Device Abstraction Layer

Draft

96 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

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

1 Introduction

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

This RFC defines a new device abstraction API in OSGi.

2 Application Domain

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

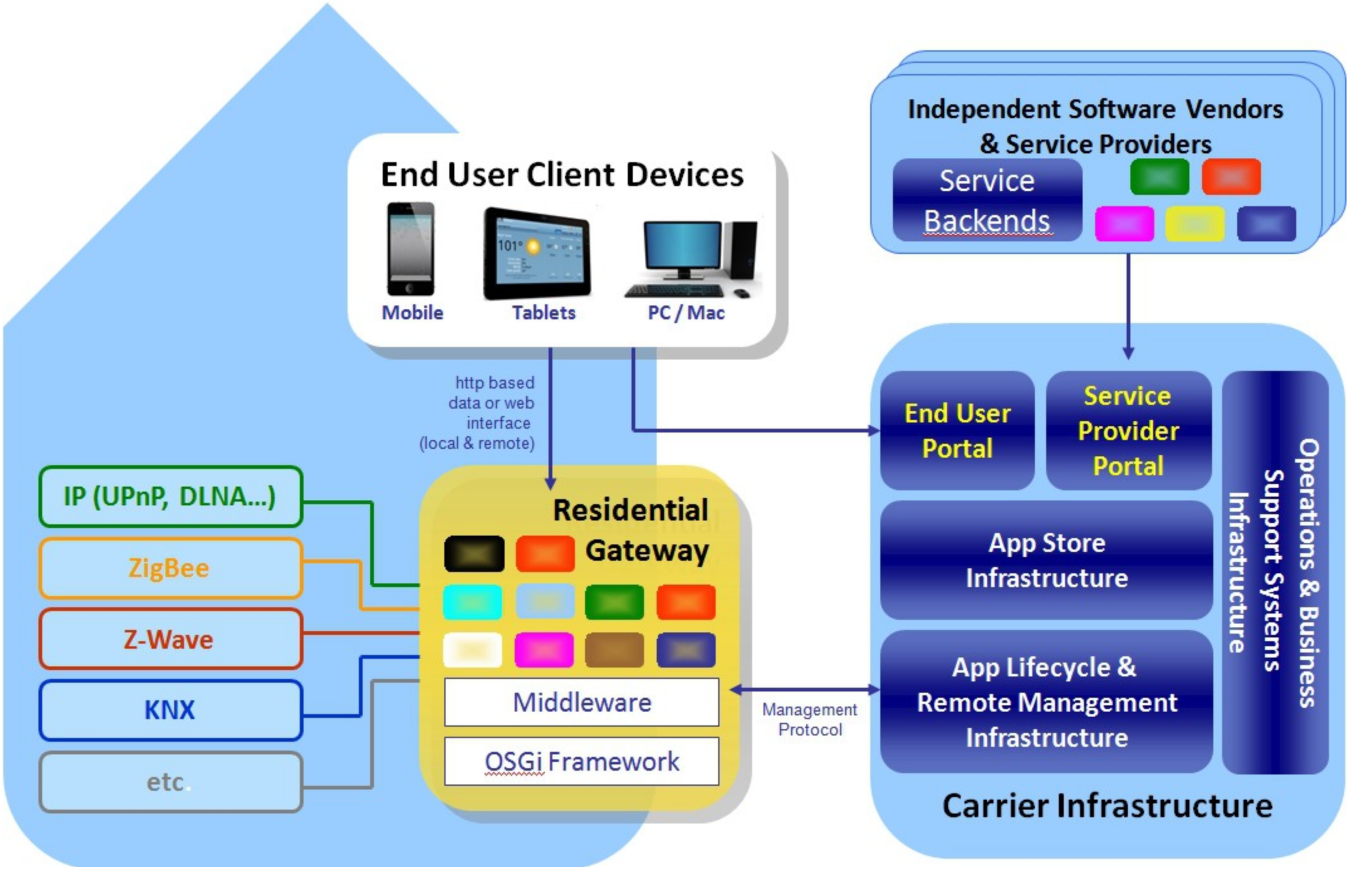

Illustration 1

In this architecture the application logic is distributed between:

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

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

The existing OSGi specifications which address related topics are:

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

3 Problem Description

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

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

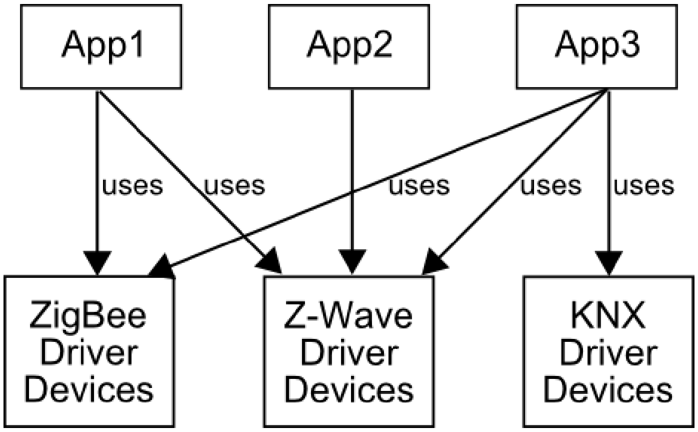

Illustration 2

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

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

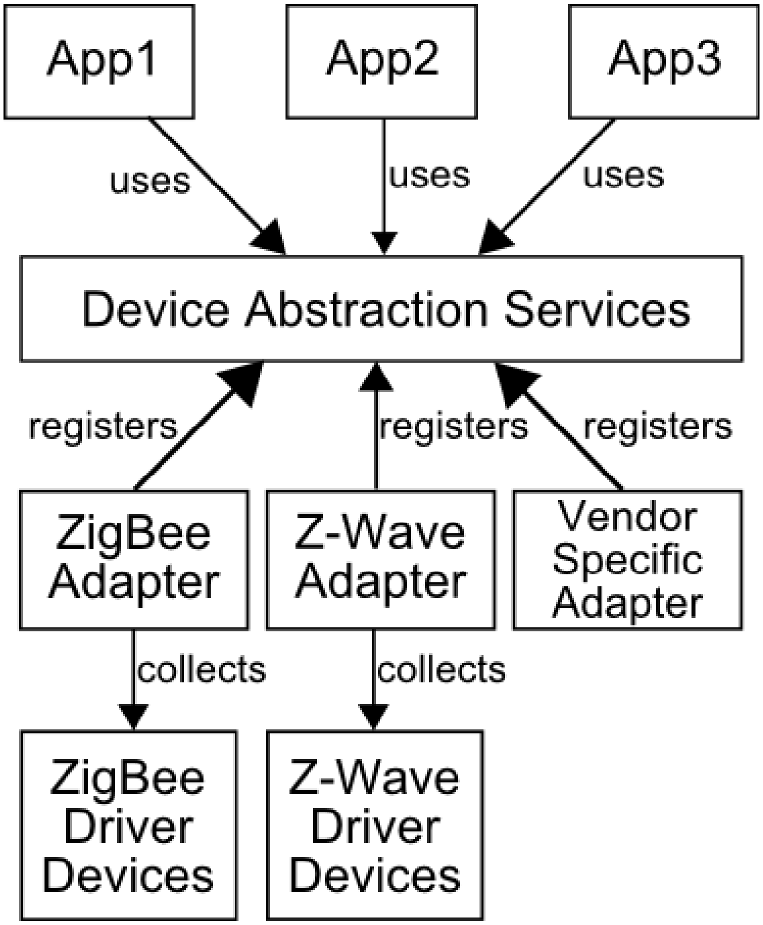

Illustration 3

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

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

4 Requirements

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

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

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

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

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

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

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

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

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

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

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

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

5 Technical Solution

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

5.1 Device Access Category

The device access category is called “FunctionalDevice”. The category name is defined as a value of `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.

5.2 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 enable, disable, 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 4:

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

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

  UID – device unique identifier

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

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

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

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

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

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

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

  - `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_DISABLED – Indicates that the device is currently disabled. The device is not available for operations. All transitions to and from this status are described in detail in Transitions to and from STATUS_DISABLED section.
  - 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.PROPERTY_STATUS_DETAIL – Provides the reason for the current device status. It's an optional property. The property value cannot be externally set or modified. The value type is java.lang.Integer. There are two value categories. Positive values indicate the reason for the current status like 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_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_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.PROPERTY_HARDWARE_VENDOR – Specifies the device hardware vendor. It's an optional property. The property value can be externally set. The value type is java.lang.String.

- Device.PROPERTY_HARDWARE_VERSION – Specifies the device hardware version. It's an optional property. The property value can be externally set. The value type is java.lang.String.

- Device.PROPERTY_FIRMWARE_VENDOR – Specifies the device firmware vendor. It's an optional property. The property value can be externally set. The value type is java.lang.String.

- Device.PROPERTY_FIRMWARE_VERSION – Specifies the device firmware version. It's an optional property. The property value can be externally set. The value type is java.lang.String.

- Device.PROPERTY_TYPES – Specified the device types. It's an optional property. The property value can be externally set or modified. The value type is java.lang.String[]. Custom types are allowed, but they must not overlap the specified types. Currently, only one type is specified:

  - Device.TYPE_PERIPHERAL – Indicates that the device is peripheral. Usually, those devices are base and contains some meta information.

- Device.PROPERTY_MODEL – Specifies the device model. It's an optional property. The property value can be externally set. The value type is java.lang.String.

- `Device.PROPERTY_SERIAL_NUMBER` – Specifies the device serial number. It's an optional property. The property value can be externally set. The value type is `java.lang.String`.

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.PROPERTY_STATUS + '=' + Device.STATUS_ONLINE + ')');
if (null == deviceSRefs) {
  return; // no such services
}
for (int i = 0; i < deviceSRefs.length; i++) {
  printDevice(deviceSRefs[i]);
}
```

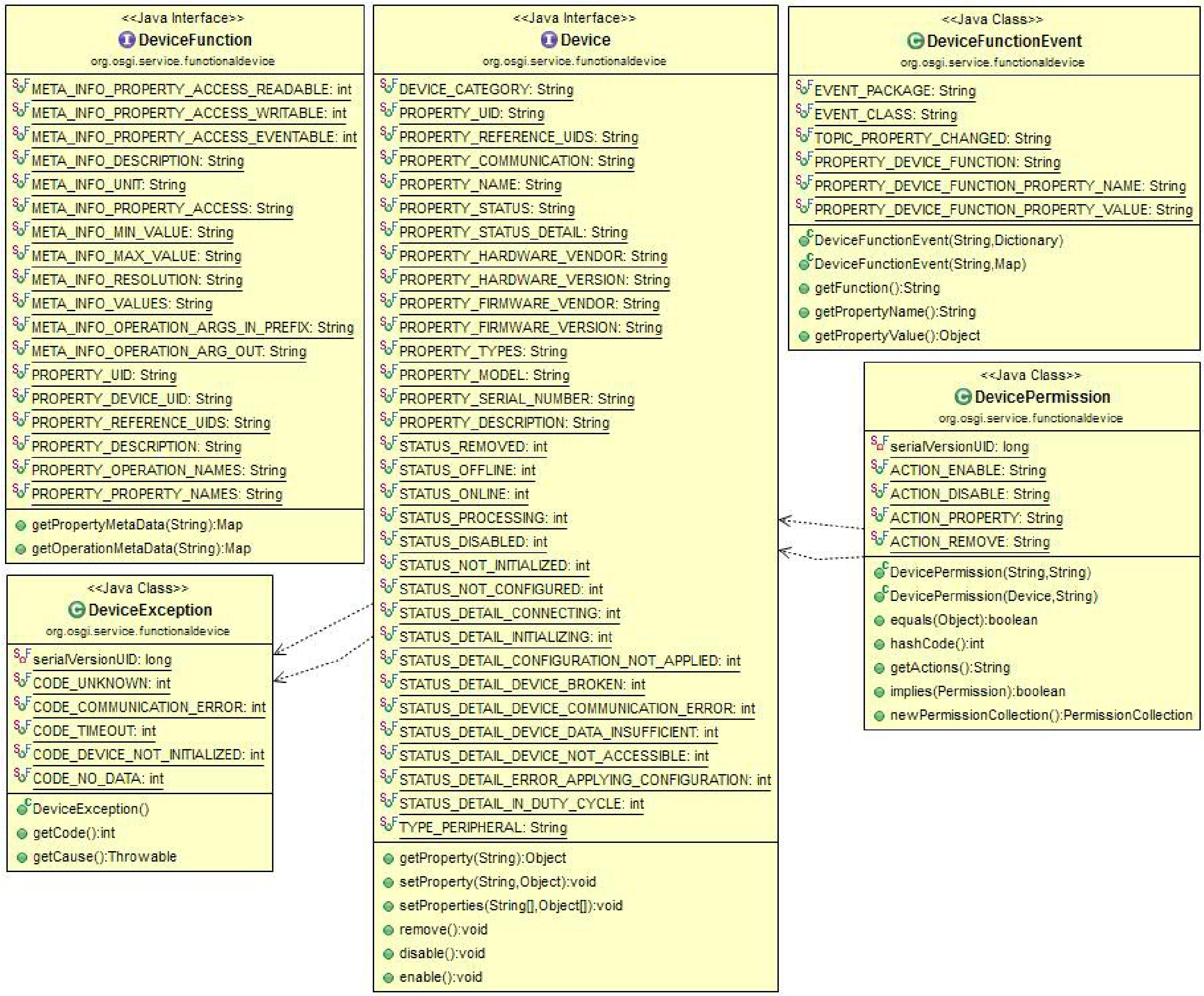
<<Java Interface>>
DeviceFunction
org.osgi.service.functionaldevice
META_INFO_PROPERTY_ACCESS_READABLE: int
META_INFO_PROPERTY_ACCESS_WRITABLE: int
META_INFO_PROPERTY_ACCESS_EVENTABLE: int
META_INFO_DESCRIPTION: String
META_INFO_UNIT: String
META_INFO_PROPERTY_ACCESS: String
META_INFO_MIN_VALUE: String
META_INFO_MAX_VALUE: String
META_INFO_RESOLUTION: String
META_INFO_VALUES: String
META_INFO_OPERATION_ARGS_IN_PREFIX: String
META_INFO_OPERATION_ARG_OUT: String
PROPERTY_UID: String
PROPERTY_DEVICE_UID: String
PROPERTY_REFERENCE_UIDS: String
PROPERTY_DESCRIPTION: String
PROPERTY_OPERATION_NAMES: String
PROPERTY_PROPERTY_NAMES: String
getPropertyMetaData(String):Map
getOperationMetaData(String):Map
<<Java Interface>>
Device
org.osgi.service.functionaldevice
DEVICE_CATEGORY: String
PROPERTY_UID: String
PROPERTY_REFERENCE_UIDS: String
PROPERTY_COMMUNICATION: String
PROPERTY_NAME: String
PROPERTY_STATUS: String
PROPERTY_STATUS_DETAIL: String
PROPERTY_HARDWARE_VENDOR: String
PROPERTY_HARDWARE_VERSION: String
PROPERTY_FIRMWARE_VENDOR: String
PROPERTY_FIRMWARE_VERSION: String
PROPERTY_TYPES: String
PROPERTY_MODEL: String
PROPERTY_SERIAL_NUMBER: String
PROPERTY_DESCRIPTION: String
STATUS_REMOVED: int
STATUS_OFFLINE: int
STATUS_ONLINE: int
STATUS_PROCESSING: int
STATUS_DISABLED: int
STATUS_NOT_INITIALIZED: int
STATUS_NOT_CONFIGURED: int
STATUS_DETAIL_CONNECTING: int
STATUS_DETAIL_INITIALIZING: int
STATUS_DETAIL_CONFIGURATION_NOT_APPLIED: int
STATUS_DETAIL_DEVICE_BROKEN: int
STATUS_DETAIL_DEVICE_COMMUNICATION_ERROR: int
STATUS_DETAIL_DEVICE_DATA_INSUFFICIENT: int
STATUS_DETAIL_DEVICE_NOT_ACCESSIBLE: int
STATUS_DETAIL_ERROR_APPLYING_CONFIGURATION: int
STATUS_DETAIL_IN_DUTY_CYCLE: int
TYPE_PERIPHERAL: String
getProperty(String):Object
setProperty(String,Object):void
setProperties(String[],Object[]):void
remove():void
disable():void
enable():void
<<Java Class>>
DeviceFunctionEvent
org.osgi.service.functionaldevice
EVENT_PACKAGE: String
EVENT_CLASS: String
TOPIC_PROPERTY_CHANGED: String
PROPERTY_DEVICE_FUNCTION: String
PROPERTY_DEVICE_FUNCTION_PROPERTY_NAME: String
PROPERTY_DEVICE_FUNCTION_PROPERTY_VALUE: String
DeviceFunctionEvent(String,Dictionary)
DeviceFunctionEvent(String,Map)
getFunction():String
getPropertyName():String
getPropertyValue():Object
<<Java Class>>
DevicePermission
org.osgi.service.functionaldevice
serialVersionUID: long
ACTION_ENABLE: String
ACTION_DISABLE: String
ACTION_PROPERTY: String
ACTION_REMOVE: String
DevicePermission(String,String)
DevicePermission(Device,String)
equals(Object):boolean
hashCode():int
getActions():String
implies(Permission):boolean
newPermissionCollection():PermissionCollection
<<Java Class>>
DeviceException
org.osgi.service.functionaldevice
serialVersionUID: long
CODE_UNKNOWN: int
CODE_COMMUNICATION_ERROR: int
CODE_TIMEOUT: int
CODE_DEVICE_NOT_INITIALIZED: int
CODE_NO_DATA: int
DeviceException()
getCode():int
getCause():Throwable

Illustration 4

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

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

5.2.1 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 5.

The related service property is Device.PROPERTY_REFERENCE_UIDS.

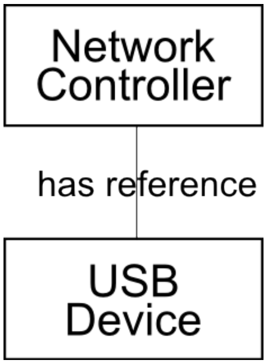

Illustration 5

5.2.2 Device Service Disabling and Enabling

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

5.2.3 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.

5.2.4 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.

5.3 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 and described in detail in the next sections.

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

Table 2

5.3.1 Transitions to STATUS_REMOVED

The 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 should be `STATUS_REMOVED`. It requires a synchronous execution of the operation.

5.3.2 Transitions to and from STATUS_OFFLINE

The `STATUS_OFFLINE` indicates that the 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_DISABLED – device is currently disabled. The status can be set as a result of Device.disable() method call.
- 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_DISABLED – Disabled device is not accessible any more. The status can be set as a result of Device.enable() method call.
- 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 6.

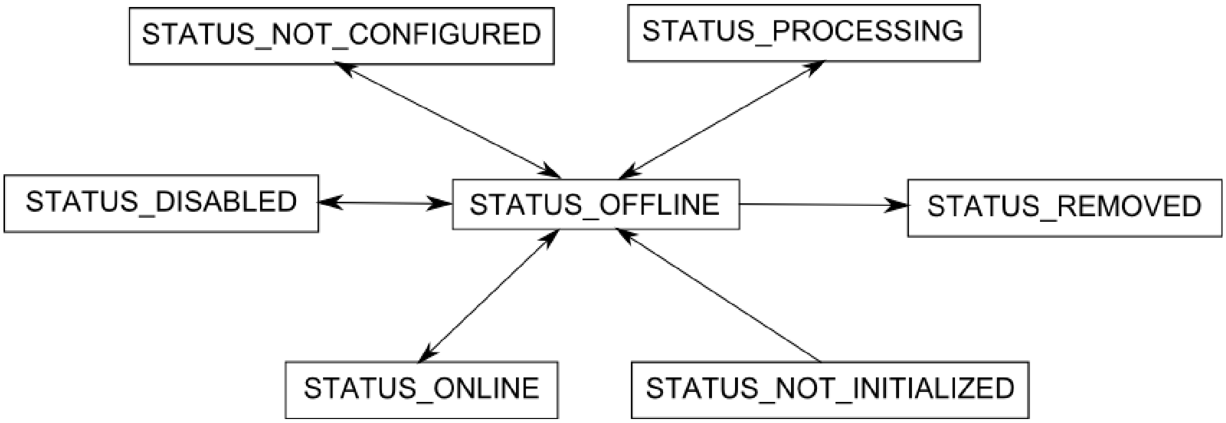

Illustration 6

5.3.3 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_DISABLED – device is currently disabled. The status can be set as a result of Device.disable() method call.
- 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_DISABLED – The device is enabled. The status can be set as a result of Device.enable() method call.
- 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 7.

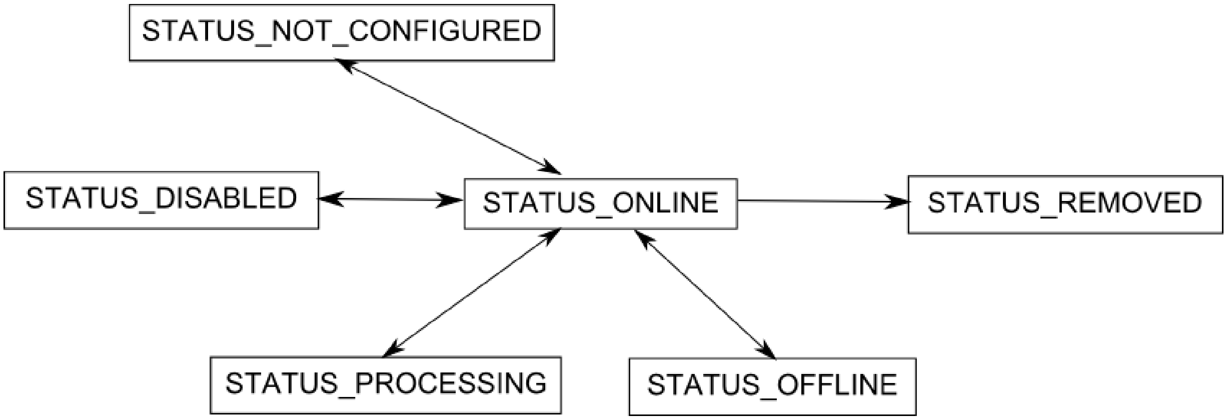

Illustration 7

5.3.4 Transitions to and from STATUS_PROCESSING

The status indicates that the device is currently busy with an operation. It can be time consuming operation and can result to any other status. The operation processing can be reached by any other status except STATUS_REMOVED. An example, offline device requires some data processing to become online. It will apply the statuses STATUS_OFFLINE, STATUS_PROCESSING and STATUS_ONLINE. Transitions to and from this status are:

- From STATUS_PROCESSING to STATUS_REMOVED – 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_DISABLED – device is currently disabled. The status can be set as a result of Device.disable() method call.
- 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_DISABLED – The device is enabled and busy with an operation. The status can be set as a result of Device.enable() method call.
- 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 8.

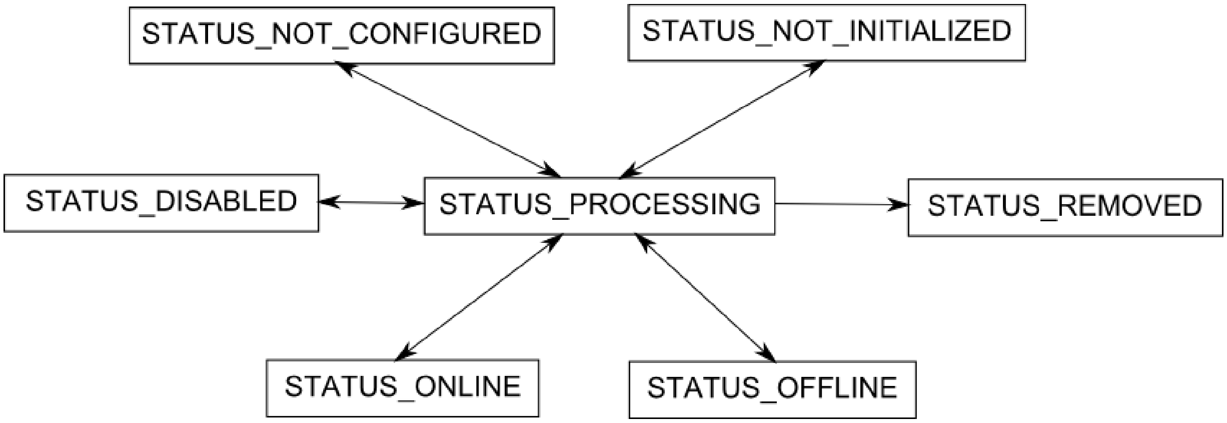

Illustration 8

5.3.5 Transitions to and from STATUS_DISABLED

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

- From STATUS_DISABLED to STATUS_REMOVED – device is removed. The status can be set as a result of Device.remove() method call.
- From STATUS_DISABLED to STATUS_PROCESSING – device is enabled and device data is processing. The status can be set as a result of Device.enable() method call.

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

The possible transitions are summarized on Illustration 9.

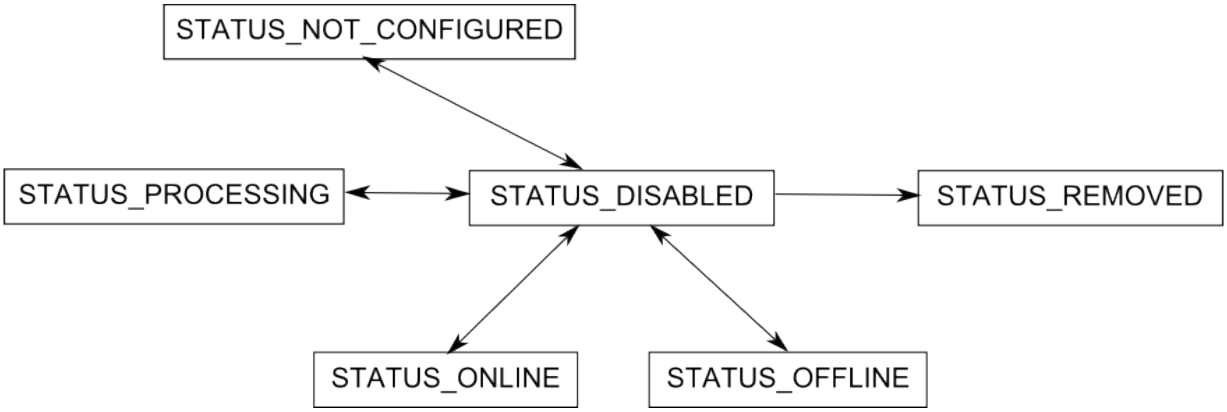

Illustration 9

5.3.6 Transitions to and from STATUS_NOT_INITIALIZED

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

- From STATUS_NOT_INITIALIZED to STATUS_REMOVED – 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_DISABLED – That transition is not possible. Not initialized device requires data processing to become operable.
- 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_DISABLED – That transition is not possible. Device has to be initialized to be operable.
- 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 10.

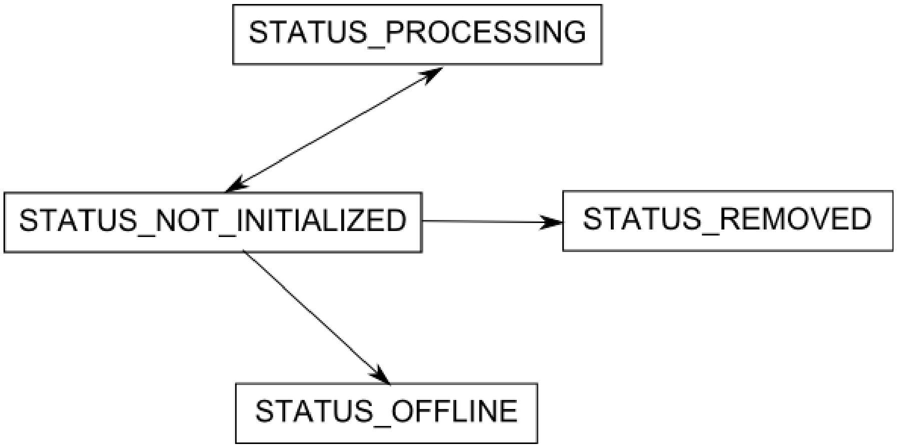

Illustration 10

5.3.7 Transitions to and from STATUS_NOT_CONFIGURED

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

- From STATUS_NOT_CONFIGURED to STATUS_REMOVED – 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_DISABLED – device is currently disabled. The status can be set as a result of Device.disable() method call.
- 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_DISABLED – The device is enabled but has a pending configuration. The status can be set as a result of Device.enable() method call.
- 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.

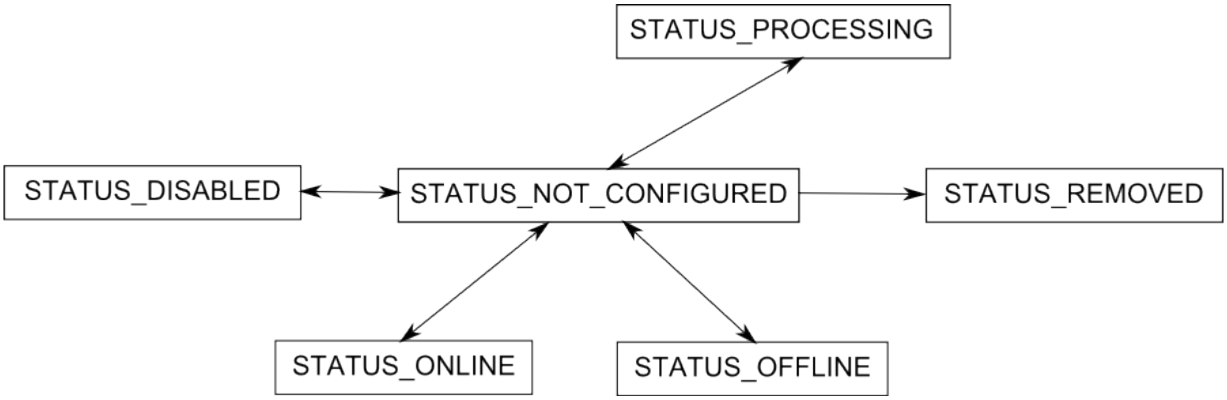

Illustration 11

The possible transitions are summarized on Illustration 11.

5.4 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.PROPERTY_UID – mandatory service property. The property value is the device function unique identifier. The value type is java.lang.String. To simplify the unique identifier generation, the property value must follow the rule:

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

  function UID – device function unique identifier

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

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

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

The DeviceFunction services are registered before the Device service. It's possible that DeviceFunction.PROPERTY_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.

5.4.1 Device Function Interface

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

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

- Java primitive type or corresponding reference type.
- java.lang.String
- Java Beans, but their properties must use those rules. Java Beans are defined in JavaBeans specification [3]. If possible, it should implement java.lang.Comparable to simplify the data comparison.
- java.util.Map instances. The map keys can be any reference type of Java primitive types or java.lang.String. The values must use those rules.
- Arrays of defined types.

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

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

5.4.2 Device Function Operations

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

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

- `DeviceFunction.META_INFO_DESCRIPTION` – Specifies a user readable description of the operation. It's an optional property. The property value type is `java.lang.String`.
- `DeviceFunction.META_INFO_OPERATION_ARG_OUT` – Specifies the operation output argument metadata. If the operation doesn't have return value, the property is missing. The value type is `java.util.Map`. The keys of the map value can be one of:
  - `DeviceFunction.META_INFO_DESCRIPTION` – Specifies a user readable description of the operation output argument. There are no additional property limitations.
  - `DeviceFunction.META_INFO_UNIT` – Specifies the measurement unit of the operation output argument as it's defined in Device Function Properties. There are no additional property limitations.
  - `DeviceFunction.META_INFO_MIN_VALUE` – Specifies the operation output argument minimum value. There are no additional property limitations.

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

  The input argument prefix must be used in the form:

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

  <argument_index> - the input argument index.

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

The operation arguments must follow the general type rules.

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

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

- DeviceFunction.META_INFO_PROPERTY_ACCESS – Specifies the access to the device property. It's a bitmap of java.lang.Integer type. The bitmap can be any combination of:

    - `DeviceFunction.META_INFO_PROPERTY_ACCESS_READABLE` – Marks the property as a readable. device function must provide a getter method for this property according to JavaBeans specification [3]. device function operations must not be overridden by this getter method.

    - `DeviceFunction.META_INFO_PROPERTY_ACCESS_WRITABLE` – Marks the property as writable. device function must provide a setter method for this property according to JavaBeans specification [3]. device function operations must not be overridden by this setter method.

    - `DeviceFunction.META_INFO_PROPERTY_ACCESS_EVENTABLE` – Marks the property as eventable. device function must not provide special methods because of this access type. `DeviceFunctionEvent` is sent on property change. Note that the event can be sent when there is no value change.

- `DeviceFunction.META_INFO_DESCRIPTION` – Specifies a user readable description of the property or the operation argument. It's an optional property. The property value type is `java.lang.String`.

- `DeviceFunction.META_INFO_UNIT` – Specifies the property or the operation argument value unit. The property value is `java.lang.String` type. These rules must be applied to unify the representation:

    - SI units (The International System of Units) must be used where it's applicable.

    - The unit must use Unicode symbols normalized with NFKD (Compatibility Decomposition) normalization form [4].

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

- `DeviceFunction.META_INFO_MIN_VALUE` – Specifies the property or the operation argument minimum value. The value type depends on the property or argument type.

- `DeviceFunction.META_INFO_MAX_VALUE` – Specifies the property or the operation argument maximum value. The value type depends on the property or argument type.

- `DeviceFunction.META_INFO_RESOLUTION` – Specifies the resolution value of a specific range. The value type depends on the property or orgument type. For example, the resolution in [0, 100] can be 10. That's the difference between two values in series.

- `DeviceFunction.META_INFO_VALUES` – Specifies the property or the operation argument possible values. The value type is `java.util.Map`, where the keys are the possible values and the values are their string representation.

5.4.4 Device Function Property Event

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

- All event source device properties.

- `DeviceFunctionEvent.PROPERTY_DEVICE_FUNCTION` – the event source function.

- `DeviceFunctionEvent.PROPERTY_DEVICE_FUNCTION_PROPERTY_NAME` – the property name.

- `DeviceFunctionEvent.PROPERTY_DEVICE_FUNCTION_PROPERTY_VALUE` – the property value.

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

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

5.5 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.

5.5.1 BinaryControl Device Function

`BinaryControl` device function provides a binary control support. The function state is accessible with `getState()` getter and `setState(BinaryData)` 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. The property eventing must follow the definition in Device Function Property Event.

`BinaryData` data structure is used to provide information about the function state. That data object contains the boolean value, the value collecting time i.e. timestamp and additional metadata. The timestamp is the difference between the value collecting time and midnight, January 1, 1970 UTC. It's measured in milliseconds. If possible, the value should be provided by the device, otherwise the device driver can generate that info. `java.lang.Long.MIN_VALUE` value means no timestamp. The immutable `BinaryData.timestamp` field is accessible with `BinaryData.getTimestamp()` getter.

The function class diagram is depicted on Illustration 12. The next code snippet sets to `true` all `BinaryControl` functions.

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

5.5.2 BinarySensor Device Function

`BinarySensor` device function provides binary sensor monitoring. It reports its state when an important event is available. The state is accessible with `BinarySensor.getState()` getter. There are no operations. The property eventing must follow the definition in Device Function Property Event.

BinarySensor and BinaryControl are using the same BinaryData data structure to provide information about the state. For more details see the definition in BinaryControl Device Function. The function class diagram is depicted on Illustration 12.

5.5.3 MultiLevelControl Device Function

MultiLevelControl device function provides multi-level control support. The function level is accessible with MultiLevelControl.getLevel() getter and MultiLevelControl.setLevel(MultiLevelData) setter. The property eventing must follow the definition in Device Function Property Event.

MultiLevelData data structure is used to provide information about the function level. That data object contains the BigDecimal value, the value collecting time i.e. timestamp, measurement unit and additional metadata. The timestamp is the difference between the value collecting time and midnight, January 1, 1970 UTC. It's measured in milliseconds. If possible, the value should be provided by the device, otherwise the device driver can generate that info. java.lang.Long.MIN_VALUE value means no timestamp. The immutable MultiLevelData.timestamp field is accessible with MultiLevelData.getTimestamp() getter. The measurement unit is used as it's defined in Device Function Properties. The immutable MultiLevelData.unit field is accessible with MultiLevelData.getUnit() getter.

The function class diagram is depicted on Illustration 12.

5.5.4 MultiLevelSensor Device Function

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

MultiLevelSensor and MultiLevelControl are using the same MultiLevelData 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 12.

5.5.5 Meter Device Function

Meter device function can measure metering information. The function provides three properties:

- PROPERTY_CURRENT – accessible with getCurrent() getter. The property contains the current consumption.
- PROPERTY_TOTAL – property accessible with getTotal() getter. The property contains the total consumption. It has been measured since the last call of resetTotal() or device initial run.
- PROPERTY_FLOW – property accessible with getFlow() getter. The property value specifies the meter flow type. For example, it can be metering consumption or generation flow.

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

Meter device function is using the same MultiLevelData 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 12.

5.5.6 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.

AlarmData data structure is used to provide information about the available alarm. That data object contains the alarm type, severity, the alarm collecting time i.e. timestamp and additional metadata. The timestamp is the difference between the value collecting time and midnight, January 1, 1970 UTC. It's measured in milliseconds. If possible, the value should be provided by the device, otherwise the device driver can generate that info. `java.lang.Long.MIN_VALUE` value means no timestamp. The immutable `MultiLevelData.timestamp` field is accessible with `AlarmData.getTimestamp()` getter.

The function class diagram is depicted on Illustration 12.

5.5.7 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.

`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, key name, the event collecting time i.e. timestamp and additional metadata. The timestamp is the difference between the value collecting time and midnight, January 1, 1970 UTC. It's measured in milliseconds. If possible, the value should be provided by the device, otherwise the device driver can generate that info. `java.lang.Long.MIN_VALUE` value means no timestamp. The immutable `KeypadData.timestamp` field is accessible with `KeypadData.getTimestamp()` getter. 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 12.

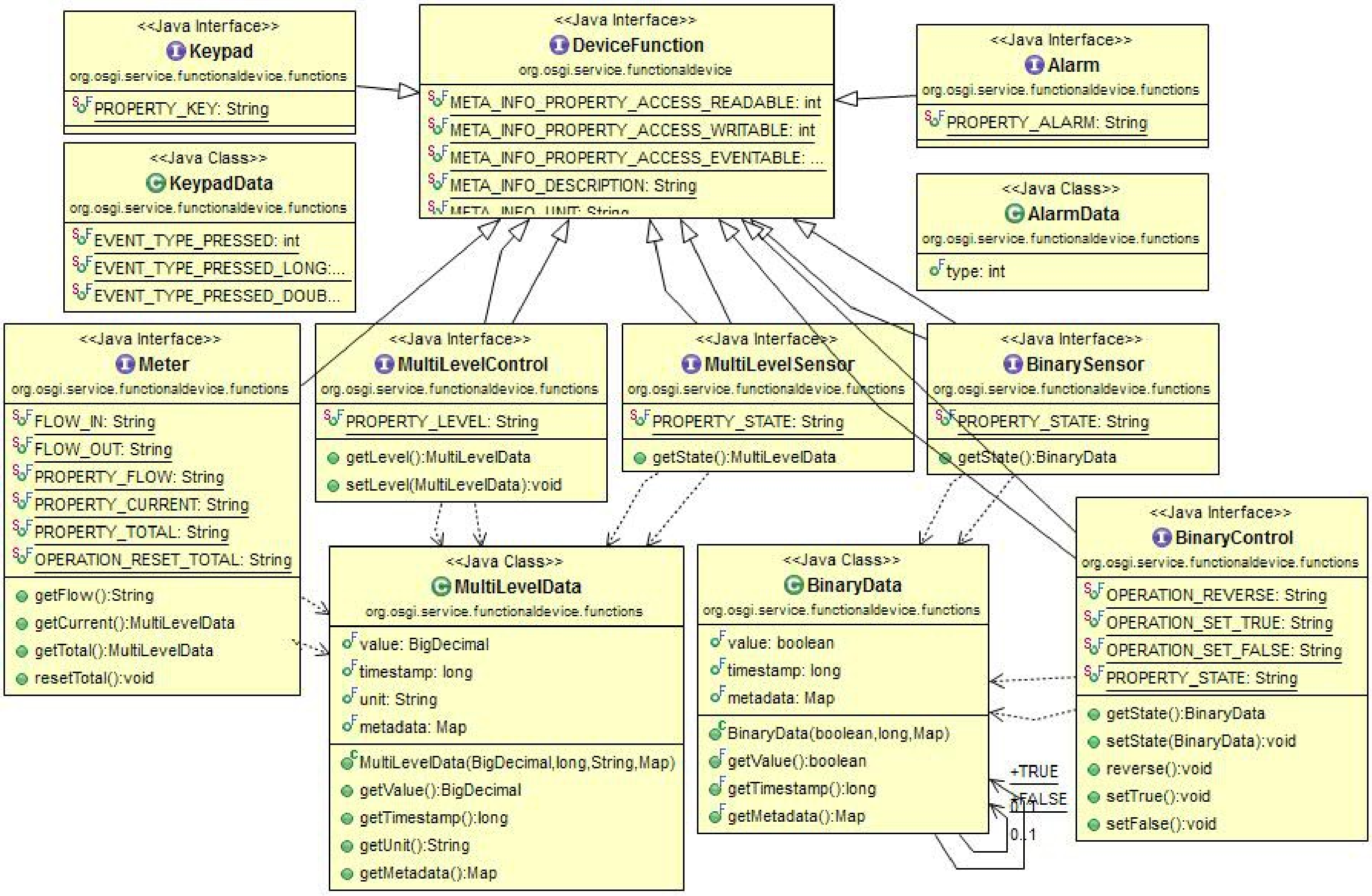
<<Java Interface>>
Keypad
org.osgi.service.functionaldevice.functions
PROPERTY_KEY: String
<<Java Class>>
KeypadData
org.osgi.service.functionaldevice.functions
EVENT_TYPE_PRESSED: int
EVENT_TYPE_PRESSED_LONG:...
EVENT_TYPE_PRESSED_DOUB...
<<Java Interface>>
DeviceFunction
org.osgi.service.functionaldevice
META_INFO_PROPERTY_ACCESS_READABLE: int
META_INFO_PROPERTY_ACCESS_WRITABLE: int
META_INFO_PROPERTY_ACCESS_EVENTABLE: ...
META_INFO_DESCRIPTION: String
META_INFO_UNIT: String
<<Java Interface>>
Alarm
org.osgi.service.functionaldevice.functions
PROPERTY_ALARM: String
<<Java Class>>
AlarmData
org.osgi.service.functionaldevice.functions
type: int
<<Java Interface>>
Meter
org.osgi.service.functionaldevice.functions
FLOW_IN: String
FLOW_OUT: String
PROPERTY_FLOW: String
PROPERTY_CURRENT: String
PROPERTY_TOTAL: String
OPERATION_RESET_TOTAL: String
getFlow():String
getCurrent():MultiLevelData
getTotal():MultiLevelData
resetTotal():void
<<Java Interface>>
MultiLevelControl
org.osgi.service.functionaldevice.functions
PROPERTY_LEVEL: String
getLevel():MultiLevelData
setLevel(MultiLevelData):void
<<Java Interface>>
MultiLevelSensor
org.osgi.service.functionaldevice.functions
PROPERTY_STATE: String
getState():MultiLevelData
<<Java Interface>>
BinarySensor
org.osgi.service.functionaldevice.functions
PROPERTY_STATE: String
getState():BinaryData
<<Java Class>>
MultiLevelData
org.osgi.service.functionaldevice.functions
value: BigDecimal
timestamp: long
unit: String
metadata: Map
MultiLevelData(BigDecimal,long,String,Map)
getValue():BigDecimal
getTimestamp():long
getUnit():String
getMetadata():Map
<<Java Class>>
BinaryData
org.osgi.service.functionaldevice.functions
value: boolean
timestamp: long
metadata: Map
BinaryData(boolean,long,Map)
getValue():boolean
getTimestamp():long
getMetadata():Map
+TRUE
0..1
+FALSE
0..1
<<Java Interface>>
BinaryControl
org.osgi.service.functionaldevice.functions
OPERATION_REVERSE: String
OPERATION_SET_TRUE: String
OPERATION_SET_FALSE: String
PROPERTY_STATE: String
getState():BinaryData
setState(BinaryData):void
reverse():void
setTrue():void
setFalse():void

Illustration 12

6 Data Transfer Objects

TODO: Do we need those objects?

7 Javadoc

OSGi Javadoc

9/9/13 5:10 PM

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

Package org.osgi.service.functionaldevice

Functional Device Package Version 1.0.

See:

[Description](#)

Interface Summary		Page
Device	Represents the functional device in the OSGi service registry.	34
DeviceFunction	Device Function service provides specific device operations and properties.	47

Class Summary		Page
DeviceFunctionEvent	Asynchronous event, which marks a Device Function property value modification.	55
DevicePermission	A bundle's authority to perform specific privileged administrative operations on the devices.	59

Exception Summary		Page
DeviceException	`DeviceExcpetion` is a special `IOException`, which is thrown to indicate that there is a device operation fail.	44

Package org.osgi.service.functionaldevice Description

Functional Device Package Version 1.0.

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

Example import for consumers using the API in this package:

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

Example import for providers implementing the API in this package:

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

Interface Device

org.osgi.service.functionaldevice

```
public interface Device
```

Represents the functional device in the OSGi service registry. Note that `Device` services are registered last. Before their registration, there is `DeviceFunction` 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.device.Constants.DEVICE_CATEGORY` service property.	35
`String`	**`PROPERTY_COMMUNICATION`** The service property value contains the device communication possibility.	36
`String`	**`PROPERTY_DESCRIPTION`** The service property value contains the device description.	38
`String`	**`PROPERTY_FIRMWARE_VENDOR`** The service property value contains the device firmware vendor.	37
`String`	**`PROPERTY_FIRMWARE_VERSION`** The service property value contains the device firmware version.	37
`String`	**`PROPERTY_HARDWARE_VENDOR`** The service property value contains the device hardware vendor.	37
`String`	**`PROPERTY_HARDWARE_VERSION`** The service property value contains the device hardware version.	37
`String`	**`PROPERTY_MODEL`** The service property value contains the device model.	38
`String`	**`PROPERTY_NAME`** The service property value contains the device name.	36
`String`	**`PROPERTY_REFERENCE_UIDS`** The service property value contains the reference device unique identifiers.	36
`String`	**`PROPERTY_SERIAL_NUMBER`** The service property value contains the device serial number.	38
`String`	**`PROPERTY_STATUS`** The service property value contains the device status.	36
`String`	**`PROPERTY_STATUS_DETAIL`** The service property value contains the device status detail.	37
`String`	**`PROPERTY_TYPES`** The service property value contains the device types like DVD, TV etc.	38
`String`	**`PROPERTY_UID`** The service property value contains the device unique identifier.	36
`int`	**`STATUS_DETAIL_CONFIGURATION_NOT_APPLIED`** Device status detail indicates that the device configuration is not applied.	39
`int`	**`STATUS_DETAIL_CONNECTING`** Device status detail indicates that the device is currently connecting to the network.	39
`int`	**`STATUS_DETAIL_DEVICE_BROKEN`** Device status detail indicates that the device is broken.	40

int	**STATUS_DETAIL_DEVICE_COMMUNICATION_ERROR** Device status detail indicates that the device communication is problematic.	40
int	**STATUS_DETAIL_DEVICE_DATA_INSUFFICIENT** Device status detail indicates that the device doesn't provide enough information and cannot be determined.	40
int	**STATUS_DETAIL_DEVICE_NOT_ACCESSIBLE** Device status detail indicates that the device is not accessible and further communication is not possible.	40
int	**STATUS_DETAIL_ERROR_APPLYING_CONFIGURATION** Device status detail indicates that the device cannot be configured.	40
int	**STATUS_DETAIL_IN_DUTY_CYCLE** Device status detail indicates that the device is in duty cycle.	40
int	**STATUS_DETAIL_INITIALIZING** Device status detail indicates that the device is currently in process of initialization.	39
int	**STATUS_DISABLED** Device status indicates that the device is currently disabled.	39
int	**STATUS_NOT_CONFIGURED** Device status indicates that the device is currently not configured.	39
int	**STATUS_NOT_INITIALIZED** Device status indicates that the device is currently not initialized.	39
int	**STATUS_OFFLINE** Device status indicates that the device is currently not available for operations.	38
int	**STATUS_ONLINE** Device status indicates that the device is currently available for operations.	38
int	**STATUS_PROCESSING** Device status indicates that the device is currently busy with an operation.	39
int	**STATUS_REMOVED** Device status indicates that the device is removed.	38
String	**TYPE_PERIPHERAL** Device type indicates that the device is peripheral.	40

Method Summary		**Page**
void	**disable**() Disables this device.	42
void	**enable**() Enables this device.	43
Object	**getProperty**(String propName) Returns the current value of the specified property.	41
void	**remove**() Removes this device.	42
void	**setProperties**(String[] propNames, Object[] propValues) Sets the given property names to the given property values.	41
void	**setProperty**(String propName, Object propValue) Sets the given property name to the given property value.	41

Field Detail

DEVICE_CATEGORY

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

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

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

PROPERTY_UID

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

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

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

UID - device unique identifier

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

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

---

PROPERTY_REFERENCE_UIDS

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

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

PROPERTY_COMMUNICATION

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

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

PROPERTY_NAME

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

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

PROPERTY_STATUS

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

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

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

PROPERTY_STATUS_DETAIL

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

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

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

PROPERTY_HARDWARE_VENDOR

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

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

PROPERTY_HARDWARE_VERSION

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

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

PROPERTY_FIRMWARE_VENDOR

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

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

PROPERTY_FIRMWARE_VERSION

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

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

PROPERTY_TYPES

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

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

PROPERTY_MODEL

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

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

PROPERTY_SERIAL_NUMBER

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

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

PROPERTY_DESCRIPTION

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

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

STATUS_REMOVED

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

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

STATUS_OFFLINE

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

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

STATUS_ONLINE

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

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

STATUS_PROCESSING

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

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

STATUS_DISABLED

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

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

STATUS_NOT_INITIALIZED

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

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

STATUS_NOT_CONFIGURED

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

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

STATUS_DETAIL_CONNECTING

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

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

STATUS_DETAIL_INITIALIZING

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

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

STATUS_DETAIL_CONFIGURATION_NOT_APPLIED

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

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

STATUS_DETAIL_DEVICE_BROKEN

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

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

STATUS_DETAIL_DEVICE_COMMUNICATION_ERROR

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

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

STATUS_DETAIL_DEVICE_DATA_INSUFFICIENT

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

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

STATUS_DETAIL_DEVICE_NOT_ACCESSIBLE

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

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

STATUS_DETAIL_ERROR_APPLYING_CONFIGURATION

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

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

STATUS_DETAIL_IN_DUTY_CYCLE

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

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

TYPE_PERIPHERAL

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

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

Method Detail

getProperty

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

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

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

Parameters:

`propName` - The property name.

Returns:

The property value

Throws:

`IllegalArgumentException` - If the property name cannot be mapped to value.

setProperty

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

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

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

Parameters:

`propName` - The property name.

`propValue` - The property value.

Throws:

`DeviceException` - If an operation error is available.

`IllegalArgumentException` - If the property name or value aren't correct.

`UnsupportedOperationException` - If the operation is not supported over this device.

`SecurityException` - If the caller does not have the appropriate `FunctionalDevicePermission[this device, DevicePermission.ACTION_PROPERTY]` and the Java Runtime Environment supports permissions.

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

setProperties

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

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

Parameters:

`propNames` - The property names.
`propValues` - The property values.

Throws:

`DeviceException` - If an operation error is available.
`IllegalArgumentException` - If the property values or names aren't correct.
`UnsupportedOperationException` - If the operation is not supported over this device.
`SecurityException` - If the caller does not have the appropriate `FunctionalDevicePermission[this device, DevicePermission.ACTION_PROPERTY]` 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.

disable

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

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

Throws:

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

enable

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

Enables this device. The device is available for operations.

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_ENABLE]` and the Java Runtime Environment supports permissions.

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

Class DeviceException

org.osgi.service.functionaldevice

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

All Implemented Interfaces:

Serializable

```
public class DeviceException
extends IOException
```

`DeviceExcpetion` 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.

Field Summary		Page
static int	`CODE_COMMUNICATION_ERROR` An exception code indicates that there is an error in the communication.	45
static int	`CODE_DEVICE_NOT_INITIALIZED` An exception code indicates that the device is not initialized.	45
static int	`CODE_NO_DATA` An exception code indicates that the requested value is currently not available.	45
static int	`CODE_TIMEOUT` An exception code indicates that the response is not produced within a given timeout.	45
static int	`CODE_UNKNOWN` An exception code indicates that the error is unknown.	44

Constructor Summary	Page
`DeviceException()`	45

Method Summary		Page
Throwable	`getCause()` Returns the cause for this throwable or `null` if the cause is missing.	45
int	`getCode()` Returns the exception error code.	45

Field Detail

CODE_UNKNOWN

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

An exception code indicates that the error is unknown.

CODE_COMMUNICATION_ERROR

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

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

CODE_TIMEOUT

`public static final int CODE_TIMEOUT = 3`

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

CODE_DEVICE_NOT_INITIALIZED

`public static final int CODE_DEVICE_NOT_INITIALIZED = 4`

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

CODE_NO_DATA

`public static final int CODE_NO_DATA = 5`

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

Constructor Detail

DeviceException

`public DeviceException()`

Method Detail

getCode

`public int getCode()`

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

Returns:
An exception code.

getCause

`public Throwable getCause()`

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

Overrides:
`getCause` in class `Throwable`

Returns:

An throwable cause.

Interface DeviceFunction

org.osgi.service.functionaldevice

All Known Subinterfaces:

Alarm, BinaryControl, BinarySensor, 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:

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

The `DeviceFunction` services are registered before the `Device` services. It's possible that `PROPERTY_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(), 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 Device Function services can be accesses, but it's not clear which are the Device Function classes.

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

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

The accessor methods must be defined according JavaBeans specification.

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

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

- Java primitive type or corresponding reference type.
- `java.lang.String`.
- `Beans`, but the beans properties must use those rules. Java Beans are defined in JavaBeans specification.
- `java.util.Map`s. The keys can be any reference type of Java primitive types or `java.lang.String`. The values must use those rules.
- Arrays of defined types.

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

- `META_INFO_DESCRIPTION`
- `META_INFO_UNIT`
- `META_INFO_MIN_VALUE`
- `META_INFO_MAX_VALUE`
- `META_INFO_RESOLUTION`
- `META_INFO_VALUES`

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

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

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

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

Field Summary		***Page***
`String`	**`META_INFO_DESCRIPTION`** Meta data key, which value represents the Device Function property, the operation argument or operation description.	*50*
`String`	**`META_INFO_MAX_VALUE`** Meta data key, which value represents the Device Function property or the operation argument maximum value.	*51*
`String`	**`META_INFO_MIN_VALUE`** Meta data key, which value represents the Device Function property or the operation argument minimum value.	*51*

String	**META_INFO_OPERATION_ARG_OUT** Meta data key, which value represents the operation output argument metadata.	52
String	**META_INFO_OPERATION_ARGS_IN_PREFIX** Meta data key prefix, which key value represents the operation input argument metadata.	51
String	**META_INFO_PROPERTY_ACCESS** Meta data key, which value represents the access to the Device Function property.	50
int	**META_INFO_PROPERTY_ACCESS_EVENTABLE** Marks the eventable Device Function properties.	50
int	**META_INFO_PROPERTY_ACCESS_READABLE** Marks the readable Device Function properties.	49
int	**META_INFO_PROPERTY_ACCESS_WRITABLE** Marks the writable Device Function properties.	50
String	**META_INFO_RESOLUTION** Meta data key, which value represents the resolution value of specific range of the Device Function property or the operation argument.	51
String	**META_INFO_UNIT** Meta data key, which value represents the Device Function property or the operation argument unit.	50
String	**META_INFO_VALUES** Meta data key, which value represents the Device Function property or the operation argument possible values.	51
String	**PROPERTY_DESCRIPTION** The service property value contains the device function description.	53
String	**PROPERTY_DEVICE_UID** The service property value contains the function device unique identifier.	53
String	**PROPERTY_OPERATION_NAMES** The service property value contains the device function operation names.	53
String	**PROPERTY_PROPERTY_NAMES** The service property value contains the device function property names.	53
String	**PROPERTY_REFERENCE_UIDS** The service property value contains the reference device function unique identifiers.	53
String	**PROPERTY_UID** The service property value contains the device function unique identifier.	52

Method Summary		**Page**
Map	**getOperationMetaData**(String operationName) Provides meta data about the given function operation.	54
Map	**getPropertyMetaData**(String propertyName) Provides meta data about the given function property.	53

Field Detail

META_INFO_PROPERTY_ACCESS_READABLE

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

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

META_INFO_PROPERTY_ACCESS_WRITABLE

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

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

META_INFO_PROPERTY_ACCESS_EVENTABLE

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

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

META_INFO_DESCRIPTION

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

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

See Also:

getPropertyMetaData(String), getOperationMetaData(String)

META_INFO_UNIT

```
public static final String META_INFO_UNIT = "unit"
```

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

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

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

See Also:

getPropertyMetaData(String)

META_INFO_PROPERTY_ACCESS

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

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

- META_INFO_PROPERTY_ACCESS_READABLE
- META_INFO_PROPERTY_ACCESS_WRITABLE
- META_INFO_PROPERTY_ACCESS_EVENTABLE

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

See Also:
`getPropertyMetaData(String)`

META_INFO_MIN_VALUE

`public static final String` **`META_INFO_MIN_VALUE`** `= "min.value"`

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

See Also:
`getPropertyMetaData(String)`

META_INFO_MAX_VALUE

`public static final String` **`META_INFO_MAX_VALUE`** `= "max.value"`

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

See Also:
`getPropertyMetaData(String)`

META_INFO_RESOLUTION

`public static final String` **`META_INFO_RESOLUTION`** `= "resolution"`

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

See Also:
`getPropertyMetaData(String)`

META_INFO_VALUES

`public static final String` **`META_INFO_VALUES`** `= "values"`

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

See Also:
`getPropertyMetaData(String)`

META_INFO_OPERATION_ARGS_IN_PREFIX

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

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

- META_INFO_DESCRIPTION
- META_INFO_UNIT
- META_INFO_MIN_VALUE
- META_INFO_MAX_VALUE
- META_INFO_RESOLUTION
- META_INFO_VALUES
- custom key

The prefix must be used in the form:

operation input argument name ::= value of `META_INFO_OPERATION_ARGS_IN_PREFIX`argument-index

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

See Also:
`getOperationMetaData(String)`

META_INFO_OPERATION_ARG_OUT

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

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

- META_INFO_DESCRIPTION
- META_INFO_UNIT
- META_INFO_MIN_VALUE
- META_INFO_MAX_VALUE
- META_INFO_RESOLUTION
- META_INFO_VALUES
- custom key

See Also:
`getOperationMetaData(String)`

PROPERTY_UID

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

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

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

function UID - device function unique identifier

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

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

PROPERTY_DEVICE_UID

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

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

PROPERTY_REFERENCE_UIDS

```
public static final String PROPERTY_REFERENCE_UIDS = "functional.device.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[]`. The property value cannot be set. It can be used to represent different relationships between the device functions.

PROPERTY_DESCRIPTION

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

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

PROPERTY_OPERATION_NAMES

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

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

PROPERTY_PROPERTY_NAMES

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

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

Method Detail

getPropertyMetaData

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

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

- META_INFO_DESCRIPTION
- META_INFO_PROPERTY_ACCESS
- META_INFO_UNIT
- META_INFO_MIN_VALUE
- META_INFO_MAX_VALUE

- META_INFO_RESOLUTION
- META_INFO_VALUES
- custom key

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

Parameters:

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

Returns:

The property meta data for the given property name. `null` if the property meta data is not supported.

Throws:

`IllegalArgumentException` - If the function property with the specified name is not supported.

getOperationMetaData

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

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

- META_INFO_DESCRIPTION
- META_INFO_OPERATION_ARG_OUT
- Different input arguments with prefix META_INFO_OPERATION_ARGS_IN_PREFIX
- custom key

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

Parameters:

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

Returns:

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

Throws:

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

Class DeviceFunctionEvent

org.osgi.service.functionaldevice

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

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

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

- `PROPERTY_DEVICE_FUNCTION_UID` - the event source function unique identifier.
- `PROPERTY_DEVICE_FUNCTION_PROPERTY_NAME` - the property name.
- `PROPERTY_DEVICE_FUNCTION_PROPERTY_VALUE` - the property value.

Field Summary		Page
static String	**EVENT_CLASS** Represents the event class.	56
static String	**EVENT_PACKAGE** Represents the event package.	56
static String	**PROPERTY_DEVICE_FUNCTION_PROPERTY_NAME** Represents an event property key for the Device Function property name.	56
static String	**PROPERTY_DEVICE_FUNCTION_PROPERTY_VALUE** Represents an event property key for the Device Function property value.	56
static String	**PROPERTY_DEVICE_FUNCTION_UID** Represents an event property key for Device Function UID.	56
static String	**TOPIC_PROPERTY_CHANGED** Represents the event topic for the Device Function property changed.	56

Constructor Summary	Page
DeviceFunctionEvent(String topic, Dictionary properties) Constructs a new event with the specified topic and properties.	57
DeviceFunctionEvent(String topic, Map properties) Constructs a new event with the specified topic and properties.	57

Method Summary		Page
String	**getFunctionUID**() Returns the property value change source function identifier.	57
String	**getPropertyName**() Returns the property name.	57
Object	**getPropertyValue**() Returns the property value.	57

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

```
toString
```

Field Detail

EVENT_PACKAGE

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

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

EVENT_CLASS

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

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

TOPIC_PROPERTY_CHANGED

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

Represents the event topic for the Device Function property changed.

PROPERTY_DEVICE_FUNCTION_UID

```
public static final String PROPERTY_DEVICE_FUNCTION_UID = "device.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_DEVICE_FUNCTION_PROPERTY_NAME

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

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

PROPERTY_DEVICE_FUNCTION_PROPERTY_VALUE

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

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

Constructor Detail

DeviceFunctionEvent

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

Constructs a new event with the specified topic and properties.

Parameters:

`topic` - The event topic.
`properties` - The event properties.

DeviceFunctionEvent

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

Constructs a new event with the specified topic and properties.

Parameters:

`topic` - The event topic.
`properties` - The event properties.

Method Detail

getFunctionUID

```
public String getFunctionUID()
```

Returns the property value change source function identifier. The value is same as the value of `PROPERTY_DEVICE_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_DEVICE_FUNCTION_PROPERTY_NAME`.

Returns:

The property name.

---

getPropertyValue

```
public Object getPropertyValue()
```

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

Returns:

The property value.

Class DevicePermission

org.osgi.service.functionaldevice

```
java.lang.Object
  └java.security.Permission
      └java.security.BasicPermission
          └org.osgi.service.functionaldevice.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_ENABLE	Device.enable()
ACTION_DISABLE	Device.disable()

ACTION_PROPERTY	Device.setProperty(String, Object)
	Device.setProperties(String[], Object[])

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

Field Summary		Page
static String	**ACTION_DISABLE** A permission action to disable the device.	*60*
static String	**ACTION_ENABLE** A permission action to enable the device.	*60*
static String	**ACTION_PROPERTY** A permission action to modify the device properties.	*60*
static String	**ACTION_REMOVE** A permission action to remove the device.	*60*

Constructor Summary	Page
DevicePermission(String filter, String actions) Creates a new `FunctionalDevicePermission` with the given filter and actions.	*60*
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: - represents the same filter and actions - represents the same device and actions	*61*

String	getActions() Returns the canonical string representation of the actions.	62
int	hashCode() Returns the hash code value for this object.	61
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.	62

Field Detail

ACTION_ENABLE

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

A permission action to enable the device.

ACTION_DISABLE

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

A permission action to disable the device.

ACTION_PROPERTY

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

A permission action to modify the device properties.

ACTION_REMOVE

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

A permission action to remove the device.

Constructor Detail

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: (abstract.device.hardware.vendor=acme)

An action list example: property, remove

Parameters:

filter - A filter expression that can use any device service property. The filter attribute names are processed in a case insensitive manner. A special value of "*" can be used to match akk devices.

actions - A comma-separated list of ACTION_DISABLE, ACTION_ENABLE ACTION_PROPERTY and ACTION_REMOVE. Any combinations are allowed.

Throws:

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

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_DISABLE, ACTION_ENABLE ACTION_PROPERTY and ACTION_REMOVE. Any combinations are allowed.

Method Detail

equals

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

Two FunctionalDevicePermission instances are equal if:

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

Overrides:

equals in class BasicPermission

Parameters:

obj - The object being compared for equality with this object.

Returns:

true if two permissions are equal, false otherwise.

---

hashCode

```
public int hashCode()
```

Returns the hash code value for this object.

Overrides:

hashCode in class BasicPermission

Returns:

Hash code value for this object.

---

getActions

`public String getActions()`

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

Overrides:
`getActions` in class `BasicPermission`

Returns:
The canonical string representation of the actions.

implies

`public boolean implies(Permission p)`

Determines if the specified permission is implied by this object. The method will throw an exception if the specified permission was not constructed by `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.

Package org.osgi.service.functionaldevice.functions

Functional Device Functions 1.0.

See:

[Description](#)

Interface Summary		*Page*
Alarm	`Alarm` Device Function provides alarm sensor support.	*64*
BinaryControl	`BinaryControl` Device Function provides a binary control support.	*68*
BinarySensor	`BinarySensor` Device Function provides binary sensor monitoring.	*75*
Keypad	`Keypad` Device Function provides support for keypad control.	*77*
Meter	`Meter` Device Function can measure metering information.	*82*
MultiLevelControl	`MultiLevelControl` Device Function provides multi-level control support.	*86*
MultiLevelSensor	`MultiLevelSensor` Device Function provides multi-level sensor monitoring.	*91*

Class Summary		*Page*
AlarmData	Device Function alarm data.	*65*
BinaryData	Device Function binary data wrapper.	*72*
KeypadData	Represents a keypad event data that is collected when a change with some key from device keypad has occurred.	*78*
MultiLevelData	Device Function multi-level data wrapper.	*88*

Package org.osgi.service.functionaldevice.functions Description

Functional Device Functions 1.0.

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

Example import for consumers using the API in this package:

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

Example import for providers implementing the API in this package:

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

Interface Alarm

org.osgi.service.functionaldevice.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.	*64*

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

`META_INFO_DESCRIPTION`, `META_INFO_MAX_VALUE`, `META_INFO_MIN_VALUE`, `META_INFO_OPERATION_ARG_OUT`, `META_INFO_OPERATION_ARGS_IN_PREFIX`, `META_INFO_PROPERTY_ACCESS`, `META_INFO_PROPERTY_ACCESS_EVENTABLE`, `META_INFO_PROPERTY_ACCESS_READABLE`, `META_INFO_PROPERTY_ACCESS_WRITABLE`, `META_INFO_RESOLUTION`, `META_INFO_UNIT`, `META_INFO_VALUES`, `PROPERTY_DESCRIPTION`, `PROPERTY_DEVICE_UID`, `PROPERTY_OPERATION_NAMES`, `PROPERTY_PROPERTY_NAMES`, `PROPERTY_REFERENCE_UIDS`, `PROPERTY_UID`

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

`getOperationMetaData`, `getPropertyMetaData`

Field Detail

PROPERTY_ALARM

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

Specifies the alarm property name. The property is eventable.

See Also:

`AlarmData`

Class AlarmData

org.osgi.service.functionaldevice.functions

```
java.lang.Object
  └org.osgi.service.functionaldevice.functions.AlarmData
```

```
public class AlarmData
extends Object
```

Device Function alarm data. It cares about the alarm type, severity and additional metadata.

See Also:
Alarm

Field Summary		Page
Map	**metadata** Represents AlarmData metadata in an unmodifiable Map.	*66*
int	**severity** Represents the alarm severity.	*66*
long	**timestamp** Represents AlarmData timestamp.	*66*
int	**type** Represents the alarm type.	*65*

Constructor Summary	Page
AlarmData(int type, int severity, long timestamp, Map metadata) Constructs new AlarmData instance with the specified arguments.	*66*

Method Summary		Page
Map	**getMetadata**() Returns AlarmData metadata.	*67*
int	**getSeverity**() Returns the alarm severity.	*66*
long	**getTimestamp**() Returns AlarmData timestamp.	*67*
int	**getType**() Returns the alarm type.	*66*

Field Detail

type

```
public final int type
```

Represents the alarm type. The immutable field is accessible with getType() getter.

severity

```
public final int severity
```

Represents the alarm severity. The immutable field is accessible with `getSeverity()` getter.

---

timestamp

```
public final long timestamp
```

Represents `AlarmData` timestamp. The timestamp is the difference between the value collecting time and midnight, January 1, 1970 UTC. It's measured in milliseconds. If possible, the value should be provided by the device, otherwise the device driver can generate that info. `Long.MIN_VALUE` value means no timestamp. The immutable field is accessible with `getTimestamp()` getter.

---

metadata

```
public final Map metadata
```

Represents `AlarmData` metadata in an unmodifiable `Map`. The immutable field is accessible with `getMetadata()` getter.

Constructor Detail

AlarmData

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

Constructs new `AlarmData` instance with the specified arguments.

Parameters:

`type` - The alarm type.
`severity` - The alarm severity.
`metadata` - The alarm metadata.

Method Detail

getType

```
public int getType()
```

Returns the alarm type.

Returns:

The alarm type.

---

getSeverity

```
public int getSeverity()
```

Returns the alarm severity.

Returns:
The alarm severity.

---

getTimestamp

```
public long getTimestamp()
```

Returns `AlarmData` timestamp. The timestamp is the difference between the value collecting time and midnight, January 1, 1970 UTC. It's measured in milliseconds. If possible, the value should be provided by the device, otherwise the device driver can generate that info. `Long.MIN_VALUE` value means no timestamp.

Returns:
`AlarmData` timestamp.

---

getMetadata

```
public Map getMetadata()
```

Returns `AlarmData` metadata.

Returns:
`AlarmData` metadata.

Interface BinaryControl

org.osgi.service.functionaldevice.functions

All Superinterfaces:

DeviceFunction

```
public interface BinaryControl
extends DeviceFunction
```

`BinaryControl` Device Function provides a binary control support. The function state is accessible with `getState()` getter and `setState(BinaryData)` 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.

See Also:

`BinaryData`

Field Summary

Field Summary		Page
String	**OPERATION_REVERSE** Specifies the reverse operation name.	69
String	**OPERATION_SET_FALSE** Specifies the operation name, which sets the control state to `false` value.	69
String	**OPERATION_SET_TRUE** Specifies the operation name, which sets the control state to `true` value.	69
String	**PROPERTY_STATE** Specifies the state property name.	69

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

`META_INFO_DESCRIPTION`, `META_INFO_MAX_VALUE`, `META_INFO_MIN_VALUE`, `META_INFO_OPERATION_ARG_OUT`, `META_INFO_OPERATION_ARGS_IN_PREFIX`, `META_INFO_PROPERTY_ACCESS`, `META_INFO_PROPERTY_ACCESS_EVENTABLE`, `META_INFO_PROPERTY_ACCESS_READABLE`, `META_INFO_PROPERTY_ACCESS_WRITABLE`, `META_INFO_RESOLUTION`, `META_INFO_UNIT`, `META_INFO_VALUES`, `PROPERTY_DESCRIPTION`, `PROPERTY_DEVICE_UID`, `PROPERTY_OPERATION_NAMES`, `PROPERTY_PROPERTY_NAMES`, `PROPERTY_REFERENCE_UIDS`, `PROPERTY_UID`

Method Summary

Method Summary		Page
BinaryData	**getState**() Returns the current state of `BinaryControl`.	69
void	**reverse**() Reverses the `BinaryControl` state.	70
void	**setFalse**() Sets the `BinaryControl` state to `false` value.	70
void	**setState**(BinaryData data) Sets the state of `BinaryControl`.	70
void	**setTrue**() Sets the `BinaryControl` state to `true` value.	70

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

`getOperationMetaData`, `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_STATE

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

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

See Also:
`BinaryData`

Method Detail

getState

```
BinaryData getState()
             throws UnsupportedOperationException,
                    IllegalStateException,
                    DeviceException
```

Returns the current state of `BinaryControl`. It's a getter method for `PROPERTY_STATE` property.

Returns:
The current state of the Binary Sensor.

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:
`BinaryData`

setState

```
void setState(BinaryData data)
       throws UnsupportedOperationException,
              IllegalStateException,
              DeviceException
```

Sets the state of `BinaryControl`. It's a setter method for `PROPERTY_STATE` property.

Parameters:

`data` - The new `BinaryControl` 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:

`BinaryData`

reverse

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

Reverses the `BinaryControl` 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 `BinaryControl` 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 `BinaryControl` 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.

Class BinaryData

org.osgi.service.functionaldevice.functions

```
java.lang.Object
  └org.osgi.service.functionaldevice.functions.BinaryData
```

```
public class BinaryData
extends Object
```

Device Function binary data wrapper. It can contain a boolean value, timestamp and additional metadata.

See Also:

BinaryControl, BinarySensor

Field Summary		Page
static BinaryData	**FALSE** BinaryData instance represents false value.	73
Map	**metadata** Represents BinaryData metadata in an unmodifiable Map.	73
long	**timestamp** Represents BinaryData timestamp.	73
static BinaryData	**TRUE** BinaryData instance represents true value.	72
boolean	**value** Represents BinaryData value.	73

Constructor Summary	Page
BinaryData(boolean value, long timestamp, Map metadata) Constructs new BinaryData instance with the specified arguments.	73

Method Summary		Page
Map	**getMetadata**() Returns BinaryData metadata.	74
long	**getTimestamp**() Returns BinaryData timestamp.	74
boolean	**getValue**() Returns BinaryData value.	73

Field Detail

TRUE

```
public static final BinaryData TRUE
```

BinaryData instance represents true value.

FALSE

```
public static final BinaryData FALSE
```

`BinaryData` instance represents `false` value.

---

value

```
public final boolean value
```

Represents `BinaryData` value. The immutable field is accessible with `getValue()` getter.

---

timestamp

```
public final long timestamp
```

Represents `BinaryData` timestamp. The timestamp is the difference between the value collecting time and midnight, January 1, 1970 UTC. It's measured in milliseconds. If possible, the value should be provided by the device, otherwise the device driver can generate that info. `Long.MIN_VALUE` value means no timestamp. The immutable field is accessible with `getTimestamp()` getter.

---

metadata

```
public final Map metadata
```

Represents `BinaryData` metadata in an unmodifiable `Map`. The immutable field is accessible with `getMetadata()` getter.

Constructor Detail

BinaryData

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

Constructs new `BinaryData` instance with the specified arguments.

Parameters:

`value` - The binary value.
`timestamp` - The value timestamp.
`metadata` - The value metadata.

Method Detail

getValue

```
public final boolean getValue()
```

Returns `BinaryData` value.

Returns:

`BinaryData` value.

getTimestamp

```
public final long getTimestamp()
```

Returns `BinaryData` timestamp. The timestamp is the difference between the value collecting time and midnight, January 1, 1970 UTC. It's measured in milliseconds. If possible, the value should be provided by the device, otherwise the device driver can generate that info. `Long.MIN_VALUE` value means no timestamp.

Returns:

`BinaryData` timestamp.

getMetadata

```
public final Map getMetadata()
```

Returns `BinaryData` metadata.

Returns:

`BinaryData` metadata.

Interface BinarySensor

org.osgi.service.functionaldevice.functions

All Superinterfaces:

DeviceFunction

```
public interface BinarySensor
extends DeviceFunction
```

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

See Also:

`BinaryData`

Field Summary		Page
`String`	**`PROPERTY_STATE`** Specifies the state property name.	75

Fields inherited from interface org.osgi.service.functionaldevice.DeviceFunction
`META_INFO_DESCRIPTION`, `META_INFO_MAX_VALUE`, `META_INFO_MIN_VALUE`, `META_INFO_OPERATION_ARG_OUT`, `META_INFO_OPERATION_ARGS_IN_PREFIX`, `META_INFO_PROPERTY_ACCESS`, `META_INFO_PROPERTY_ACCESS_EVENTABLE`, `META_INFO_PROPERTY_ACCESS_READABLE`, `META_INFO_PROPERTY_ACCESS_WRITABLE`, `META_INFO_RESOLUTION`, `META_INFO_UNIT`, `META_INFO_VALUES`, `PROPERTY_DESCRIPTION`, `PROPERTY_DEVICE_UID`, `PROPERTY_OPERATION_NAMES`, `PROPERTY_PROPERTY_NAMES`, `PROPERTY_REFERENCE_UIDS`, `PROPERTY_UID`

Method Summary		Page
`BinaryData`	**`getState`**`()` Returns the `BinarySensor`current state.	75

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

Field Detail

PROPERTY_STATE

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

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

Method Detail

getState

```
BinaryData getState()
               throws UnsupportedOperationException,
                      IllegalStateException,
                      DeviceException
```

Returns the `BinarySensor`current state. It's a getter method for `PROPERTY_STATE` property.

Returns:

The `BinarySensor` 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:

`BinaryData`

Interface Keypad

org.osgi.service.functionaldevice.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.	77

Fields inherited from interface org.osgi.service.functionaldevice.DeviceFunction
META_INFO_DESCRIPTION, META_INFO_MAX_VALUE, META_INFO_MIN_VALUE, META_INFO_OPERATION_ARG_OUT, META_INFO_OPERATION_ARGS_IN_PREFIX, META_INFO_PROPERTY_ACCESS, META_INFO_PROPERTY_ACCESS_EVENTABLE, META_INFO_PROPERTY_ACCESS_READABLE, META_INFO_PROPERTY_ACCESS_WRITABLE, META_INFO_RESOLUTION, META_INFO_UNIT, META_INFO_VALUES, PROPERTY_DESCRIPTION, PROPERTY_DEVICE_UID, PROPERTY_OPERATION_NAMES, PROPERTY_PROPERTY_NAMES, PROPERTY_REFERENCE_UIDS, PROPERTY_UID

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

Class KeypadData

org.osgi.service.functionaldevice.functions

```
java.lang.Object
  └org.osgi.service.functionaldevice.functions.KeypadData
```

```
public class KeypadData
extends Object
```

Represents a keypad event data that is collected when a change with some key from device keypad has occurred.

See Also:
Keypad

Field Summary		Page
static int	**EVENT_TYPE_PRESSED** Represents a keypad event type for a key pressed.	79
static int	**EVENT_TYPE_PRESSED_DOUBLE** Represents a keypad event type for a double key pressed.	79
static int	**EVENT_TYPE_PRESSED_DOUBLE_LONG** Represents a keypad event type for a double and long key pressed.	79
static int	**EVENT_TYPE_PRESSED_LONG** Represents a keypad event type for a long key pressed.	79
static int	**EVENT_TYPE_RELEASED** Represents a keypad event type for a key released.	79
int	**eventType** Represents the keypad event type.	79
int	**keyCode** Represents the key code.	79
String	**keyName** Represents the key name, if it's available.	80
Map	**metadata** Represents `KeypadData` metadata in an unmodifiable `Map`.	80
long	**timestamp** Represents `KeypadData` timestamp.	80

Constructor Summary	Page
KeypadData(int eventType, int keyCode, String keyName, long timestamp, Map metadata) Constructs new `KeypadData` instance with the specified arguments.	80

Method Summary		Page
int	**getEventType**() Returns the event type.	80
int	**getKeyCode**() The code of the key.	81
String	**getKeyName**() Represents a human readable name of the corresponding key code.	81

Map	**getMetadata**() Returns `KeypadData` metadata.	*81*
long	**getTimestamp**() Returns `KeypadData` timestamp.	*81*

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 immutable field is accessible with `getEventType()` getter.

keyCode

```
public final int keyCode
```

Represents the key code. The immutable field is accessible with `getKeyCode()` getter.

keyName

```
public final String keyName
```

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

timestamp

```
public final long timestamp
```

Represents `KeypadData` timestamp. The timestamp is the difference between the value collecting time and midnight, January 1, 1970 UTC. It's measured in milliseconds. If possible, the value should be provided by the device, otherwise the device driver can generate that info. `Long.MIN_VALUE` value means no timestamp. The immutable field is accessible with `getTimestamp()` getter.

metadata

```
public final Map metadata
```

Represents `KeypadData` metadata in an unmodifiable `Map`. The immutable field is accessible with `getMetadata()` getter.

Constructor Detail

KeypadData

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

Constructs new `KeypadData` instance with the specified arguments.

Parameters:

`eventType` - The event type.
`keyCode` - The key code.
`keyName` - The key name, if available.
`timestamp` - The event timestamp.
`metadata` - The `KeypadData` metadata.

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.

getTimestamp

```
public long getTimestamp()
```

Returns `KeypadData` timestamp. The timestamp is the difference between the value collecting time and midnight, January 1, 1970 UTC. It's measured in milliseconds. If possible, the value should be provided by the device, otherwise the device driver can generate that info. `Long.MIN_VALUE` value means no timestamp.

Returns:

`KeypadData` timestamp.

getMetadata

```
public Map getMetadata()
```

Returns `KeypadData` metadata.

Returns:

`KeypadData` metadata.

Interface Meter

org.osgi.service.functionaldevice.functions

All Superinterfaces:

DeviceFunction

```
public interface Meter
extends DeviceFunction
```

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

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

See Also:

`MultiLevelData`

Field Summary		**Page**
String	**FLOW_IN** Represents the metering consumption flow.	83
String	**FLOW_OUT** Represents the metering generation flow.	83
String	**OPERATION_RESET_TOTAL** Specifies the reset total operation name.	83
String	**PROPERTY_CURRENT** Specifies the current consumption property name.	83
String	**PROPERTY_FLOW** Specifies the metering flow property name.	83
String	**PROPERTY_TOTAL** Specifies the total consumption property name.	83

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

`META_INFO_DESCRIPTION, META_INFO_MAX_VALUE, META_INFO_MIN_VALUE, META_INFO_OPERATION_ARG_OUT, META_INFO_OPERATION_ARGS_IN_PREFIX, META_INFO_PROPERTY_ACCESS, META_INFO_PROPERTY_ACCESS_EVENTABLE, META_INFO_PROPERTY_ACCESS_READABLE, META_INFO_PROPERTY_ACCESS_WRITABLE, META_INFO_RESOLUTION, META_INFO_UNIT, META_INFO_VALUES, PROPERTY_DESCRIPTION, PROPERTY_DEVICE_UID, PROPERTY_OPERATION_NAMES, PROPERTY_PROPERTY_NAMES, PROPERTY_REFERENCE_UIDS, PROPERTY_UID`

Method Summary		**Page**
MultiLevel Data	**getCurrent**() Returns the current metering info.	84
String	**getFlow**() Returns the metering flow.	84

MultiLevelData	**getTotal**() Returns the total metering info.	*84*
void	**resetTotal**() Resets the total metering info.	*85*

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

Field Detail

FLOW_IN

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

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

FLOW_OUT

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

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

PROPERTY_FLOW

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

Specifies the metering flow property name. The property can be read with `getFlow()` getter.

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

getFlow

```
String getFlow()
        throws UnsupportedOperationException,
               IllegalStateException,
               DeviceException
```

Returns the metering flow. It's a getter method for `PROPERTY_FLOW` property.

Returns:
The metering flow.

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.

---

getCurrent

```
MultiLevelData 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:
`MultiLevelData`

---

getTotal

```
MultiLevelData 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:
`MultiLevelData`

---

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.functionaldevice.functions

All Superinterfaces:

DeviceFunction

```
public interface MultiLevelControl
extends DeviceFunction
```

`MultiLevelControl` Device Function provides multi-level control support. The function level is accessible with `getLevel()` getter and `setLevel(MultiLevelData)` setter.

See Also:

`MultiLevelData`

Field Summary		**Page**
String	**PROPERTY_LEVEL** Specifies the level property name.	86

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

`META_INFO_DESCRIPTION, META_INFO_MAX_VALUE, META_INFO_MIN_VALUE, META_INFO_OPERATION_ARG_OUT, META_INFO_OPERATION_ARGS_IN_PREFIX, META_INFO_PROPERTY_ACCESS, META_INFO_PROPERTY_ACCESS_EVENTABLE, META_INFO_PROPERTY_ACCESS_READABLE, META_INFO_PROPERTY_ACCESS_WRITABLE, META_INFO_RESOLUTION, META_INFO_UNIT, META_INFO_VALUES, PROPERTY_DESCRIPTION, PROPERTY_DEVICE_UID, PROPERTY_OPERATION_NAMES, PROPERTY_PROPERTY_NAMES, PROPERTY_REFERENCE_UIDS, PROPERTY_UID`

Method Summary		**Page**
MultiLevelData	**getLevel**() Returns `MultiLevelControl` level.	87
void	**setLevel**(MultiLevelData level) Sets `MultiLevelControl` level.	87

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

`getOperationMetaData, getPropertyMetaData`

Field Detail

PROPERTY_LEVEL

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

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

Method Detail

getLevel

```
MultiLevelData getLevel()
                    throws UnsupportedOperationException,
                           IllegalStateException,
                           DeviceException
```

Returns `MultiLevelControl` level. It's a getter method for `PROPERTY_LEVEL` 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:
`MultiLevelData`

---

setLevel

```
void setLevel(MultiLevelData level)
       throws UnsupportedOperationException,
              IllegalStateException,
              DeviceException
```

Sets `MultiLevelControl` level. It's a setter method for `PROPERTY_LEVEL` property.

Parameters:
`level` - The new `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:
`MultiLevelData`

Class MultiLevelData

org.osgi.service.functionaldevice.functions

```
java.lang.Object
  └org.osgi.service.functionaldevice.functions.MultiLevelData
```

```
public class MultiLevelData
extends Object
```

Device Function multi-level data wrapper. It can contain a `BigDecimal` value, timestamp, measurement unit and additional metadata.

See Also:

MultiLevelControl, MultiLevelSensor, Meter

Field Summary		Page
`Map`	**metadata** Represents `MultiLevelData` metadata in an unmodifiable `Map`.	89
`long`	**timestamp** Represents `MultiLevelData` timestamp.	89
`String`	**unit** Represents `MultiLevelData` measurement unit as it's described in DeviceFunction.META_INFO_UNIT.	89
`BigDecimal`	**value** Represents `MultiLevelData` value.	88

Constructor Summary	Page
MultiLevelData`(BigDecimal value, long timestamp, String unit, Map metadata)` Constructs new `MultiLevelData` instance with the specified arguments.	89

Method Summary		Page
`Map`	**getMetadata**`()` Returns `MultiLevelData` metadata.	90
`long`	**getTimestamp**`()` Returns `MultiLevelData` timestamp.	90
`String`	**getUnit**`()` Returns `MultiLevelData` measurement unit as it's described in DeviceFunction.META_INFO_UNIT .	90
`BigDecimal`	**getValue**`()` Returns `MultiLevelData` value.	89

Field Detail

value

```
public final BigDecimal value
```

Represents `MultiLevelData` value. The immutable field is accessible with `getValue()` getter. The field type is `BigDecimal` instead of `double` to guarantee value accuracy.

---

timestamp

```
public final long timestamp
```

Represents `MultiLevelData` timestamp. The timestamp is the difference between the value collecting time and midnight, January 1, 1970 UTC. It's measured in milliseconds. If possible, the value should be provided by the device, otherwise the device driver can generate that info. `Long.MIN_VALUE` value means no timestamp. The immutable field is accessible with `getTimestamp()` getter.

---

unit

```
public final String unit
```

Represents `MultiLevelData` measurement unit as it's described in `DeviceFunction.META_INFO_UNIT`. It's an immutable field.

---

metadata

```
public final Map metadata
```

Represents `MultiLevelData` metadata in an unmodifiable `Map`. The immutable field is accessible with `getMetadata()` getter.

Constructor Detail

MultiLevelData

```
public MultiLevelData(BigDecimal value,
                      long timestamp,
                      String unit,
                      Map metadata)
```

Constructs new `MultiLevelData` instance with the specified arguments.

Parameters:

`value` - The multi-level value.
`timestamp` - The value timestamp.
`unit` - The value measurement unit.
`metadata` - The value metadata.

Method Detail

getValue

```
public BigDecimal getValue()
```

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

Returns:

The `MultiLevelData` value.

---

getTimestamp

```
public long getTimestamp()
```

Returns `MultiLevelData` timestamp. The timestamp is the difference between the value collecting time and midnight, January 1, 1970 UTC. It's measured in milliseconds. If possible, the value should be provided by the device, otherwise the device driver can generate that info. `Long.MIN_VALUE` value means no timestamp.

Returns:

`MultiLevelData` timestamp.

---

getUnit

```
public String getUnit()
```

Returns `MultiLevelData` measurement unit as it's described in `DeviceFunction.META_INFO_UNIT` .

Returns:

The `MultiLevelData` measurement unit.

---

getMetadata

```
public Map getMetadata()
```

Returns `MultiLevelData` metadata.

Returns:

`MultiLevelData` metadata.

Interface MultiLevelSensor

org.osgi.service.functionaldevice.functions

All Superinterfaces:

DeviceFunction

```
public interface MultiLevelSensor
extends DeviceFunction
```

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

See Also:

`MultiLevelData`

Field Summary		Page
String	**PROPERTY_STATE** Specifies the state property name.	91

Fields inherited from interface org.osgi.service.functionaldevice.DeviceFunction
META_INFO_DESCRIPTION, META_INFO_MAX_VALUE, META_INFO_MIN_VALUE, META_INFO_OPERATION_ARG_OUT, META_INFO_OPERATION_ARGS_IN_PREFIX, META_INFO_PROPERTY_ACCESS, META_INFO_PROPERTY_ACCESS_EVENTABLE, META_INFO_PROPERTY_ACCESS_READABLE, META_INFO_PROPERTY_ACCESS_WRITABLE, META_INFO_RESOLUTION, META_INFO_UNIT, META_INFO_VALUES, PROPERTY_DESCRIPTION, PROPERTY_DEVICE_UID, PROPERTY_OPERATION_NAMES, PROPERTY_PROPERTY_NAMES, PROPERTY_REFERENCE_UIDS, PROPERTY_UID

Method Summary		Page
MultiLevelData	**getState**() Returns the `MultiLevelSensor` current state.	92

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

Field Detail

PROPERTY_STATE

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

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

See Also:

`MultiLevelData`

Method Detail

getState

```
MultiLevelData getState()
                 throws UnsupportedOperationException,
                        IllegalStateException,
                        DeviceException
```

Returns the `MultiLevelSensor` current state. It's a getter method for `PROPERTY_STATE` 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:

`MultiLevelData`

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_ENABLE	Device.enable()
ACTION_DISABLE	Device.disable()

ACTION_PROPERTY	Device.setProperty(String, Object)
	Device.setProperties(String[], Object[])

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

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

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

```
org.osgi.services.functionaldevice.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.services.functionaldevice.Device.remove()` method must check that the caller has an access to the operation:

```
public class DeviceImpl implements FunctionalDevice {
   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, enable etc. Once the Functional Device Permission is checked against the caller the implementation will proceed with the actual operation. The operation can require a number of other permissions to complete. The implementation must isolate the caller from such permission checks by use of proper privileged blocks.

10 Document Support

10.1 References

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

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

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

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

10.2 Author's Address

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

10.3 Acronyms and Abbreviations

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

10.4 End of Document