviceFunctionData
META_INFO_DESCRIPTION, metadata, timestamp

Constructor Summary	Page
KeypadData(Map fields) Constructs new `KeypadData` instance with the specified field values.	126
KeypadData(long timestamp, Map metadata, int eventType, int keyCode, String keyName) Constructs new `KeypadData` instance with the specified arguments.	126

Method Summary		Page
int	**compareTo**(Object o)	127

int	**getEventType**() Returns the event type.	126
int	**getKeyCode**() The code of the key.	127
String	**getKeyName**() Represents a human readable name of the corresponding key code.	127

Methods inherited from class org.osgi.service.dal.DeviceFunctionData
getMetadata, getTimestamp

Field Detail

EVENT_TYPE_PRESSED

`public static final int EVENT_TYPE_PRESSED = 1`

Represents a keypad event type for a key pressed.

EVENT_TYPE_PRESSED_LONG

`public static final int EVENT_TYPE_PRESSED_LONG = 2`

Represents a keypad event type for a long key pressed.

EVENT_TYPE_PRESSED_DOUBLE

`public static final int EVENT_TYPE_PRESSED_DOUBLE = 3`

Represents a keypad event type for a double key pressed.

EVENT_TYPE_PRESSED_DOUBLE_LONG

`public static final int EVENT_TYPE_PRESSED_DOUBLE_LONG = 4`

Represents a keypad event type for a double and long key pressed.

EVENT_TYPE_RELEASED

`public static final int EVENT_TYPE_RELEASED = 5`

Represents a keypad event type for a key released.

eventType

`public final int eventType`

Represents the keypad event type. The vendor can define own event types with negative values. The field is accessible with `getEventType()` getter.

keyName

```
public final String keyName
```

Represents the key name, if it's available. The field is accessible with `getKeyName()` getter.

keyCode

```
public final int keyCode
```

Represents the key code. This field is mandatory and it holds the semantics(meaning) of the key. The field is accessible with `getKeyCode()` getter.

Constructor Detail

KeypadData

```
public KeypadData(Map fields)
```

Constructs new `KeypadData` instance with the specified field values. The map keys must match to the field names. The map values will be assigned to the appropriate class fields. For example, the maps can be: {"eventType"=Integer(1)...}. That map will initialize the "eventType" field with 1.

Parameters:

`fields` - Contains the new `KeypadData` instance field values.

KeypadData

```
public KeypadData(long timestamp,
                  Map metadata,
                  int eventType,
                  int keyCode,
                  String keyName)
```

Constructs new `KeypadData` instance with the specified arguments.

Parameters:

`timestamp` - The data timestamp.
`metadata` - The data metadata.
`eventType` - The data event type.
`keyCode` - The data key code.
`keyName` - The data key name.

Method Detail

getEventType

```
public int getEventType()
```

Returns the event type. The vendor can define own event types with negative values.

Returns:

The event type.

getKeyCode

```
public int getKeyCode()
```

The code of the key. This field is mandatory and it holds the semantics(meaning) of the key.

Returns:
The key code.

---

getKeyName

```
public String getKeyName()
```

Represents a human readable name of the corresponding key code. This field is optional and sometimes it could be missed(might be `null`).

Returns:
A string with the name of the key or `null` if not specified.

---

compareTo

```
public int compareTo(Object o)
```

Specified by:
`compareTo` in interface `Comparable`

Class LevelData

org.osgi.service.dal.functions.data

```
java.lang.Object
  └org.osgi.service.dal.DeviceFunctionData
      └org.osgi.service.dal.functions.data.LevelData
```

All Implemented Interfaces:

Comparable

```
public class LevelData
extends DeviceFunctionData
```

Device Function level data wrapper. It supports all properties defined in `DeviceFunctionData`.

See Also:

MultiLevelControl, MultiLevelSensor, Meter, DeviceFunctionData

---

Field Summary		Page
BigDecimal	**level** Represents the current level.	129
String	**unit** Represent the unit as it's defined in `PropertyMetadata.META_INFO_UNITS`.	129

Fields inherited from class org.osgi.service.dal.DeviceFunctionData
META_INFO_DESCRIPTION, metadata, timestamp

Constructor Summary	Page
LevelData(Map fields) Constructs new `LevelData` instance with the specified field values.	129
LevelData(long timestamp, Map metadata, String unit, BigDecimal level) Constructs new `LevelData` instance with the specified arguments.	129

Method Summary		Page
int	**compareTo**(Object o)	130
BigDecimal	**getLevel**() Returns `LevelData` value.	129
String	**getUnit**() Returns `LevelData` unit as it's specified in `PropertyMetadata.META_INFO_UNITS` or `null` if the unit is missing.	130

Methods inherited from class org.osgi.service.dal.DeviceFunctionData
getMetadata, getTimestamp

Field Detail

unit

```
public final String unit
```

Represent the unit as it's defined in `PropertyMetadata.META_INFO_UNITS`. The field is optional. The field is accessible with `getUnit()` getter.

level

```
public final BigDecimal level
```

Represents the current level. It's mandatory field. The field is accessible with `getLevel()` getter.

Constructor Detail

LevelData

```
public LevelData(Map fields)
```

Constructs new `LevelData` instance with the specified field values. The map keys must match to the field names. The map values will be assigned to the appropriate class fields. For example, the maps can be: {"level"=BigDecimal(1)...}. That map will initialize the "level" field with 1.

Parameters:

`fields` - Contains the new `LevelData` instance field values.

LevelData

```
public LevelData(long timestamp,
                 Map metadata,
                 String unit,
                 BigDecimal level)
```

Constructs new `LevelData` instance with the specified arguments.

Parameters:

`timestamp` - The data timestamp.
`metadata` - The data metadata.
`unit` - The data unit.
`level` - The level value.

Method Detail

getLevel

```
public BigDecimal getLevel()
```

Returns `LevelData` value. The value type is `BigDecimal` instead of `double` to guarantee value accuracy.

Returns:

The `LevelData` value.

getUnit

```
public String getUnit()
```

Returns `LevelData` unit as it's specified in `PropertyMetadata.META_INFO_UNITS` or `null` if the unit is missing.

Returns:

The value unit or `null` if the unit is missing.

compareTo

```
public int compareTo(Object o)
```

Specified by:

`compareTo` in interface `Comparable`

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 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	Device.remove()

ACTION_SET_NAME	Device.setName(String)

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.service.dal.DevicePermission(“dal.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.service.dal.DevicePermission(“*”, “remove”)
```

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

```
public class DeviceImpl implements Device {
   public void start() {
      securityManager.checkPermission(new DevicePermission(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. 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