

FELIX UPnP Documentation

Introduction

The Felix UPnP project provides an implementation of the OSGi UPnP specification (version 1.1) as described in the OSGi Service Compendium (Release 4) [1]. The specification is implemented by the *org.apache.felix.upnp.basedriver* bundle and it comes with other bundles, which have been developed to ease the writing and testing of UPnP code.

The OSGi UPnP specification defines a set of interfaces which should be used by the developers in order to write UPnP Devices and UPnP Control Points on the OSGi Service Platform. From the OSGi point of view, UPnP devices are services registered with the framework, thus the different phases of the UPnP protocol stack, as defined in the UPnP™ Device Architecture (UDA 1.0) [2], have been mapped to the discovery and notification mechanisms offered by the OSGi framework.

The specification defines a UPnP Base Driver component that acts as software bridge between UPnP networks and OSGi. Developers writing UPnP code do not need to interact directly with the driver through some API. The driver works in background by exporting the registered services as UPnP devices, and by registering as services the UPnP devices discovered on UPnP networks. However, the Felix UPnP project has defined few additional interfaces, so a base knowledge of the way the UPnP Base Driver works is useful and will help developers to write their code.

The document is organized as follow:

Getting Started	2
Common Problems.....	3
Overview of the Base Driver Architecture	4
Testing UPnP devices	6
The UPnP examples	7
Sample TV and Clock	8
The BinaryLight example	8
Writing UPnP Devices and Control Points	9
The Extra bundle and the driver interfaces	10
Known problems	11
Acknowledgments.....	11
References	11

Getting Started

Assuming that, as described in “[Building Felix](#)” web page, you have checked out the Felix project in the \$FELIX_HOME directory, the Felix UPnP project is located at \$FELIX_HOME/trunk/upnp directory. The project is organized in different directories shown in Figure 1.

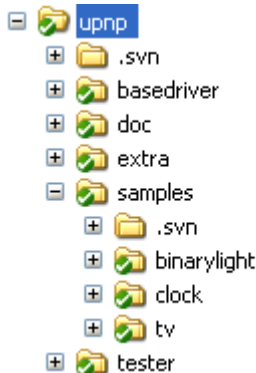


Figure 1 The Felix UPnP project structure

The *basedriver* directory contains the project of the bundle implementing the UPnP spec. In the *samples* directory there are three projects implementing simple test devices, while the *tester* and *extra* directories are projects providing additional utilities for the UPnP development. At last, the *doc* directory contains this documentation and a script file to launch all the Felix UPnP bundles.

After building the Felix project, you can start the script file “*upnp.sh.bat*” inside the */upnp/doc* directory; it launches a Felix runtime with all the UPnP bundles released by the project. The script file defines a profile called “*upnp*” and the list of bundles¹ installed with the profile is shown in Figure 2. The UPnP Tester is a bundle that provides a browser utility to control and subscribe all the UPnP devices registered with the OSGi framework. After executing the script, you should see in the window opened by the UPnP Tester bundle three UPnP devices (left panel in Figure 3.a), which correspond to the TV, Clock and BinaryLight devices shown in Figure 3. Of course the number of discovered devices may be higher if other UPnP devices are installed in your local network

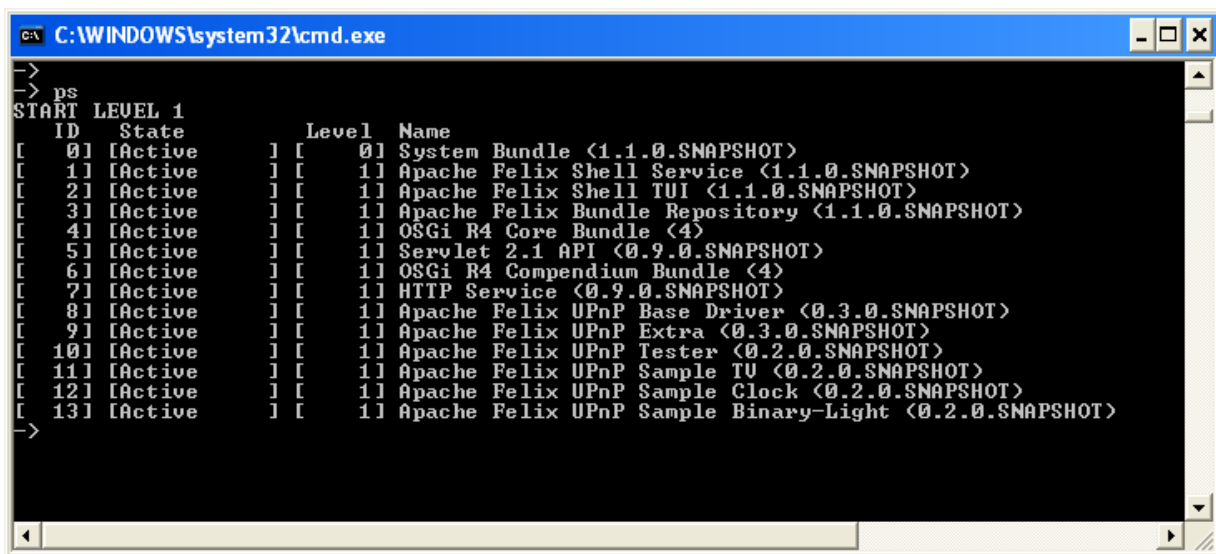


Figure 2 Bundles installed by the script “*upnp.sh.bat*”

¹ The actual version of the bundles may be different.

To stop the devices launched by the script you can close their windows, while to start them again type “start 10 11 12 13” from the Felix shell. See the sections “Testing UPnP devices” and “The UPnP Examples” for details on how to use the these bundles.

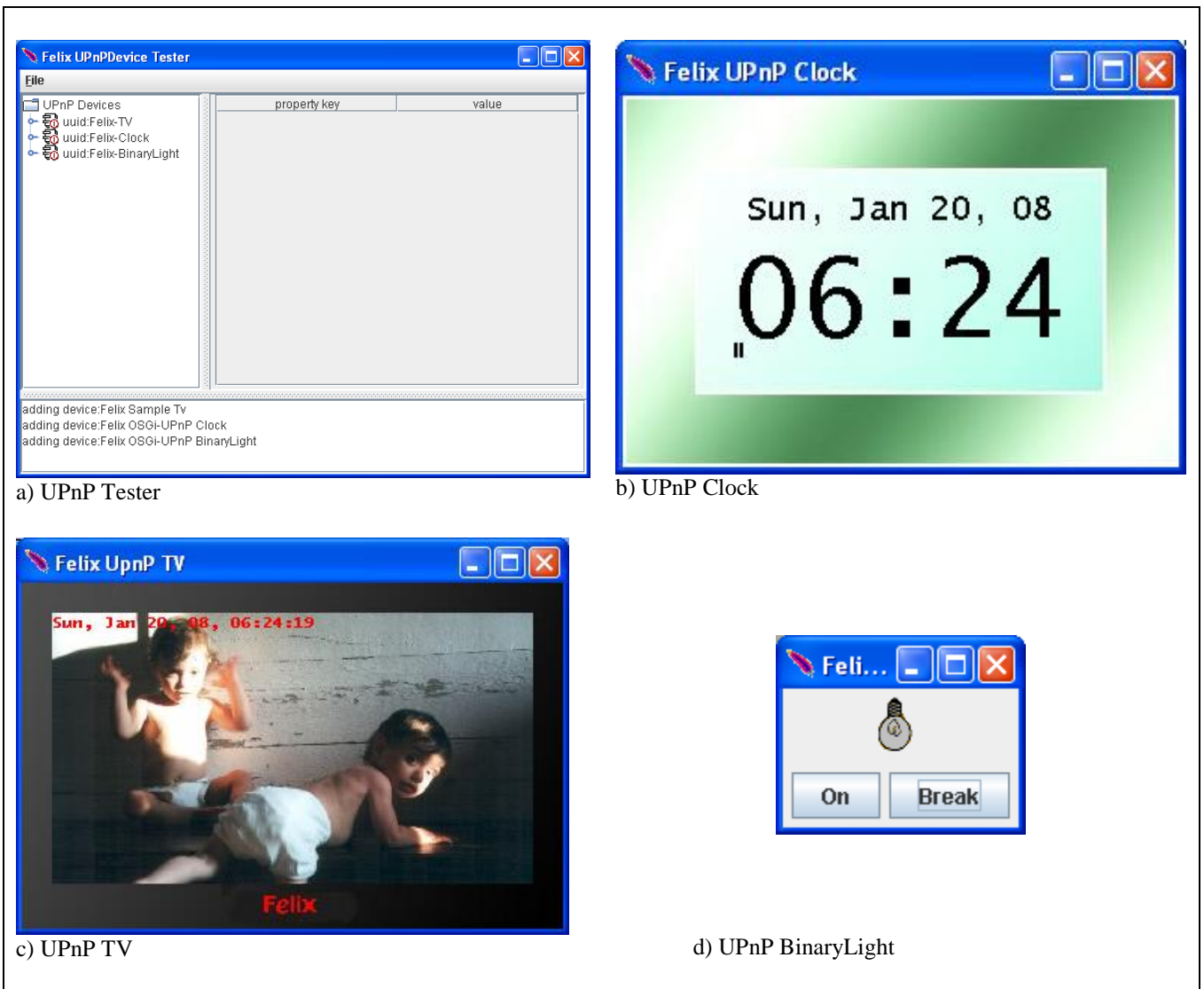


Figure 3 The GUIs started by the script “upnp.sh.bat”

The Felix build process by default uses the JDK1.4 as target class library for all the UPnP bundles. The UPnP Base Driver can be built also with the JDK1.3 as target; to this end you have to define the “platform” property in the command line: type “mvn Dplatform=jdk13 install” from the /upnp/basedriver directory. For details on configuring your Eclipse IDE see [3].

Common Issues

If you experience problems discovering the UPnP devices of your network:

- Check the configuration of your firewalls. UPnP discovery is based on multicast messages over UDP that usually are not filtered by firewalls, on the contrary the XML description of devices is retrieved using HTTP protocol; usually bound to

non standard ports which might be blocked. Check whether firewall is active on your host or on the host of the device you want discover.

- Install a loopback interface if needed.

The base driver by default is configured for not using the localhost as loopback interface. If you want to run and test UPnP devices on a machine disconnected by any network, you should install and activate a loopback interface. Pay attention to disable the loopback interface when you are connected to a network again, otherwise both interfaces will be used to expose the UPnP services registered with the framework.

Overview of the Base Driver Architecture

The Figure 4 shows a simplified component view of the base driver. The driver is composed of two components, the *exporter* and *importer*; both using the CyberDomo library, which is a modified version of the library released by the CyberLink for Java project [4], maintained by the Domoware project [5]. The library implements a full UPnP stack. The base driver acts as a bridge between OSGi and the UPnP networks².

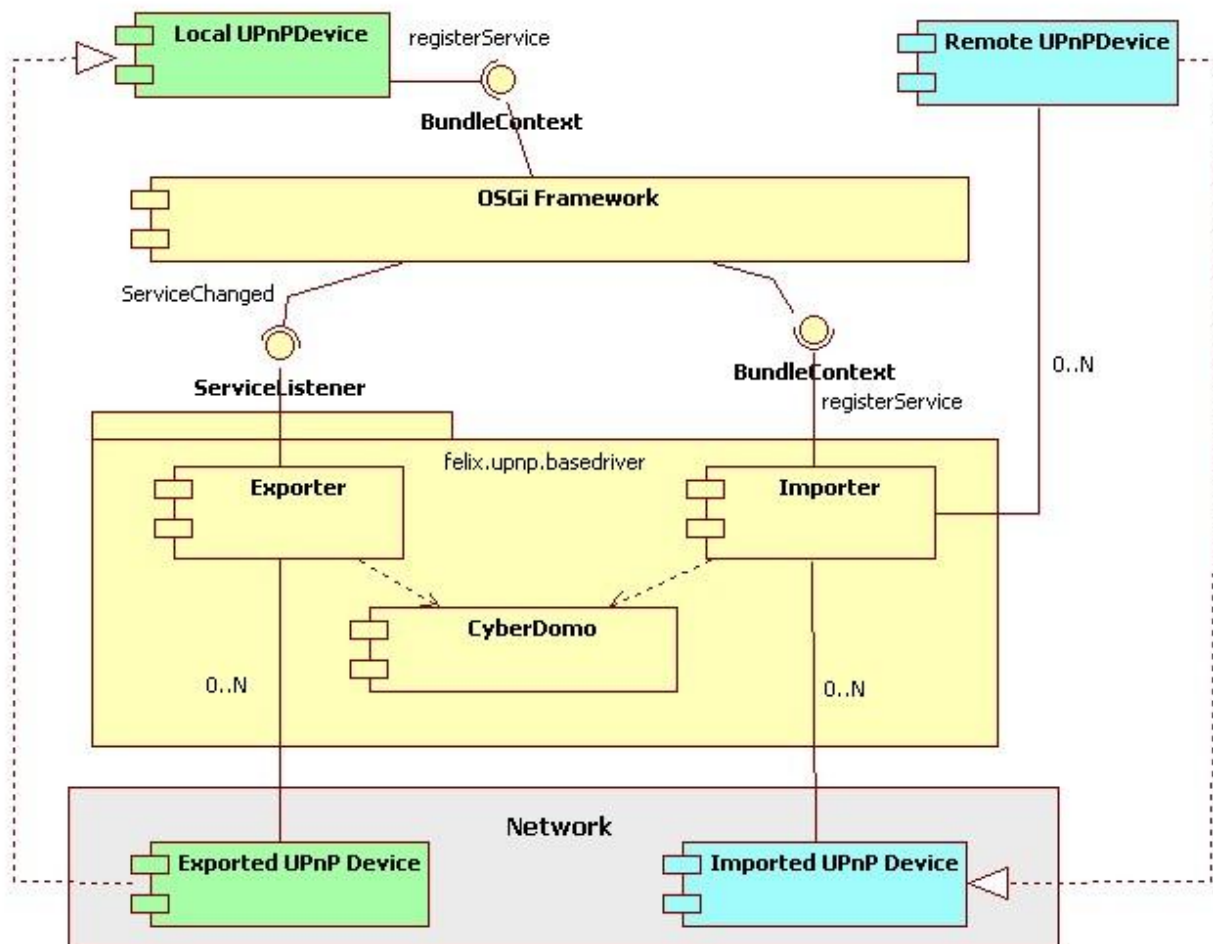


Figure 4 The UPnP Base Driver architecture

In order to instantiate UPnP devices, developers must register services implementing the interfaces represented in Figure 5 and provided by the *org.osgi.compendium* bundle. The *exporter* is registered

² By default the Base Driver has to bind all the available network interfaces

as [ServiceListener](#) with the framework and it automatically exposes on the networks each [UPnPDevice](#) service registered with the registration property [UPNP_EXPORT](#). The *importer* listens to the advertisements sent on the networks by external devices and registers with the framework one or more UPnPDevice services. Even if it is not required by the specification, the devices imported by the Felix base driver are labeled with the registration property [UPNP_IMPORT](#).

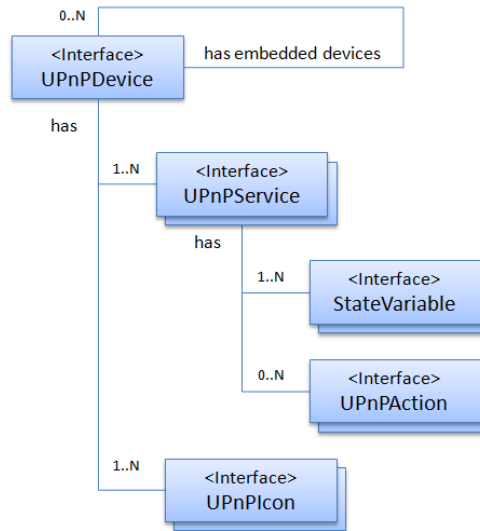


Figure 5 The UPnP Device interfaces

Working with UPnP Device from the OSGi point of view means to operate with services; the discovery, controlling and eventing phases of the UPnP protocol are naturally mapped to the OSGi service layer, which allows to publish, find, bind and notify events. There are some aspects that make it different to work with UPnP in OSGi with respect to other UPnP libraries, due basically to the centralized nature of the OSGi registry opposed to the distributed approach used in UPnP networks; some hints are provided in the section "Writing UPnP Devices and Control Points"

The Felix base driver comes with some system properties you can use to configure it at startup.

The system properties:

- `cyberdomo.ssdp.mx` (default 5)
- `cyberdomo.ssdp.bufferSize` (default 2048)
- `cyberdomo.ssdp.port` (default 1900)

are used by the UPnP stack library during the UPnP discovery process. The paper "Adaptive Jitter Control for UPnP M-Search" [6] provides a good analysis of the tuning of such parameters related to scalability issues. The MX parameter default has been set to 5 sec. Higher values improve the discovery effectiveness but increase the latency for new device discovery. The Intel "Device Spy" tool [7] uses a delay of 10 sec, the CyberLink for Java [4] library 3 sec. The SSDP port in UPnP specification is by default 1900 we allow the modification of such parameter.

The following system properties:

- `felix.upnpbase.exporter.enabled` (default true)
- `felix.upnpbase.importer.enabled` (default true)

can be used to enable or disable the two main components of the base driver. For example with small devices (ARM-based processor), disabling the exporting or importing of devices might reduce the resource consumption.

- `felix.upnpbase.log` (default `2`)
- `felix.upnpbase.cyberdomo.log` (default `false`)

are properties used to enable the different logging facilities offered by the base driver. You can also modify them at run time by using the GUI provided by the UPnP Tester bundle

Finally the following properties are used to set the networking parameters. The loopback interface is usually disabled :

- `felix.upnpbase.cyberdomo.net.loopback` (default `false`)
- `felix.upnpbase.cyberdomo.net.onlyIPV4` (default `true`)
- `felix.upnpbase.cyberdomo.net.onlyIPV6` (default `false`)

Testing UPnP devices

The `org.apache.felix.upnp.tester` bundle installs a component that shows the UPnP devices registered with the OSGi framework. It provides a GUI shown in the Figure 6 that can be used for controlling the discovered devices: by invoking actions and by subscribing for the state variable changes occurring on them.

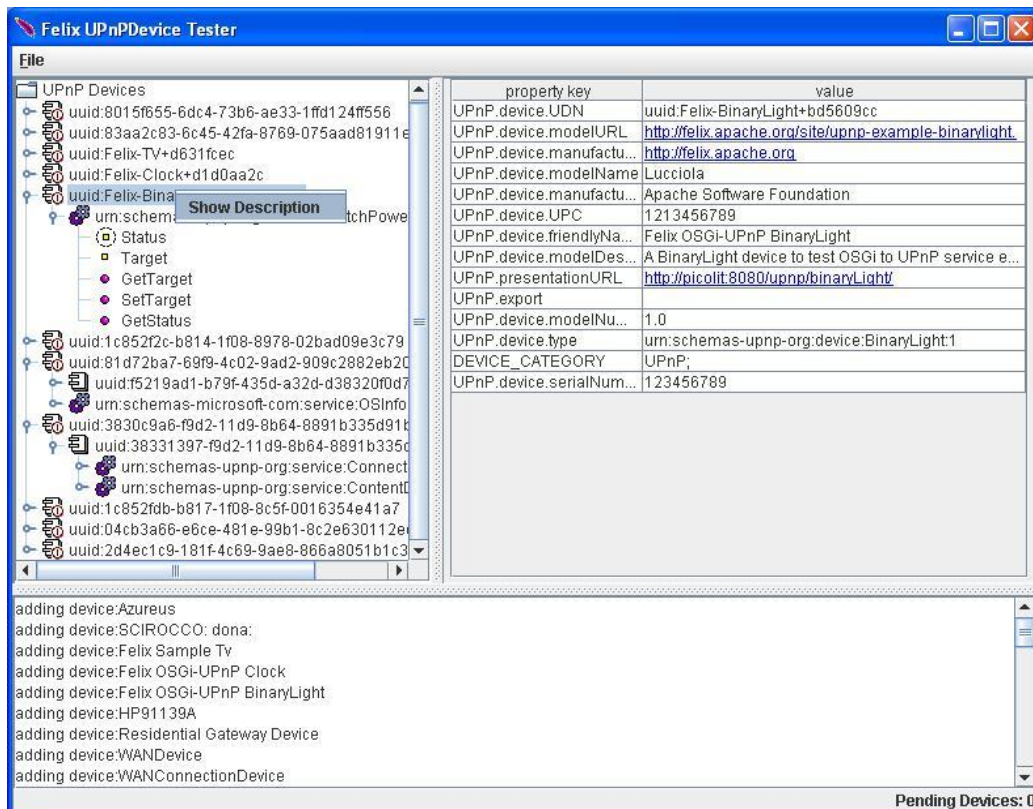









Figure 6 The UPnP Tester GUI

In the left panel you can browse the devices discovered on the OSGi platform. Remember that they may be registered by the base driver as well as by other bundles installed on the platform. Therefore stopping the base driver you will continue to see the local devices, but they will be no more

exported. The devices with their hierarchy of the embedded devices and services are represented as nodes of a tree with the following icons:

-  Root Device
-  Embedded Device
-  Service
-  Action
-  State Variable
-  Evented State Variable
-  Subscribed State Variable

By clicking on the Root Device icon, the register properties defined by the device are shown on the right panel. You can distinguish between exported and imported devices by looking for the property key “UPnP.export” and “UPnP.device.imported” respectively.

By right-clicking on the Root Device, Embedded device, or Service icon a context menu is displayed to open the XML description of devices and services (see Figure 6)

By clicking on the Service icon, the Service Id and Type are shown in the right panel together with buttons for subscribing all the state variables of the service. The received notify message is displayed on the bottom panel.

By clicking on the Action icons, the right panel displays a form where the input parameters of the action can be inserted. Use the “do Action” button to execute it; the results, if any, will be displayed in the table below the input parameters.

Finally, by clicking on the state variables shows the associated property keys as the default value and minimum and maximum value, if any.

Menus

- The “Search” menu forces the UPnP Base Driver to execute a UPnP M-Search for UPnP Root Devices or for all types of devices. This search is usually automatically executed during the start up of the base driver.
- The “Felix Logger” and “Cyber Debugger” menus enable displaying of the messages received and sent by the base driver (i.e. the content of the UDP communication).
- The “Print Pending Devices” is a utility menu to verify whether incomplete hierarchy of embedded devices have been registered with the framework.
- The “Check Errata UPnPDevices” menu may help the user verify that all the local UPnP Devices have been registered with the mandatory properties, otherwise they would not be exported.

The UPnP examples

The UPnP examples released by the UPnP project are simple UPnP devices developed as a proof of concept. The first two examples, the TV and Clock, are used to check the importing and exporting capabilities of the base driver. The third one, the Binary Light, implements a standard UPnP DCP and provides additionally a UPnP presentation page.

Sample TV and Clock

These devices are dual version of the sample devices developed by the project “Cyberlink for Java” by Satoshi Konno. They have been rewritten according to the OSGi specification and can be used to check the importing and exporting capabilities of the base driver. The simulated TV screen is used to show the messages received by the Clock device and other simulated device like the Air Conditionator and Washing Machine. When launching the original version of such devices you will see that the Felix TV running on the OSGi platform is able to receive the messages from UPnP devices running on different platform and imported in OSGi. At the same time, the Cyberlink TV is able to receive the time event generated by the Felix Clock device and exported by the base driver.



Figure 7 The Felix UPnP TV GUI



Figure 8 The Felix UPnP Clock GUI

If you want to avoid installing the Cyberlink devices, you can run a second instance of Felix by clicking on the batch file again. In this case the Felix TV and Clock will be exported and re-imported by both Felix runtimes and you will see a duplicated TV and Clock device on each platform. Notice that you can stop in any moment a device by closing its window. You can start it again from the Felix shell by selecting the respective bundle ID. Starting with two running instances of the Felix Clock, you can stop the first one and the TVs will lose for a moment the time signal. In fact, being subscribed to the Clock device type and not to a specific device instance, they will receive the time event from the remaining device Clock. One TV will be notified from the clock running on the same platform, while the other will receive the events from an imported TV device. As soon as you stop also the second clock device, the Time message will disappear from both the TVs.

The BinaryLight example

The Binary Light device, according the UPnP DCP, shows a graphical interface you can use to switch on/off the light and to simulate the breaking of the lamp bulb. In this last circumstance you can see, by using the Felix UPnPDevice Tester interface, that the values of the “Status” and “Target” variable may be different. While the “Target” variable represents the expected status after invoking the related action, the “Status” variable describes the real status of the Light Device.

This example, by exploiting the Felix HTTP Service implementation, installs a UPnP presentation page. By code you can retrieve the presentation page URL by looking for the service property called “UPnP.presentationURL”. This property is also visible, as link, through the interface provided by the Tester bundle. Accessing the presentation page by means of a web browser you can switch the light status by clicking on the Light image: the icon on the device windows changes accordingly.



Figure 9 The BinaryLight GUI and the presentation page

The source code for the Binary light is slightly different from the one for TV and Clock code because it has been written starting from a Light model which notifies its changes through the *PropertyChangeListener* interface.

Writing UPnP Devices and Control Points

The OSGi UPnP Specification (v 1.1) [1] dedicates section 111.6 “Working with a UPnP Device” to describe the details of implementing UPnP Devices on OSGi. Here we provide some hints on the main differences you may encounter working with OSGi with respect to the UDA 1.0 specification [112].

The first peculiarity is that OSGi provides a centralized register for discovering of UPnP devices as opposed to the distributed mechanism of the UPnP protocol stack. Thus, while in the UPnP networks the steps for subscribing the services of some device are typically 1) **discover** the required device and 2) **subscribe** the service, within the OSGi platform a Control Point may register an interest in receiving notify events even before the device is really plugged on the network. This is possible because the subscription mechanism is based on the [UPnPEventListener](#) interface that is used for registering OSGi services, which ultimately handles the notify messages sent by the producers of the events. The base driver (importer) keeps track of such UPnPEventListener services and as soon as a matching service is discovered on the UPnP network, a subscription is made on behalf of the registered listeners.

On the other hand, even if it is enough to register a service implementing the [UPnPDevice](#) interface to expose it as UPnP devices on the network, the developers have to implement on their own the event management required by the UPnP technology. From this point of view, for each evented state variable declared by the UPnP device, the developers have to monitor [UPnPEventListener](#) services that is error prone³. The correct implementation of the UPnP eventing phase is left entirely to developers. In particular, in UDA 1.0, the first time a Control Point subscribes a service, the current value of its state variables should soon be delivered to it. To manage this situation in a standard way, the last OSGi UPnP specification defined the extended interface [UPnPLocalStateVariable](#). In fact, the previous basic interface UPnPStateVariable provided only a descriptive interface which did not enable to get the value of a state variable without knowing the final implementation class. Every developer should use this new interface in order to allow the specification of helper classes that ease the subscription/notify management (see [UPnPEventNotifier](#) below).

We have factorized and released part of the code used by the UPnP examples with the `org.apache.felix.upnp.extra` bundle.

³ Developers should monitor UPnPEventListener services with a [filter](#) matching either the own service Id or service type, either the own device Id or device type and even a empty filter which are usually used to express interest for every UPnP device

The Extra bundle and the driver interfaces

We provide some utility classes and services through the extra bundle and the services registered by the UPnP Base Driver.

In the Extra bundle the class [org.apache.felix.upnp.extra.util.UPnPSubscriber](#) can be instantiated to subscribe one or more services. The constructor takes two parameters a [BundleContext](#) reference and a [UPnPEventListener](#) reference. In this class the method `subscribe(Filter aFilter)` is a general and powerful way to subscribe to any service by using an [LDAP filter](#). For example by using the string :

```
"(& (UPnP.device.type=urn:schemas-upnp-org:device:BinaryLight:1)
  (UPnP.service.type= urn:schemas-upnp-org:service:SwitchPower:1))"
```

we would subscribe to the SwitchPower service offered by any device implementing the BinaryLight profile. Looking at the Felix UPnP TV sample code, the UPnPSubscriber class is used in the file [org.apache.felix.upnp.sample.tv.TVDevice](#) to subscribe to the different service types offered by the Cyberlink sample devices. However, in this case, the utility method `subscribeEveryServiceType` is used to provide just the device and service types.

```
private final static String CLOCK_DEVICE_TYPE = "urn:schemas-upnp-org:device:clock:1";
private final static String TIME_SERVICE_TYPE = "urn:schemas-upnp-org:service:timer:1";

private final static String LIGHT_DEVICE_TYPE = "urn:schemas-upnp-org:device:light:1";
private final static String POWER_SERVICE_TYPE = "urn:schemas-upnp-org:service:power:1";

private final static String AIRCON_DEVICE_TYPE = "urn:schemas-upnp-org:device:aircon:1";
private final static String TEMP_SERVICE_TYPE = "urn:schemas-upnp-org:service:temp:1";

private final static String WASHER_DEVICE_TYPE = "urn:schemas-upnp-org:device:washer:1";
private final static String STATUS_SERVICE_TYPE = "urn:schemas-upnp-org:service:state:1";

public void doSubscribe()
{
    subscriber = new UPnPSubscriber(Activator.context, this);
    subscriber.subscribeEveryServiceType(CLOCK_DEVICE_TYPE, TIME_SERVICE_TYPE);
    subscriber.subscribeEveryServiceType(AIRCON_DEVICE_TYPE, TEMP_SERVICE_TYPE);
    subscriber.subscribeEveryServiceType(LIGHT_DEVICE_TYPE, POWER_SERVICE_TYPE);
    subscriber.subscribeEveryServiceType(WASHER_DEVICE_TYPE, STATUS_SERVICE_TYPE);
}

public void undoSubscribe() {
    subscriber.unsubscribeAll();
}
}
```

The class [org.apache.felix.upnp.extra.util.UPnPEventNotifier](#) is a utility class that manages the delivery of notifications for you. There are two constructors. The first one takes a [BundleContext](#), a [UPnPDevice](#), and a [UPnPService](#) reference. They are internally used to keep trace of all the registered UPnPEventListener that are interested in monitoring events generated by your UPnP service. [UPnPEventNotifier](#) implements the java beans [PropertyChangeListener](#) interface; once changes of the service state variables occurs you should call the method `propertyChange(PropertyChangeEvent evt)`. Alternatively, you may use the second constructor to pass a reference to a model implementing the interface: [EventSource](#) defined in the Extra bundle. This model should use the [PropertyChangeSupport](#) to keep trace of [PropertyChangeListener](#), and the related method `firePropertyChange` to notify changes. The [EventSource](#) interface is used internally by the [UPnPEventNotifier](#) to register itself as `propertyChangeListener` of the model. Thus, in this case, you don't have to call `propertyChange()` directly: it is a duty of your model. As an example, take a look at [LightModel](#) class in the BinaryLight bundle.

The Felix UPnP base driver registers a non standard service implementing two interfaces: [org.apache.felix.upnp.basedriver.controller.DevicesInfo](#);

org.apache.felix.upnp.basedriver.controller.DriverController;

The former can be used to retrieve the XML description of both devices and services. Other than be used for debugging purpose, it allows access to the UPnP schema extensions defined by UPnP Vendors. According to the UDA 1.0 [2] they consist of elements inserted in different points of the XML description and by convention starting with the prefix “X_”. This interface is used by the context menu handler of the UPnP Tester bundle.

The latter interface can be used to change the log messages of the base driver at runtime. Two different methods are available to modify the log level of the base driver or to enable the visualization of low level messages related to the UPnP stack protocol (CyberDomo). Furthermore, the interface allows developers to send an M-SEARCH discovery message to the UPnP networks, thus refreshing the list of imported devices.

Known issues

Currently the bundle does not support the following requirements:

- upnp.ssd.address Configuration Service
- exported device changes: if a service already exported as UPnP Device changes its own configuration, i.e.: implements new service, changes the friendly name, etc., the new service description is not reflected on the UPnP Device
- icons for exported device are not tested
- no localization support

Acknowledgments

The Felix UPnP base driver and related bundles were originally developed by the Domoware project [5] which targeted the OSGi R3. The driver is based on a modified version of the UPnP for Java library released by the Cyberlink project [4]. This version, called CyberDomo, is currently maintained by the Domoware project team and aligned to the Cyberlink library regularly.

References

1. OSGi Specifications - <http://www.osgi.org/Specifications/HomePage>
2. UPnP Device architecture - <http://www.upnp.org/resources/documents/CleanUPnPDA101-20031202s.pdf>
3. Integrating Felix inside Eclipse - <http://felix.apache.org/site/integration-of-felix-inside-eclipse.html>
4. Cyberlink for Java - <http://www.cybergarage.org/net/upnp/java/index.html>
5. Domoware Project - <http://domoware.isti.cnr.it/>
6. K. Mills, C. Dabrowski “Adaptive Jitter Control for UPnP M-Search” IEEE International Conference on Communications, 2003. ICC '03. page(s): 1008- 1013 vol.2 - <http://w3.antd.nist.gov/~mills/papers/Paper521.pdf>
7. Intel® Software for UPnP* Technology - <http://www.intel.com/cd/ids/developer/asmo-na/eng/downloads/upnp/index.htm>