EBS UPnP Device Software Development Kit (SDK)

User Manual

Revised July 2006

EBS Inc. 39 Court Street Groton MA 01450 USA
http://www.ebsembeddedsoftware.com

Table Of Contents

PART I - EBS UPnP Device User Guide

Section 1: Introduction

Universal Plug and Play (UPnP) is an open networking architecture for peer to peer network connectivity of UPnP enabled devices. UPnP provides a device the capability to discover and control other devices on a network. Devices act as servers providing the clients, known as control points, access and control to its published capabilities. Control points have the ability to invoke actions on any UPnP device on a network, control points can also subscribe to a device to continuously monitor the state of a device and its services.

UPnP architecture builds on existing networking protocols, such as IP, TCP, UDP, HTTP, HTML, SOAP, SSDP, GENA etc. and web standards like XML to make the communication and control possible. Any device having a TCP/IP network stack is capable of running UPnP regardless of its underlying operating system and hardware.

Section 2: UPnP Phases

UPnP device operates in phases, the following section briefly explains these phases and elaborates on the role of a device in each of these phase. Every UPnP phase has related network protocols which the device must support. These phases or steps collectively define how a UPnP device behaves on the network.

- *Addressing*. When the device is turned on it joins an IP network and acquires a unique address which other devices and control points can use to communicate with it. Address acquisition is done either using server based DHCP (if available) or using a server less Auto-IP protocol. Auto-IP is a method where the device on a network may automatically choose an IP address and subnet mask in the absence of a server. The underlying TCP/IP stack should make DHCP and Auto-IP functionality available to a UPnP device.

- *Discovery*. This is the next phase in which a UPnP device advertises itself and its services on the network to indicate their availability (or to announce their departure). The control point, in this phase, searches for devices and services. If the searched device or service is found the control point retrieves their description document which contains their detailed information.

  Simple Service Discovery Protocol (SSDP) is a discovery protocol used by the device and control point in this phase. This protocol allows a device to send presence announcement, indicating its availability, to a multicast address, which is listened to by all the UPnP devices and control points available on the network. The protocol also allows a control point to search for a specific device or a service on the network by issuing search requests on the multicast address.

  The device sends responses and advertisements that contain a URL to access device description document. This URL provides control points with the information they need to retrieve the device and service descriptions, using which the control point obtains complete information about the device and the services it offers.

- *Description*. In the description phase a control point develops detailed understanding about a device or a service by parsing and reading their description document which it obtained in the discovery phase. A description documents contains all the information that control point needs to start monitoring and controlling a target service on a device.
  Every UPnP device needs to list information about itself and its capabilities/services in form of XML-based description documents. These documents are strictly based on standard schema defined by UPnP forum. The schema clearly defines all the required and optional fields (in form of xml tags) that a description document must have.
  A device must maintain two types of description documents

  A. Device description document that contains all the information about the device such as manufacturer, make, model, serial number, base URL etc.; a list of services provided by the device; list of embedded devices and

  B. Services description documents for each of its service. A service description document describes detailed information about the service, including all its associated variables. This information is essential in-order to be able to monitor and control that service.

- *Control*. In this phase a control-point can control a device by invoking action on a service hosted by the device. Simple Object Access Protocol (SOAP) is used to communicate action requests and responses between the control point and the device during this phase. SOAP is a control protocol used to perform web based messaging and remote procedure calls (RPC). Control point constructs action requests using SOAP and delivers them over HTTP to the control URL of the service. The server parses the request, performs the action and sends an action response indicating a success or a failure.

- *Eventing*. In the eventing phase a control point can register (or subscribe) to receive event notifications from a device whenever the state of a service associated with the device changes. The UPnP architecture employs a Subscriber / Publisher model in which the control point can subscribe to a service offered by the device. The device acts as a publisher sending event notification to all the

subscribers whenever there is a change in the value of any state variable of the service. This allows the control points to constantly monitor the state of a service by subscribing to it, thus providing it with a capability to respond automatically to a state change. This phase employs General Event Notification Architecture (GENA), a publisher/subscriber system, to allow control points to request, renew or cancel a subscription on an event. The service maintains a list of all the subscribers that is updated upon receiving subscription, renewal, or cancellation messages from the subscriber and also upon change of an event. GENA messages, like SOAP messages, are delivered using HTTP over TCP/IP. All messages contians information in XML format, using standard xml tags defined by upnp forum.

Section 3: Server and Client Interaction Model

UPnP requires client and servers use both unicast and multicast messages for communication. Figure 1. depicts all the multicast messsages passed between client and server in a UpnP architecture. Multicast channel (IP address 239.255.255.250 :1900 for UPnP) is used by the server (or client) for messages that are intended to be received by all client (or servers) available on the network.

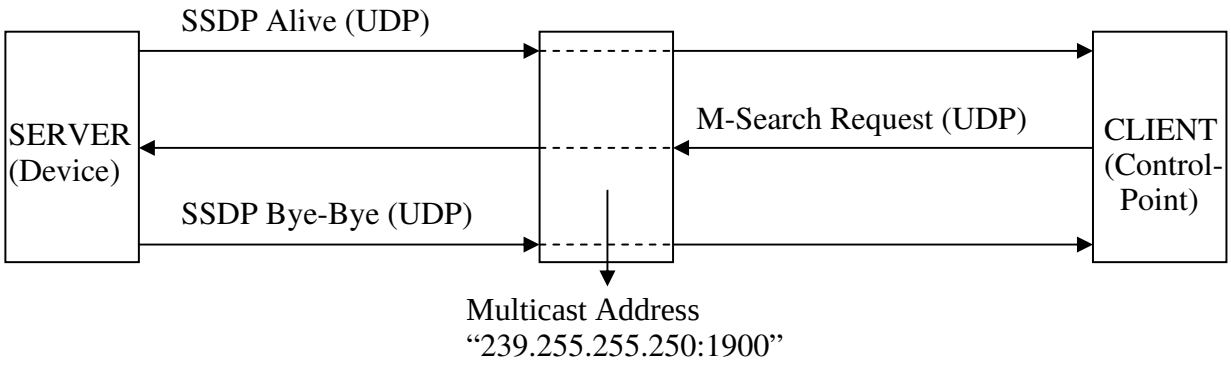

Figure 1: UPnP client server multicast communication

There are three such multicast message transitions:

- SSDP Alive: Device Available - The device sends presence notification to all the control points available on the network by sending alive messages to the multicast address. (Discovery Phase)
- SSDP Bye-Bye: Device Unavailable - Before shutting down the device indicates its unavailability by sending bye-bye notification to all the control points available on the network. (Discovery Phase)
- M-Search Request: Discovery Request - A control point (client) may send M-Search request to query to all devices (servers) available on the network to search for a specific service or device. (Discovery Phase)

All the other server client communications use unicast messages. Figure 2 depicts the client server interaction over unicast channel.

- M-Search Response: Discovery Response - If a device matches the search target of an multicast M-Search request issued by a control point, the device responds with a unicast message to the control point supplying it the url of the target (Discovery Phase)
- Get Description Document: Upon discovering a service or a device which matched the search criterion, the control point sends a unicast message requesting for their description documents from the device (Description Phase)
- OK 200: Acknowledgment - The device responds to the client's request for a description document for itself or one of it's service by sending the respective description document to the control point. Depending on the nature of the request a device may also send an error message as an acknowledgement (Description Phase)
- SOAP M-POST: Action Request - The control point can control and invoke action on a service offered by a device by sending it a unicast action request. (Control Phase)
- OK 200: Action Response - The device responds to action requests by sending status messages back to the control point indicating the success or failure of the action request.(Control Phase)

- Event Subscription: Subscription Request - A control point can send a unicast subscription request, renew request or cancel request message to a device. A control point can subscribe or unsubscribe to a service which enables it to monitor the state the service throughout the term of the subscription. (Eventing Phase)

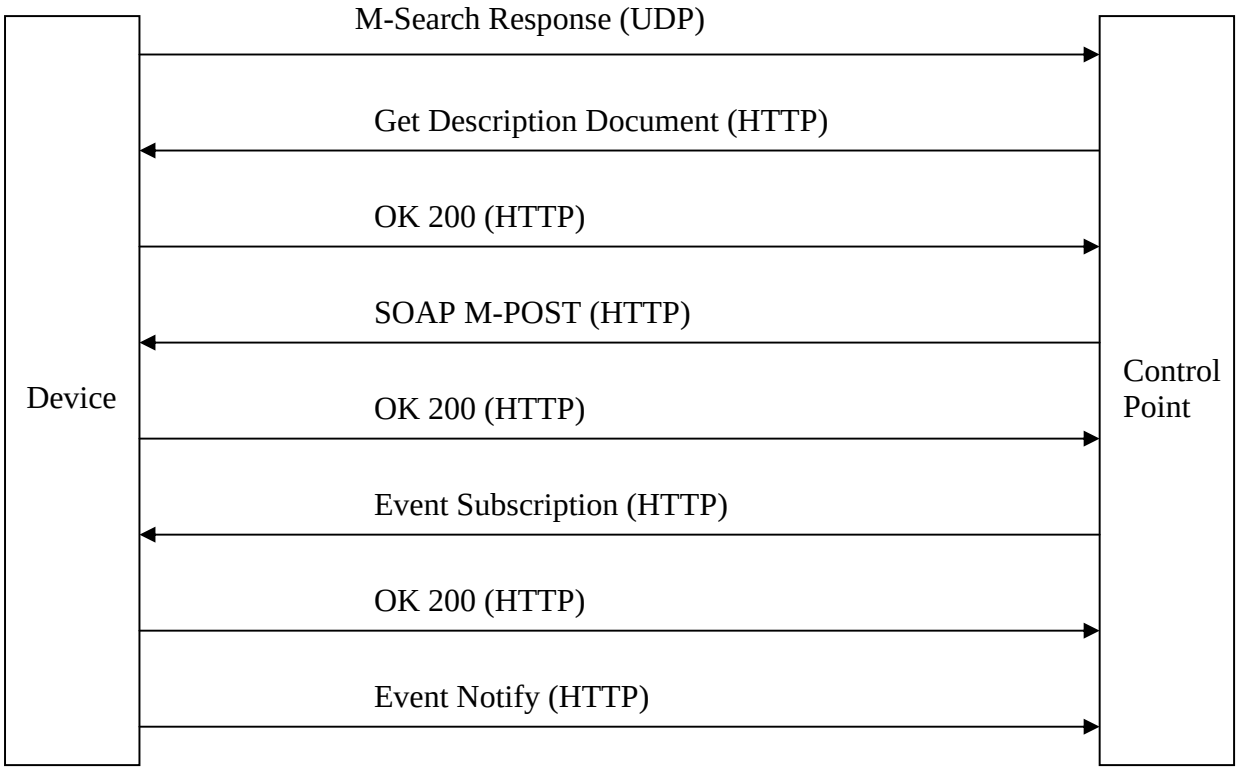

Figure 2: UPnP client server unicast communication

- OK 200: Subscription Response - The device sends a unicast status message to the control point in response to subscription request, renew request or cancel request messages. This response indicates the success or failure of the device to perform the request.(Eventing Phase)

- Event Notify: Each service maintains state variables which control the state of a service. If any of these state variables changes the device sends a unicast event message to all the subscriber of a service reflecting the change.(Eventing Phase)

Section 4: Software Development Kit (SDK) Architecture:

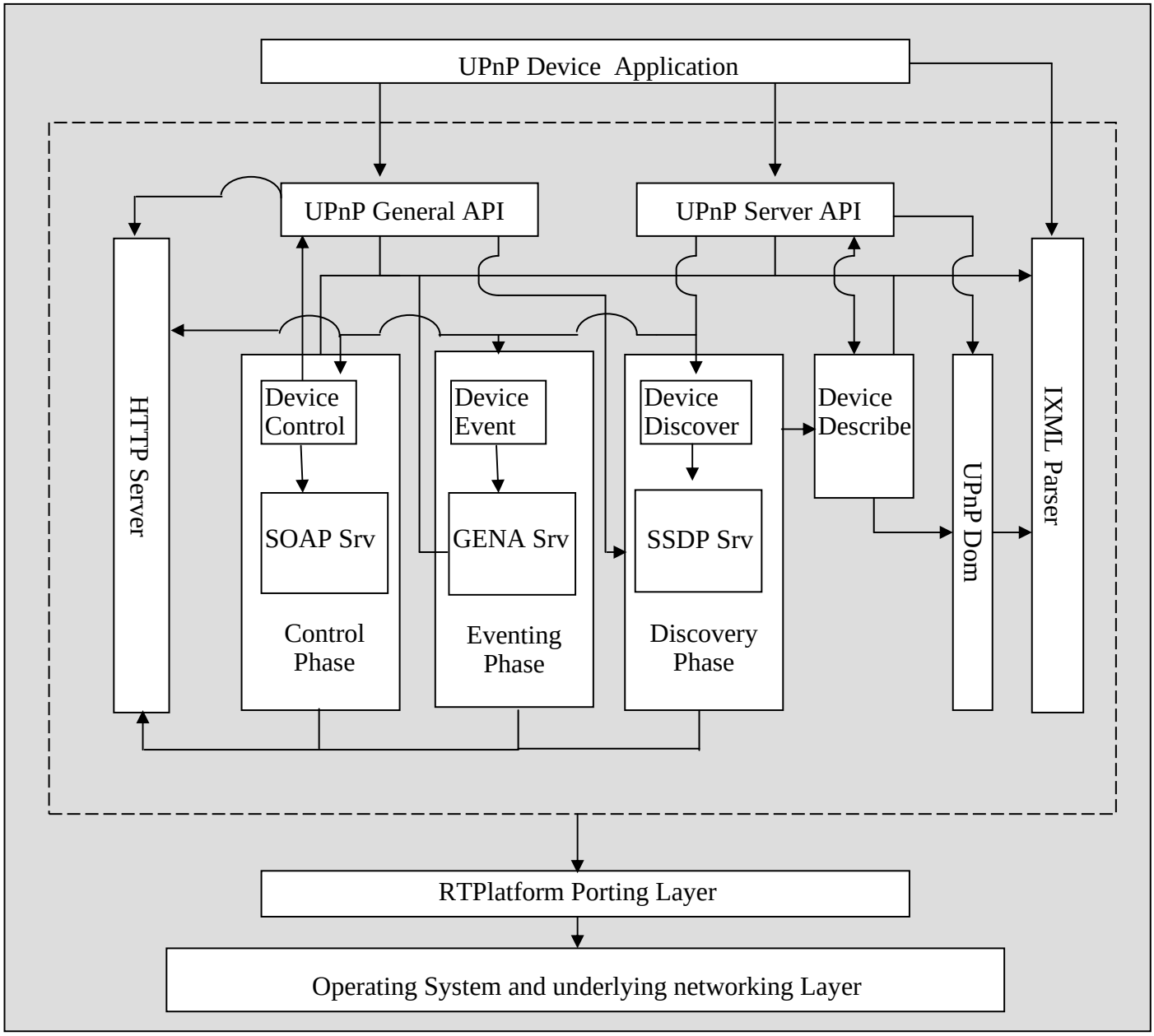

Figure 3: EBS UPnP 1.0 Architecture

Figure 3, shows where the different modules of the SDK sit in the UPnP library. Described below is a brief description of UPnP library modules

4.1 RTPLATFORM

RTPlatform is EBS's porting layer that allows the entire application to sit on any platform under it. The UPnP stack sits on RTPlatform; this makes it independent of the underlying operating system, or kernel that your system might be running on.
In order for your UPnP application to work on your platform you might need to modify a few RTPlatform files as explained in the RTPlatform manual

4.2 Application

The developer will need to develop an application that will drive the SDK to perform device or server side operations. The application sits on top of UPnP stack interacting directly with upnp.c, upnpsrv.c and IXML parser.

4.3 IXML Library

This library is used to parse and generate the XML documents as required by UPnP. XML is widely used in UPnP as the format in which all the information in UPnP is transferred, this provides UPnP with platform independence. Most modules of the SDK use IXML library.

4.4 UPnP.DOM

This module uses IXML library to perform some UPnP specific xml operations on an xml document or its Dom (data object model) representation.

4.5 HTTP Library (HTTP SERVER AND HTTP PARSER)

This library contains the web server and http parser used by UPnP. This library as other libraries sits on top of RTPlatform interacting directly with the internal modules of the UPnP stack.

4.6 UPnP.c and UPnPSrv.c

The UPnP device application interacts with these two modules. They provide the API's for the UPnP SDK. UPnPSrv.C module contains few API's that are specific to the device or the server side, while UPnP.C, is the file the containing general UPnP API's which provides the developers full control of UPnP stack.

4.7 DeviceAction.c and SoapSrv.c

These modules contain functions implementing the control phase of UPnP. See section 2 for a brief description of purpose of this phase.

4.8 DeviceEvent.c and GenaSrv.c

These modules contain functions implementing the eventing phase of UPnP. See section 2 for a brief description of purpose of this phase.

4.9 DeviceDescribe.c

This module contains functions that extract all the relevant information from the device and service description documents. This information is then stored in tables that are used at run time by various modules of the SDK.

4.10 DeviceDiscover.c and SsdpSrv.c

These modules contain functions implementing the discovery phase of discovery. See section 2 for a brief description of purpose of this phase.

Section 5: Getting Started

5.1 Creating Description Documents

UPnP device maintains a description document for itself, each of its services. These documents contain complete information about the device, and the services it offers. The first step is to create these XML-based description documents which are strictly based on standard schema defined by UPnP forum. These documents must conform to the UPnP Template Language, the XML syntax defined by the UPnP Forum for creating device and service descriptions. The standard scheme defines all the required and optional fields which a description document must contain.

All the newly created description documents must be placed in a directory which will be set as the root directory of UPnP's internal web server. In the supplied sample device this directory is named 'www-root', all the description documents for the sample device are placed in this directory. The section *writing your application* will explain in details how to set this new directory as the root diectory of the web server.

5.1.1 Creating a Device Description Document

This is the first step in developing an UPnP enabled device. Creating a description document is as simple as a 'fill in the blank' exercise. It does not require the developer to have any background knowledge of XML standard or XML technology.

Shown below is a sample of standard format of a device descriptor. Every UPnP device description document has the same format as shown in this sample. The fields (in form of xml tags) present in such documents are standard and are defined by UPnP forum.

To create a device description document, start by copying the sample shown below in a text editor.
The next step is to replace the italicized text in the sample below with device specific information. Table 5.1 contains detailed information about each field which is filled with an italicized description in the sample. Refer to table 5.1 to determine the correct / allowed value to fill for each of these fields. Once all the values for the necessary fields are inserted, the device description document is ready. The developer should save this file with an .xml extension and place it in a directory which will later be set as the root directory for the internal web server.

```
<?xml version="1.0"?>
<root xmlns="urn:schemas-upnp-org:device-1-0">
  <specVersion>
    <major>1</major>
    <minor>0</minor>
  </specVersion>
  <URLBase>base URL for all relative URLs</URLBase>
  <device>
    <deviceType>urn:schemas-upnp-org:device:deviceType:v</deviceType>
    <friendlyName>short user-friendly title</friendlyName>
    <manufacturer>manufacturer name</manufacturer>
    <manufacturerURL>URL to manufacturer site</manufacturerURL>
    <modelDescription>long user-friendly title</modelDescription>
    <modelName>model name</modelName>
    <modelNumber>model number</modelNumber>
    <modelURL>URL to model site</modelURL>
    <serialNumber>manufacturer's serial number</serialNumber>
    <UDN>uuid:UUID</UDN>
    <UPC>Universal Product Code</UPC>
    <iconList>
      <icon>
        <mimetype>image/format</mimetype>
```

```
      <width>horizontal pixels</width>
      <height>vertical pixels</height>
      <depth>color depth</depth>
      <url>URL to icon</url>
    </icon>
    XML to declare other icons, if any, go here
  </iconList>
  <serviceList>
    <service>
      <serviceType>urn:schemas-upnp-org:service:serviceType:v</serviceType>
      <serviceId>urn:upnp-org:serviceId:serviceID</serviceId>
      <SCPDURL>URL to service description</SCPDURL>
      <controlURL>URL for control</controlURL>
      <eventSubURL>URL for eventing</eventSubURL>
    </service>
    Declarations for other services defined by a UPnP Forum working
    committee (if any) go here
    Declarations for other services added by UPnP vendor (if any) go here
  </serviceList>
  <deviceList>
    Description of embedded devices defined by a UPnP Forum working
    committee (if any) go here
    Description of embedded devices added by UPnP vendor (if any) go here
  </deviceList>
  <presentationURL>URL for presentation</presentationURL>
 </device>
</root>
```

Source: http://www.upnp.org/download/Device-Template-Non-Annotated1.01.doc

Element	Required	Value
URLBase	Yes (For EBS SDK)	URL which is the base URL for the device. All relative URLs are appended to this base URL. The developer must supply the IPaddress of the device and the port number that the web server will use to serve the description documents. Eg. *http://192.168.0.6:80/* If a unique address for the device is not available or known to the developer then zero's (or corporate IP or localhost IP) can be filled in the IP field (e.g. *http://0.0.0.0/*) and when writing the application AUTO_ADDR should be set to 1 in UpnP_RegisterRootDevice API. This automatically assigns the device a unique address and a port number to serve its description pages.
deviceType	Yes	Describes the kind of device this is. The format for this field is: **urn:schemas-upnp-org:device:deviceType:v** Standard device type defined by UPnP committee must begins with **urn:schemas-upnp-org:device:** and are followed by the device type suffix, "**:**", and an integer device version. For example, say the device type for the device is Temperature Controller and the version for the device is 1. Then this is how its deviceType will look like : *urn:schemas-upnp-org:device:Temperature Controller:1*
friendly Name	Yes	A short user-friendly description of the device. For example, *EBS Temperature Controller* is the friendly title for EBS's temperature controller device.
manufacturer	Yes	The name of the manufacturer
manufacturerURL	No	The URL of manufacturer's website. This URL may be relative to the base URL.
modelDescription	No	Text describing the device for an end user.
modelName	Yes	The model name for the device.
modelNumber	No	If the model of the device has a model number, then this number will go here.
modelURL	No	URL of the website for this model of the device.
serialNumber	No	The serial number for the device or model.
UDN	Yes	The Unique Device Name for the device. This is the unique identifier for the device that must remain unchanged even when the device reboots. This field must begin with "**uuid:**" which should be followed by a unique device identifier. Here is an example of the value of UDN in a device descriptor: *uuid:3de8ae7e-a535-4a50-a689-a6d8923fb73f*
UPC	No	Universal Product Code. If the device has a UPC code which is a 12 digit device code, used to identify consumer package.

IconList	If Available	IconList contains list of icons that are associated with the device that can be used by control point user interfaces to represent the device. IconList and elements under it should be included in the device description document only if the application programmer plans on associating icons with the device. The next 5 elements are needed only if inconList is used in the device description document.
mimetype	Yes	Single MIME image type or format
Width	Yes	Width of the icon in pixels
Height	Yes	Height of the icon in pixels
Depth	Yes	Number of color bits per pixel
url	Yes	URL to access the icon image. May be relative to the base URL.
serviceList	If Available	Service List contains the list of all services that the device offers for eventing and control.
service	If Available	Contains the complete information for a service. The subtypes for service are described below.
serviceType	Yes	UPnP service type. For a standard service types that is defined by UPnP Forum working committee, use the following format: urn:*schemas-upnp-org**:service:***serviceType:v**, where the serviceType the type as defined by the committee and v is the integer indicating service version. For example: *urn:schemas-upnp-org:service:TemperatureService:1*
serviceId	Yes	Unique identifier for this service. No two services can have the same serviceId within the device description. This element has the following format: urn:upnp-org:serviceId:**serviceId**, where the bold serviceId indicated the identifier for the device. For example : *urn:upnp-org:serviceId:TemperatureService.0001_* the serviceID for this service is 'TemperatureService.0001'_
SCPDURL	Yes	The URL for service description. This URL may be relative to base URL. For example, *_TemperatureService.0001_scpd.xml*
controlURL	Yes	The URL at which the device receives control messages for a service. May be relative to base URL. For example, *_TemperatureService.0001_control*
eventSubURL	Yes	The URL for event-related messages for a service. May be relative to base URL. No other service of a device can use the same event URL. This is a required field and should be listed even in case that the service contains no evented state variable. An example of this field is: *_TemperatureService.0001_event*

deviceList	If Available	List of all embedded devices that the device contains. The <device> sub element of this list contains details of an embedded device. The format of <device> is same as that of root device. There is one device sub element for each embedded device. The <deviceList> element is required if and only if the root device has embedded devices.

Table 5.1 – Device Sub-Elements

5.1.2 Creating a Service Description Document

A UPnP device can host one or more services each dedicated to perform a set of tasks or actions which may be remotely invoked by a control point. A service can also allow the control point to subscribe to it in order to receive notifications of any change occurring on the state of this service.

The device must maintain a service description document for each of such service that it hosts. A service description document contains all the information a control point needs to perform eventing and control on a service.

Shown below is a sample of standard format of a service descriptor. Process of creating a service description document is similar to creating a device description document. The developer should replace the italicized text in the sample below with service specific information. Table 2 contains detailed information about each field which is filled with an italicized description in the sample. Refer to table 2 to determine the correct / allowed value to fill for each of these fields.

Tag ‘scpd’ in a service description document identifies this as a service description document. Table 5.2 explains the values for the standard tags or elements which are needed to create this document.
Shown below is a sample of standard format of a service descriptor.

```
<?xml version="1.0"?>
<scpd xmlns="urn:schemas-upnp-org:service-1-0">
  <specVersion>
    <major>1</major>
    <minor>0</minor>
  </specVersion>
  <actionList>
    <action>
    <name>actionName</name>
      <argumentList>
        <argument>
          <name>formalParameterName</name>
          <direction>in xor out</direction>
          <retval />
          <relatedStateVariable>stateVariableName</relatedStateVariable>
        </argument>
        Declarations for other arguments defined by UPnP Forum working
        committee (if any)go here
      </argumentList>
    </action>
    Declarations for other actions defined by UPnP Forum working
    committee (if any)go here
    Declarations for other actions added by UPnP vendor (if any) go here
  </actionList>
  <serviceStateTable>
    <stateVariable sendEvents="yes">
      <name>variableName</name>
      <dataType>variable data type</dataType>
      <defaultValue>default value</defaultValue>
      <allowedValueList>
```

```
      <allowedValue>enumerated value</allowedValue>
      Other allowed values defined by UPnP Forum working committee (if
      any) go here
    </allowedValueList>
  </stateVariable>
  <stateVariable sendEvents="yes">
    <name>variableName</name>
    <dataType>variable data type</dataType>
    <defaultValue>default value</defaultValue>
    <allowedValueRange>
      <minimum>minimum value</minimum>
      <maximum>maximum value</maximum>
      <step>increment value</step>
    </allowedValueRange>
  </stateVariable>
  Declarations for other state variables defined by UPnP Forum working
  committee(if any) go here
  Declarations for other state variables added by UPnP vendor (if any)
  go here
 </serviceStateTable>
</scpd>
```

Source: www.upnp.org/

Element	Required	Value
ActionList	If Available	List of all the actions that the service can perform. Each action is described inside <action> element. The action element contains a name for the action and an argument list which is list of arguments that this action may have
Name	Yes	This is the name of the action. Does not support “-” and “#” character.
argumentList	Yes/No	Argument list must be included if action has arguments associated with it. Each action may have zero or more arguments in it. An argumentList lists all the arguments that are available for the action. This element contains 4 sub elements which are described below.
Name	Yes	Name of a formal parameter. Should be less than 32 characters and not contains a hyphen character. This element should be a name of a state variable which is a function of action.
Direction	Yes	An argument may be used as an input parameter or as an output parameter. Determines whether argument is an input or output. Direction can be 'in' or 'out' (not both) indicating whether the argument is used as input or output variable of an action. Requirement is that any 'in' arguments must be listed before any 'out' arguments.
Retval	No	Identifies at most one out argument as the return value. If included must be the first out argument.
relatedStateVariable	Yes	Name of a state variable that is associated with this action.

serviceStateTable	If Available	A service uses state variables to model its state at runtime. This element lists all the state variables available for the service. Each state variable is described within a stateVariable element which is described below.
stateVariable	If Available	Describes a state variable. This element has a required 'sendEvents' attribute which is used to specify whether this state variable can be evented or not. A 'yes' value of sendEvents indicates that on an event, state variable will send event notifications to all its subscribers, while a value of 'No' indicated that no event notification will be send out for this state variable.
Name	Yes	Name of state variable (cannot contain a hyphen character)
dataType	Yes	Data type of the state variable. One of the datatypes as defined by XML Schema, Part 2: Datatypes. For standard state variables data types are defined by the UPnP working committee.
defaultValue	No	Expected, initial value contained in the state variable. Must be consistent with the data type falling within the allowed value list and between the allowed value range which are explained below.
allowedValueList	If Available	Sub element of a stateVariable. Used to specify the values a variable can take. This element contains the entire list of possible values a variable can take which are specified inside allowedValue tag
allowedValue	YES	This sub - element is required if allowedValueList is used. Specifies a legal value for the variable
allowedValueRange	If Available	Range of acceptable values that can be specified for a numeric variable.
Minimum	No	Lower limit of the range.
Maximum	No	Upper limit of the range.
Step	No	Number of steps or difference between any two values.

Table 5.2 – Service Sub-Elements

5.2 Writing Your Application

UPnP stack needs to access state of the device at all times during the lifetime of the device. UpnPDeviceRuntime structure shown below is designed to hold the runtime information of device. Another structure UpnPRuntime holds general runtime information for the SDK. Although these structures are used internally, the application developer will need to initialize and populate these structures using supplied API's.

These structures are described briefly below,

```
struct s_UPnPDeviceRuntime
{
                                        DLListNode          rootDeviceList;
                                        UPnPRuntime*        upnpRuntime;
                                        UPNP_INT32          announceFrequencySec;
                                        UPNP_INT32          nextAnnounceTimeSec;
                                        UPNP_INT32          remoteCacheTimeoutSec;
                                        int (*announceAll) (UPnPDeviceRuntime *runtime);
                                        SSDPCallback        deviceSSDPCallback;

 #ifdef UPNP_MULTITHREAD
                                        RTP_MUTEX           mutex;
 #endif
};

struct s_UPnPRuntime
{
                                        SSDPServerContext   ssdpContext;
                                        HTTPServerContext   httpServer;
                                        UPnPDeviceRuntime*  deviceRuntime;
                                        UPnPControlPoint*   controlPoint;
                                        UPNP_INT16          ipType;
  #ifdef UPNP_MULTITHREAD
                                        UPnPDaemonState     daemonState;
                                        RTP_MUTEX           mutex;
  #endif
};
```

5.2.1 Initializing and Setting up UPnP Runtime

UPnP application must start by initializing and setting up UpnPRuntime structure which holds runtime information for UPnP stack using UpnP_RuntimeInit ( ) API, for example,

```
result = UpnP_RuntimeInit ( &upnpRuntime, serverAddr, serverPort, ipType, "c:\\www-root\\", maxConnections, maxHelperThreads );
```

where, address of uninitialized UpnPRuntime structure is supplied as the first parameter, this structure will hold sdk's runtime information. Argument serverAddr is the server (device) IPaddress. The format for serverAddr is: In case of an Ipv4 address is ServerAddr [] = {0,0,0,0}; In case of ipv6 a char string holding ipv6 address can be supplied for example ServerAddr [] = {"fe80::20b:dbff:fe2f:c162"}
It is important to note that this value should be same as the device IP address as filled in the URLBase field of device description document. If device IP address is not known pass 0 or NULL as serverAddr parameter. In this case it is a must to turn on AUTOIP (to automatically detect device's address) when registering the device (this is done at a later stage and is explained in this documentation in registering root device section - 5.2.4). Argument serverPort is the port number on which web server serves http requests. This number should be same as mentioned in the URLBase field of the device description document. If a zero is supplied as serverPort and AutoIP is enabled while registering the root device as explained later then a port number is automatically assigned to the web server. ipType is the version of underlying ipstack (ipv4 or ipv6) on which this upnp stack will run. The valid values for ipType are RTP_NET_TYPE_IPV4 for ipv4 network or

RTP_NET_TYPE_IPV6 for an ipv6 network. Next argument is supplied to setup the root directory of the internal web server. In this example, string *"c:\\www-root\\"* is complete path to the directory which will be set as root directory of stack's internal web server. This must be same directory where description documents for the device and its services are stored. The next argument, maxConnections indicates the maximum number of connections that the internal web server will handle at any given time.
In the multithreaded mode, UPnP sdk will spawn at least 2 threads, one for web server and one for the ssdp server. The last argument maxHelperThreads specifies the maximum number of helper threads that the http (web) server can spawn when needed. If maxHelperThreads is equal to n then, maximum threads that sdk can spawn is 2 + n.

5.2.2 Initializing UPnP Device

Next, the application developer must initialize the runtime structure for upnp device. The device maintains its own runtime states and values which needs to be initialized before starting it up. Use UpnP_DeviceInit ( ) in-order to initialize the device runtime structure, for example:

```
result = UpnP_DeviceInit (&deviceRuntime, &upnpRuntime);
```

As first parameter, supply address of an uninitialized buffer of type UPnPDeviceRuntime, this will hold the device runtime information. The address of variable of type UpnpRuntime which was initialized in the previous step is supplied as the second argument.

5.2.3 Loading the Description Documents

As the next step, load the DOM (Data Object Model) representation of device description document using ixmlLoadDocument( ), an IXML parser module API function. Prepare a variable of type IXML_Document to hold the address of this dom tree. In the example below, xmlDevice holds the dom tree representation of device description document 'device.xml', stored at ' c:\\www-root'.

```
xmlDevice = ixmlLoadDocument ("c:\\www-root\\device.xml");
```

This function takes the full path to the device description document as an argument. The DOM tree representation is used by the SDK to extract and set device and service information.

5.2.4 Registering Root Device

Next step is to register the root device using UPnP_RegisterRootDevice( ) API provided. Shown below is an example that uses this API

```
result = UPnP_RegisterRootDevice (&deviceRuntime, "device.xml", xmlDevice, AUTO_ADDR, deviceCallback, 0, &rootDevice, ADVERTISE);
```

Address of variable of UpnPDeviceRuntime type which holds device's runtime information is supplied as the first parameter. Argument 2, "device.xml", is the file name of the root device description document for this example. Using this file name and the base url available in the description document, address of whose DOM tree representation is supplied in argument 3, this API internally creates a full url to access the device description document.
If AUTO_ADDR is set to 1 this function automatically discovers the address of the device, using AutoIP, and uses this address to create the path to the description document. If AUTO_ADDR is 0 then the device address is extracted from the device description document.
Next pass the address of the application callback function, this is the function which will be invoked when control points make any action or subscription requests to the device. The application callback is handles any asynchronous requests from the control point. Section 6 describes the role of an application callback in detail. The next argument is the callback data or cookie that can be passed from the application to the callback function. This API sets up and initializes an un-initialized handle to this device. This handle is used internally by the sdk and also to send advertisements (in the next sections), in this example, rootDevice is of type UpnPRootDeviceHandle and its address is passed to this API. The last parameter gives the developer ability (or flexibility) to control advertising on the device. If ADVERTISE is turned on (set to 1) the device will be set up to send periodic presence announcements. This ability can be turned off by setting ADVERTISE to 0.

5.2.5 Device Advertisement Settings

If last parameter of UPnP_RegisterRootDevice( ) is set to 1, the device will send periodic alive announcements for itself, it's services and any embedded devices. The next API is used to set up advertisement parameters like frequency of sending periodic advertisements, and the time period for which the client (control point) will cache the advertisement. Here is an sample usage of this API,

```
UpnP_DeviceAdvertise (rootDevice, announceFreq, cacheTimeOut);
```

This API sets up the device with device handle 'rootDevice' (argument 1 above, initialized in step 5.2.4), to send advertisements every 'announceFreq' seconds and setting the remote recipient to cache the information for cacheTimeOut seconds. When this device shuts down it will send a bye – bye notification to all the devices and control points present on the network.

5.2.6 Starting up the Device

Calling UpnP_DeviceStartDaemon ( ) starts up the device in multithreaded environment.

```
UpnP_DeviceStartDaemon (&upnpRuntime);
```

Where, address of UPnP's runtime is supplied as parameter.

The developer can choose to run the device in polled mode by calling UpnP_ProcessState ( ) API.

```
UpnP_ProcessState(&upnpRuntime, pollTimeMsec);
```

This API gets polled every pollTimeMsec milliseconds.

5.2.7 Application Body Implementation

Now that the UPnP engine has started, insert the body of application here. If a value associated with any service hosted by the device is changed inside the application an event reflecting this change needs to be sent to all the subscribers. To see how to do this, see section 'Sending event notifications from within an application'.

5.2.8 Shutting Down the Device

API UpnP_DeviceFinish ( ) is designed to stop device, by killing the threads and freeing up their resources in a multithreaded environment. This function frees up the resources used by the device by un-registering the device, sends bye-bye notification for the device and its services and shuts down the ssdp server and the internal web server.

```
UpnP_DeviceFinish (&deviceRuntime);
```

Where, address of device's runtime structure/ buffer is supplied as parameter.
Finally calling IXML parser API ixmlDocument_free ( ) frees up the DOM tree representation of the description document.

```
ixmlDocument_free (xmlDevice);
```

In this example, xmlDevice is the variable holding address of device description document which was assigned previously in section 5.2.3

Section 6: Writing a device Callback

Why a device callback?
A Device callback function needs to be implemented to handle a SOAP/Action request and to handle subscription requests directed to the device or its services. Shown below is example from the sample device.

To make things even clear, Figure 4 describes the device application, device callback and the control point interaction

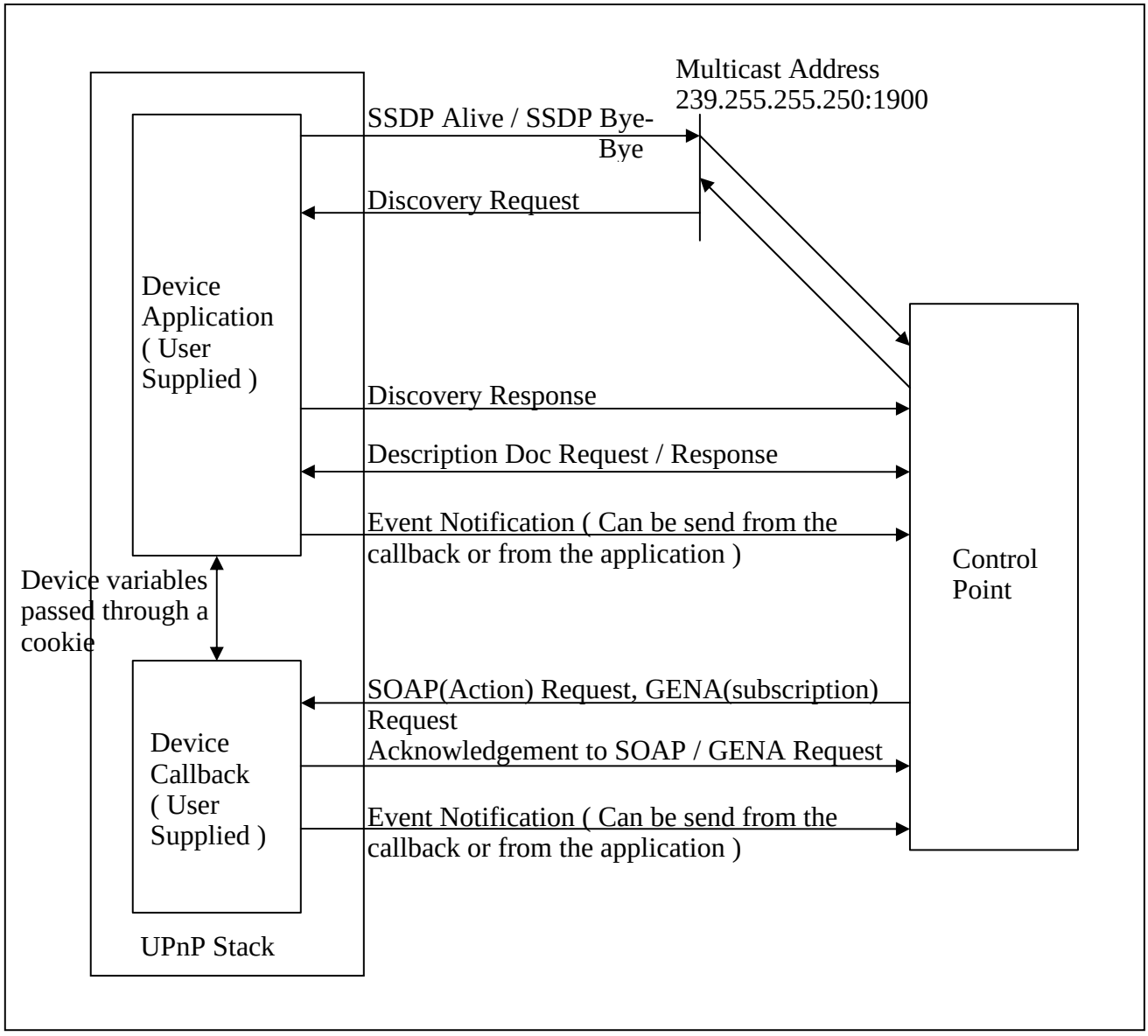

Figure 4 Interaction between device application, device callback and control point.

```
DeviceCallback (&userData, &deviceRuntime, rootDevice, operationType,
&operationStruct);
```

Here, userData is a cookie (callback data) which is passed into the callback from the application. This is the same cookie which was passed in the UpnP_RegisterRootDevice API when registering the device. Second parameter deviceRuntime is address of structructure holding device information at runtime. Argument rootDevice is the handle to the current device.

OperationType indicates what type an operation this callback will perform, operationType is of type “UpnPDeviceEventType” and can indicate one of the three event types as indicated below.

```
typedef enum e_UPnPDeviceEventType
{
                                        UPNP_UNKNOWN_DEVICE_EVENT = -1,
                                        UPNP_DEVICE_EVENT_ACTION_REQUEST,
                                        UPNP_DEVICE_EVENT_SUBSCRIPTION_REQUEST,
                                        UPNP_NUM_DEVICE_EVENT_TYPES
}
UpnPDeviceEventType;
```

An 'UPNP_DEVICE_EVENT_ACTION_REQUEST' operationType indicates control action request, while 'UPNP_DEVICE_EVENT_SUBSCRIPTION_REQUEST’ indicates a subscription request.

The purpose of a callback is to either handle an action request or to handle a subscription request. The first step of callback operation should be to determine whether the request that invoked the callback is an action request or a request for subscription. OperationType holds the information needed to resolve the type of request.

Shown below is a sample usage:

```
switch (operationType)
{
   case UPNP_DEVICE_EVENT_ACTION_REQUEST:
   {
      UPnPActionRequest *request = (UPnPActionRequest *) eventStruct;
      ..........................................................
      ..........................................................
   break;
   }
   case UPNP_DEVICE_EVENT_SUBSCRIPTION_REQUEST:
   {
      UPnPSubscriptionRequest *request = (UPnPSubscriptionRequest *) eventStruct;
      ..........................................................................
      ..........................................................................
      ..........................................................................
      break;
   }
}
```

The callback handles action requests and subscription requests differently which makes it necessary for the developer to classify the two cases separately as shown above. The last parameter, operationStruct, holds the actual request information. This parameter should be type-casted to be of form UPnPActionRequest * in case of action request whereas in-case of subscription request, eventStruct should be typecasted to be of type UPnPSubscriptionRequest *.

6.1 Serving an Action Request

Implementation of a callback to handle action requests roughly involves four steps
1. Checking the request to determine the target service and target action on the device
2. Performing the requested action.
3. Sending acknowledgement / response indicating success or failure to the requester.
4. Sending event notification to all the subscribers of the service indicating any change which may have occurred to service arguments or its state variables as a result of this action.

If the request is an action/control request, then, operationStruct (last parameter in the callback function) is a structure of 'UpnPActionRequest' type. The first step therefore is to typecast operationStruct to be able to hold address of type 'UpnPActionRequest', here is an example

```
UPnPActionRequest *request = (UPnPActionRequest *) operationStruct;
```

6.1.1 Operation

Start by extracting the device name supplied in the request, checking this supplied unique device name against the unique device name (UDN) of your device will determine if the request is targeted to your device. Example,

```
targetDeviceName= UPnP_GetRequestedDeviceName(operationStruct, operationType);
```

here, operationStruct is passed to the callback and operationType is the fourth parameter of the callback which indicated the nature of the request (action or subscription).

Request from a control point holds the service Identifier (serviceId) of the target service containing the target action. Since a device can host one or more services, the developer needs to extract the supplied serviceID from the request message to determine which service this request is targeted to.
Example,

```
targetServiceId = UPnP_GetRequestedServiceId(operationStruct, operationType);
```

Where, operationStruct and operationType are same as described in the previous call.

Once the target service is located the next step is to determine or extract the target action requested by control point. A service can offer one or more actions, it is therefore important to ascertain which action this request wants to invoke. The action name is extracted from the request using UpnP_GetRequestedActionName API, example,

```
targetActionName = UPnP_GetRequestedActionName(operationStruct, operationType);
```

It is important to note that each action holds an argumentList which is a listing of all the arguments that the action contains. Each argument has an associated direction ('in or 'out') which classifies the argument as an input or an output parameter. For example, In-arguments are passed to a service when an action is invoked, while out arguments return values as a result of the action.

Now, with requested action name known, if the action has arguments having 'in' direction, next step is to extract the value of such arguments supplied in the action request. For each 'in' argument use UpnP_GetArgValue ( ) API to extract its input(supplied) value. Shown below is an example on how to extract the supplied value of an argument.

```
UPNP_CHAR * newValue = UpnP_GetArgValue (request, "newTargetValue");
```

where, request which is of type UpnPActionRequest, is the address of buffer holding the action request data. String 'newTargetValue' is an argument of direction 'in' of the requested action which is obtained from visually inspecting the service decription document of the requested service.

An important point to note here is that the request contains an argument and its value if and only if the target action contains an 'in' argument. Also, UPnP specifications require the request to contain every 'in' argument in the defination of the action in the service description.

Keeping the above point in mind the developer should extract the supplied value of all the 'in' arguments of the target action and should not try to extract the value of 'out' argument of an action as 'out' argument are not part of the action request.

Next step is to perform requested action with the new input values for the arguments. If the service is not designed to accept and operate upon this value, or if the service is not able to complete this action, it needs to send an error response message to the control point. This can be done using UpnP_SetActionErrorResponse API. For example,

```
UPnP_SetActionErrorResponse(request, "Invalid Action", 402);
```

where, request pointer to action request, "Invalid Action" is the string describing the error and 402 is the error code. The string describing the error and the numerical error code are specified by UPnP forum. The possible values for argument 2 and 3 for the above call are listed in table 6.1 shown below

Error Code	Error Description	Description
401	Invalid Action	The service has no action of the name provided.
402	Invalid Args	Problem with supplied input arguments; not enough arguments, too many arguments, wrong name or wrong type.
403	Out of Sync	Out of synchronization
501	Action Failed	Current state of service prevents invoking the action.
600 – 699	UPnP Forum defined	Common action errors as defined by UPnP Forum.
700 – 799	Depends on device type	Action-specific errors for standard actions. Defined by UPnP forum working committee.
800 – 899	Vendor-defined	Action-specific error for non standard actions. Available to be defined by the UPnP device vendor.

If action is performed successfully, two steps will need to be taken

1. If the operation resulted in changing the value of any of the services state variables, the developer needs to send an event notification to all the subscribers of the service which contains this action indicating the new changes.

2. Send a response to the requesting client indicating successful operation, the message must contain name and value of all the 'out' arguments of this action.

Lets look at implementation details of these two steps. If the completion of action results in changes to the state of evented state variables included in the service, all these changes should be notified to all the subscribers.

To create an event notification, start by creating a variable of IXML_Document type which will be used to point to the event message (internally send as an envelop called property set) , initially this variable point to NULL. For example,

```
IXML_Document *propertySet = 0
```

Next step is to initialize and populate the event notification message. In order to do so, the evented state variables which were affected following the action operation have to be added to the event message along with their current value. Each call to UpnP_AddToPropertySet API adds a variable and its new value to the event notification message. Here is an example,

```
UpnP_AddToPropertySet (&propertySet, "Status", value);
```

Where, 'status' is the name of the variable and 'value' is its new value.

Event notification message is now ready to be send out, two API are available UpnP_DeviceNotify ( ), a blocking call, or UpnP_DeviceNotifyAsync ( ) for non blocking call, to all the subscribers

```
UpnP_DeviceNotify (deviceRuntime, rootDevice, deviceName, serviceId, propertySet);

UpnP_DeviceNotifyAsync (deviceRuntime, rootDevice, deviceName, serviceId, propertySet);
```

The XML document created for sending event notifications can now be freed, this can be done using IXML API ixmlDocument_free ().

```
ixmlDocument_free (propertySet);
```

Finally, the operation ends with creating and sending a response to the requesting client indicating successful completion of action. The call shown below creates the response message which will be send to the client.

```
UpnP_CreateActionResponse (request);
```

Where, request is a pointer to the action request.

Again important to note that UPnP specifications require the response to contain name and value of every 'out' arguments of the target action. If action has an argument marked as retval, this argument must be the first element. The order inclusion must be same as specified in service description.

Keeping the above point in mind the developer needs to insert name and value of 'out' arguments of the action to the response message. As described above if the action has an argument marked as retval insert this argument and its value first. In order to insert an argument and its value to the response message use *UpnP_SetActionResponseArg* API every time a name value pair for an argument needs to be inserted.
For example,

```
UPnP_SetActionResponseArg(request, "RetLoadLevelStatus", argValue);
```

Where, request is a pointer to the action request. String '*RetLoadLevelStatus*' is an argument of direction 'out' in the target action which is obtained from the XML service decription document of the requested service, argValue is the current value of '*RetLoadLevelStatus*' argument.

Once the response message is created to indicate success or error, it is send internally by the SDK.

6. 2 Serving a Subscription Request

If the request is a subscription request, then callback's last parameter, operationStruct, holds subscription request information in form of 'UPnPSubscriptionRequest' type.
The first step to handle a subscription request is to typecast operationStruct to hold address of 'UPnPSubscriptionRequest', here is an example

```
UPnPSubscriptionRequest *request = (UPnPSubscriptionRequest *) operationStruct;
```

6.2.1 Operation

Check device name and serviceId to see if the subscription request targets your device and one of its services. To do this extract the device name supplied in the request, check this supplied unique device name against the unique device name (UDN) of your device. Use *UpnP_GetRequestedDeviceName* API to obtain the target device name from the request, for example

```
targetDeviceName= UPnP_GetRequestedDeviceName(operationStruct, operationType);
```

here, operationStruct holds the request information and operationType is the fourth parameter of the callback which indicated the nature of the request (action or subscription).

The service Identifier (serviceId) is extracted using *UpnP_GetRequestedServiceId* API, here is an example,

```
targetServiceId = UPnP_GetRequestedServiceId(operationStruct, operationType);
```

Where, operationStruct and operationType are same as described in the previous call.

In-order to send a response to the subscription request, an xml document called property set, which hold the body of the response, will be next created. Create a variable of type IXML_Document which is used to point to the property set, initially this variable point to NULL. For example,

```
IXML_Document *propertySet = 0
```

Next step is to populate property set by adding state variables, whose sendEvents is set to "yes" in the target service's description document. For each name value pair for a state variable use UpnP_AddToPropertySet ( ) API to add a state variable and its current value to the property set. Here is an example,

```
UpnP_AddToPropertySet (&propertySet, "Status", value);
```

Where, "status" is the name of the variable and value is its current value.

Next step is to accept the new subscription request using UpnP_AcceptSubscription ( ) or UpnP_AcceptSubscriptionAsync ( ) - for non blocking call. These APIs internally add a new subscriber to the service's subscriber list, generates a unique subscription Id for this subscriber, set a duration in seconds for the subscription to be valid and sends a subscription response indicating success or failure of subscription request. Here is an usage example,

```
UpnP_AcceptSubscription (request, 0, 0, propertySet);
```

```
UpnP_AcceptSubscriptionAsync (request, 0, 0, propertySet);
```

Where, request point to a structure of type 'UpnPSubscriptionRequest' holding request information. 0 is passed for subscriptionId and subscription duration timeout in seconds indicating that a subscriptionId and a timeout needs to be generated and set for the response, propertySet pointer to xml document containing the main body (<property set>) of the notify Message.
Final step is to free the IXML document you created earlier using IXML library API ixmlDocument_free ().

```
ixmlDocument_free (propertySet);
```

Section 7: Sending Event Notifications from within Device Application

This section explains the techniques and scenarios in which you will need to send event notifications from outside the above callback OR from inside an application body.

What is a subscribed state variable?
You may want the control point (client) to be able to subscribe to some of the services offered by your device. Any change to the state of such a service needs to be notified to all the available subscribers. The services that you want to be available to be subscribed should be listed as a service tag in the device description document.
A service description xml document needs to be prepared for this service. The 'serviceStateTable' tag inside this description document should list all the state variables available for this service. Among these state variables those which have sendEvents set to yes in the description document can be subscribed. The control point can subscribe to these state variables in order to monitor the state of a service.

When do you send event notification from an application?
If you application changes the value of any state variable that could be subscribed than it needs to send an event notification to all the subscribers of the variable indicating the new value.

How to send these notifications?
To create an event notification, start by creating an empty IXML document to store the property set, <propertyset> element is the container for the current state of the service

```
IXML_Document *propertySet = 0
```

Use UpnP_AddToPropertySet( ) API to add a state variable and its new value to the property set. Here is an example,

```
UpnP_AddToPropertySet (&propertySet, "Status", value);
```

Where, 'status' is the name of the state variable and 'value' is its new value.

Once the property set is filled, we have our event notification message ready to be send out, using UpnP_DeviceNotify ( ), to all the subscribers

```
UpnP_DeviceNotify (deviceRuntime, rootDevice, deviceUDN, serviceId, propertySet);
```

Where, device UDN is the uuid of the device available in UDN tag of the device description document and serviceId is the unique service Id as in serviceId tag of the device description document.

Now that you are done with the IXML document you created earlier, you can free it using IXML API ixmlDocument_free ( ).

```
ixmlDocument_free (propertySet)
```

PART II – Porting and Configuration Guide

EBS UPnP Device SDK

Porting and Configuration

Revised July 2006

UPnP SDK Source Code Structure

The UPnP SDK source code package is comprised of 4 independent modules

1. Ixml
2. Http
3. Upnp
4. Rtplatform

UPnP SDK's core files located in upnp modules utilizes ixml module for xml operations, http module for http server and http client operations and rtplatform for providing abstraction from underlying software / hardware platform.

The following table describes the directory structure and includes comments specific to individual subdirectories in the release tree. Subsequent sections of this document provide greater where it is necessary.

Module	Subdirectory	Description	Comments
UPnP	include	Core UPnP header files	Header files common to upnp device and control point are located in include/ Device specific header files are located in include/device/ Control point specific header files are located in include/controlPoint/
	source	Core UPnP ".c" files	".c" file common to device and control point are located in source/ Device specific ".c" source file are located in source/device Control point specific ".c" source file are located in source/controlPoint
	project	Sample win32 projects	Two sample visual studio projects are available, one for device and one for control point. These projects are ready to be built and demo programs can be executed. Note: Sample Linux projects are available upon request.
	doc	UPnP reference documentation in html format	Device API reference in html format can be accessed through doc/devicehtml/index.html Control Point API reference in html format can be accessed through doc/controlpointhtml/index.html
ixml	inc	XML parser module header files	Header files for xml parser
	src	XML parser module ".c" source files and header files	Contains ".c" files used by xml parser. This directory also contains some header files used by xml parser which are located in src/inc directory
	doc	XML parser documentation in html	XML parser's internal API reference in html format which can be

		format	accessed through doc/html/index.html
http	include	HTTP server and client module header files	Contains header files used by HTTP server and HTTP client. httpp.h - contains module wide definitions and declarations used by both server and client. httpsrv.h – header file for http server httpcli.h – header file for http client httpmcli.h – header file for http managed client
	source	HTTP server and client module ".c" source files	Contains ".c" source files used by http server and http client. Http files used by upnp device Fileext.c Filetype.c Httpp.c Httpsrv.c Http files used by upnp control point: Fileext.c Filetype.c Httpp.c Httpsrv.c Httpmcli.c Httpcli.c Urlparse.c
	doc	Http module specific reference documentation in html format	Http module's internal API reference in html format which can be accessed through doc/html/index.html
Rtplatform	include	RTPlatform header files	All header files in /include directory are common to all platforms, operating systems, network stacks etc. Note: In particular case when a common header file is not suitable for a platform, such header files may be located in platform specific directory within include directory. For example, linux specific rtpprint.h is located in include/linux and win32 specific rtpprint.h is located in include/ms32/rtpprint.h
	source	Platform specific RTPlatform source files.	Contains platform specific source files for rtplatform library. All source files ported to a platform are located in a subdirectory named after that platform. For example, RTPlatform source files ported to win32 platform are located in source/win32/ While source files ported to linux platform are located in source/linux/ and so on.

			Two special subdirectories are available 1. Generic – source/generic All source files located in generic are platform independent implementation of underlying routine. In some cases it is desirable to use platform specific file rather than a generic file, in such case a file from source/generic may be ported and placed in platform specific subdirectory. For example, file rtpdate.c has a generic version located in source/generic/rtpdate.c but a win32 specific version of this file is located in source/win32/rtpdate.c and a linux specific in source/linux/rtpdate.c 2. Template – source/template Files under template contain stubbed function with detailed comments and explanation on how to port the function. Files under template provide a good starting point for creating a fresh port for a platform. Following RTPlatform source files are used by UPnP SDK Rtpchar.c Rtpdate.c Rtpdebug.c Rtpdobj.c Rtpdutil.c Rtpfile.c Rtphelper.c Rtpmem.c Rtpnet.c Rtpnetsm.c Rtpprint.c Rtprand.c Rtpscnv.c Rtpsignl.c Rtpstdup.c Rtpstr.c Rtpthrd.c Rtptime.c
	doc	RTPlatform reference documentation in html format	RTPlatform reference documentation in html format can be accessed through doc/html/index.html

Configuring UPnP SDK

UPnP requires zero configurations. All configuration values are passed as options in UPnP runtime and device / control point initializations APIs.

By default UPnP is configured to run in multitasking mode. To turn off multitasking comment the following definition in /include/upnp.h

#define UPNP_MULTITHREAD

Porting UPnP SDK

Porting EBS UPnP software development kit to alternate platforms simply requires creating a port in rtplatform to implement the operating system, network stack, file system, and timing functions.

RTPlatform is EBS's cross-platform runtime environment. It defines an interface between the high-level platform-independent code, UPnP SDK in this case, and the lower-level operating system/hardware environment.

RTPlatform is divided into a number of modules, each providing an interface to a specific service. For example, there is the rtpnet module, which defines a sockets-style interface to TCP/IP networking services, and rtpfile, which defines a roughly POSIX-style interface to file system services.

Some of the RTPlatform modules have platform-independent, or generic, implementations; others must rely on platform-specific code for their implementation (these are the environment-specific or non-portable modules). The release distribution of RTPlatform may include many different versions of the non-portable modules, each in a different directory that indicates the target environment. For example, the rtpnet module has an implementation for Linux's TCP/IP stack, in "source/linux/rtpnet.c", and an implementation for the Winsock library on 32-bit Microsoft Windows environments in "source/win32/rtpnet.c". All the ".c" files in "source/generic/" directory are platform-independent and may not require any porting. Template directory "source/template" located in the source tree provides an excellent starting point for creating any platform specific "rtpxxx.c" file.

Although there may be many ".c" files (one for each target) corresponding to a particular module, there is usually only one ".h" file, located in the "include" directory. Therefore, because all of the header files are platform-independent, there are often no environment-specific header files included by UPnP source files. This greatly simplifies the porting process because it eliminates any potential symbol/namespace collisions.

Following rtplatform files are used by EBS UPnP SDK –

Rtplatform Source File	Generic Version Available
Rtpchar.c	Yes
Rtpdate.c	Yes
Rtpdebug.c	Yes
Rtpdobj.c	No
Rtpdutil.c	Yes
Rtpfile.c	No
Rtphelper.c	Yes
Rtpmem.c	No
Rtpnet.c	No

Rtpnetsm.c	Yes
Rtpprint.c	Yes
Rtprand.c	Yes
Rtpscnv.c	Yes
Rtpsignl.c	No
Rtpstdup.c	Yes
Rtpstr.c	Yes
Rtpthrd.c	No
Rtptime.c	No

Note: Even though generic version of some file may be available it may still be required to port them to suit a particular platform needs (e.g. rtpdate.c). Please see windows and linux ports of rtplatform as an example of proper porting.

For more details on implementing an rtplatform port for your software / hardware platform please see rtplatform user guide.

PART III - EBS UPnP Device API Reference Manual

EBS UPnP Device SDK

API Reference Manual

Revised July 2006

Section 1: Introduction

Universal Plug and Play (UPnP) is an open networking architecture for peer to peer network connectivity of UPnP enabled devices. UPnP provides a device the capability to discover and control other devices on a network. Devices act as servers providing the clients, known as control points, access and control to its published capabilities. Control points have the ability to invoke actions on any UPnP device on a network, control points can also subscribe to a device to continuously monitor the state of a device and its services.

UPnP architecture builds on existing networking protocols, such as IP, TCP, UDP, HTTP, HTML, SOAP, SSDP, GENA etc. and web standards like XML to make the communication and control possible. Any device having a TCP/IP network stack is capable of running UPnP regardless of its underlying operating system and hardware.

Section 2: EBS UPnP Device API

API	Description
UPnP_RuntimeInit	Initialize a UpnPRuntime
UPnP_RuntimeDestroy	Destroy a UpnPRuntime
UPnP_AddVirtualFile	Create a virtual file on the HTTP server.
UPnP_RemoveVirtualFile	Remove a virtual file from the server
UPnP_ProcessState	Process asynchronous operations in non-threaded mode.
UPnP_StartDaemon	Start the UPnP Daemon thread
UPnP_StopDaemon	Kill the UPnP Daemon thread
UPnP_DeviceInit	Initialize a UpnPDeviceRuntime
UPnP_DeviceFinish	Destroy a UpnPDeviceRuntime
UPnP_RegisterRootDevice	Configures the root device and its serives for UpnP
UPnP_UnRegisterRootDevice	Free root device from its server bindings
UPnP_DeviceAdvertise	Set up the device to send periodic SSDP announcements
UPnP_DeviceNotify	Sends an event notification message to all the subscribers of the service
UPnP_DeviceNotifyAsync	Sends a non blocking event notification message to all the subscribers of the service
UPnP_AcceptSubscription	Accept a new subscription request
UPnP_AcceptSubscriptionAsync	Send Subscription Accept asynchronously
UPnP_GetRequestedDeviceName	Extracts Unique device name for the target device from control/subscription reques
UPnP_GetRequestedServiceId	Extracts service identifier from a control/subscription request
UPnP_GetRequestedActionName	Extracts name of the target action from action/subscription request
UPnP_SetActionErrorResponse	Sets error code and error description for a response to an action request
UPnP_GetPropertyValueByName	Get the value of a named property in a GENA notify message
UPnP_GetPropertyNameByIndex	Get the name of the nth property
UPnP_GetPropertyValueByIndex	Get the value of the nth property

UPnP_AddToPropertySet	Add name and value pair to GENA notify message property set
UPnP_CreateActionResponse	Creates a SOAP action response message
UPnP_CreateAction	Create a SOAP action request
UPnP_SetActionArg	Set an argument for a SOAP action response/request
UPnP_GetArgValue	Extracts the value of a given argument from an action
UPnP_SetActionResponseArg	Inserts name and value of an argument to an action response message

2.1 UPnP_RuntimeInit

FUNCTION

Initialize a UPnP runtime context – 'UPnPRuntime'.

SUMMARY

int UPnP_RuntimeInit (UPnPRuntime* rt, UPNP_UINT8* serverAddr, UPNP_UINT16 serverPort,
UPNP_INT16 ipType , UPNP_CHAR* wwwRootDir, UPNP_INT16 maxConnections,
UPNP_INT16 maxHelperThreads)

UPnPRuntime* rt	Pointer to uninitialized UPnPRuntime struct
UPNP_UINT8* serverAddr	IP address to bind HTTP server to (NULL for IP_ADDR_ANY)
UPNP_UINT16 serverPort	Port to bind HTTP server to (0 for ANY_PORT)
UPNP_INT16 ipType	Type of IP version used (ipv4 or ipv6), (RTP_NET_TYPE_IPV4 for v4 and RTP_NET_TYPE_IPV6 for v6)
UPNP_CHAR* wwwRootDir	HTTP root dir on local file system
UPNP_INT16 maxConnections	The maximum limit on simultaneous HTTP server connections
UPNP_INT16 maxHelperThreads	If UPNP_MULTITHREAD is defined, the max number of helper threads to spawn

DESCRIPTION

Initializes the given UPnPRuntime struct, and sets up an HTTP server instance to receive control/event messages. This function must be called before any other function in the UPnP SDK.

RETURNS

0	Operation was a success
-1	Operation failed

2.2 UPnP_RuntimeDestroy

FUNCTION

Destroy and clean up a UPnPRuntime.

SUMMARY

void UPnP_RuntimeDestroy (UPnPRuntime* rt)

UPnPRuntime* rt	Pointer to UPnPRuntime struct

DESCRIPTION

This function frees all the resources allocated by UPnPRuntime. This function must be called after all other UPnP SDK calls to clean up runtime data for UPnP.

RETURNS

No value

2.3 UPnP_AddVirtualFile

FUNCTION

Create a virtual file on the HTTP server.

SUMMARY

int UPnP_AddVirtualFile (UPnPRuntime* rt, const UPNP_CHAR* serverPath, const UPNP_UINT8* data, UPNP_INT32 size, const UPNP_CHAR* contentType)

UPnPRuntime* rt	
UPNP_CHAR* serverPath	
UPNP_UINT8* data	
UPNP_INT32 size	
UPNP_CHAR* contentType	

DESCRIPTION

Makes the data buffer passed in available at the given path on the HTTP server.

RETURNS

0	Operation was a success
-1	Operation failed

SEE ALSO

UPnP_RemoveVirtualFile ( )

2.4 UPnP_RemoveVirtualFile

FUNCTION

Remove a virtual file from the server.

SUMMARY

int UPnP_RemoveVirtualFile (UPnPRuntime* rt, const UPNP_CHAR* serverPath)

UPnPRuntime* rt	Pointer to UPnPRuntime struct
UPNP_CHAR* serverPath	

DESCRIPTION

This function removes a virtual file from the server. This function must be called before UPnP_RuntimeDestroy to remove any virtual files added using UPnP_AddVirtualFile.

RETURNS

0	Operation was a success
-1	Operation failed

2.5 UPnP_ProcessState

FUNCTION

Process asynchronous operations in non-threaded mode.

SUMMARY

int UPnP_ProcessState (UPnPRuntime* rt, UPNP_INT32 msecTimeout)

UPnPRuntime* rt	Pointer to UPnPRuntime struct
UPNP_INT32 msecTimeout	Time in milliseconds for which this task will block and perform upnp operations.

DESCRIPTION

This function blocks for at most msecTimeout milliseconds, processing any asynchronous operations that may be in progress on either the control point or device runtime attached to the given UPnPRuntime.

This function must be called in order to receive events if an application is running with the UPnP SDK in single-threaded mode.

RETURNS

0	Operation was a success
-1	Operation failed

2.6 UPnP_StartDaemon

FUNCTION

Start the UPnP Daemon thread.

SUMMARY

int UPnP_StartDaemon (UPnPRuntime* rt)

UPnPRuntime* rt	Pointer to UPnPRuntime struct

DESCRIPTION

This function must be called in multithreaded mode to start the UPnP daemon, which listens for requests / announcements on the network, and sends any events to the attached control point / device runtime.

RETURNS

0	Operation was a success
-1	Operation failed

SEE ALSO

UPnP_StopDaemon ( )

2.7 UPnP_StopDaemon

FUNCTION

Kill the UPnP Daemon thread.

SUMMARY

int UPnP_StopDaemon (UPnPRuntime* rt, UPNP_INT32 secTimeout)

UPnPRuntime* rt	Device runtime to stop
UPNP_INT32 secTimeout	Time in seconds time to wait for daemon to stop.

DESCRIPTION

This function stops the UPnP daemon from executing. It will wait for at most secTimeout seconds for all helper threads to terminate. If this function returns negative error code, it means the timeout expired without the successful termination of one or more helper threads. In this case, calling UPnP_RuntimeDestroy may cause a fault since there are still helper threads running that may try to access the data structures pointed to by the UPnPRuntime.

RETURNS

0	Operation was a success
-1	Operation failed

2.8 UPnP_DeviceInit

FUNCTION

Initialize a UPnP device runtime context ‘UpnPDeviceRuntime’.

SUMMARY

int UPnP_DeviceInit (UPnPDeviceRuntime* deviceRuntime, UPnPRuntime* rt)

UPnPDeviceRuntime* deviceRuntime	Pointer to the device runtime buffer
UPnPRuntime* rt	Pointer to an initialized upnp runtime buffer

DESCRIPTION

Initializes all device state data in a UPnPDeviceRuntime struct (allocated by the calling application), and binds the device to the specified UPnPRuntime. The UPnPRuntime must be initialized via UPnP_RuntimeInit before this function is called. Only one device may be bound to a single UPnPRuntime at once. This function must be called before all other device related functions.

RETURNS

0	Operation was a success
-1	Operation failed

SEE ALSO

UPnP_DeviceFinish ( )

2.9 UPnP_DeviceFinish

FUNCTION

Destroy a device runtime context.

SUMMARY

int UPnP_DeviceFinish (UPnPDeviceRuntime* deviceRuntime)

UPnPDeviceRuntime* deviceRuntime	Address of runtime of device to destroy

DESCRIPTION

Cleans up all data associated with a UPnPDeviceRuntime structure. Once this function has been called, it is safe to free the memory used by the UPnPDeviceRuntime structure.

RETURNS

0	Operation was a success
-1	Operation failed

SEE ALSO

UPnP_DeviceInit ( )

2.10 UPnP_RegisterRootDevice

FUNCTION

Configures the root device and its serives for UPnP.

SUMMARY

int UPnP_RegisterRootDevice (UPnPDeviceRuntime* deviceRuntime, const UPNP_CHAR* descDocURL,
IXML_Document* description, UPNP_BOOL autoAddr,
UPnPDeviceCallback callback, void* userData,
UPnPRootDeviceHandle* retHandle, UPNP_BOOL deviceAdvertise)

UPnPDeviceRuntime* deviceRuntime	Pointer to device runtime context
UPNP_CHAR* descDocURL	Relative url of device description document
IXML_Document* description	Address of DOM representation of the device description document
UPNP_BOOL autoAddr	Select swtich for Auto IP if 1 - uses AutoIP if 0 - extracts address from the device description document
UPnPDeviceCallback callback	Pointer to the callback function for the device.
void* userData	User data for callback
UPnPRootDeviceHandle* retHandle	Handle to the current root device
UPNP_BOOL deviceAdvertise	Switch to turn ON and OFF device advertising If 1 - device will be set up to send periodic SSDP announcements. If 0 - no ssdp announcements will be send

DESCRIPTION

Sets up the device to serve UPnP requests from the clients; set up devcive for ssdp announcements if deviceAdvertise is turned on.

RETURNS

0	Operation was a success
-1	Operation failed

2.11 UPnP_UnRegisterRootDevice

FUNCTION

Free root device from its server bindings.

SUMMARY

int UPnP_UnRegisterRootDevice (UPnPRootDeviceHandle rootDevice)

UpnPRootDeviceHandle rootDevice	Handle to root device

DESCRIPTION

Unregisters the root device from the internal server, so that the future UPnP requests will not be served for this root device.

RETURNS

0	Operation was a success
-1	Operation failed

2.12 UPnP_DeviceAdvertise

FUNCTION

Set up the device to send periodic SSDP announcements.

SUMMARY

int UPnP_DeviceAdvertise (UPnPRootDeviceHandle rootDevice, UPNP_INT32 frequencySec,
UPNP_INT32 remoteTimeoutSec)

UPnPRootDeviceHandle rootDevice	Handle to the device
UPNP_INT32 frequencySec	Interval in seconds between two announcements
UPNP_INT32 remoteTimeoutSec	Time in seconds for which the remote client will cache the information in the announcement

DESCRIPTION

This function prepares the device to send periodic announcements every frequecySec seconds.

RETURNS

0	Operation was a success
-1	Operation failed

2.13 UPnP_DeviceNotify

FUNCTION

Sends an event notification message to all the subscribers of the service.

SUMMARY

int UPnP_DeviceNotify (UPnPDeviceRuntime* deviceRuntime, UPnPRootDeviceHandle rootDevice,
const UPNP_CHAR* deviceUDN, const UPNP_CHAR* serviceId,
IXML_Document* propertySet)

UPnPDeviceRuntime* deviceRuntime	Device runtime information
UPnPRootDeviceHandle rootDevice	Handle to the device
UPNP_CHAR* deviceUDN	Unique device identifier (UUID in the device description document) for the device
UPNP_CHAR* serviceId	Unique service identifier (serviceID in the device description document) for the service
IXML_Document* propertySet	Contains the evented variable and its value in XML format.

DESCRIPTION

Sends an event notification message to all control points which are subscribed to service with supplied service ID on this device.

RETURNS

0	Operation was a success
-1	Operation failed

SEE ALSO

UPnP_DeviceNotifyAsync ( )

2.14 UPnP_DeviceNotifyAsync

FUNCTION

Sends a non blocking event notification message to all the subscribers of the service.

SUMMARY

int UPnP_DeviceNotifyAsync (UPnPDeviceRuntime* deviceRuntime, UPnPRootDeviceHandle rootDevice,
const UPNP_CHAR* deviceUDN, const UPNP_CHAR* serviceId,
IXML_Document* propertySet)

UPnPDeviceRuntime* deviceRuntime	Device runtime information
UPnPRootDeviceHandle rootDevice	Handle to the device
UPNP_CHAR* deviceUDN	Unique device identifier (UUID in the device description document) for the device
UPNP_CHAR* serviceId	Unique service identifier (serviceID in the device description document) for the service
IXML_Document* propertySet	Contains the evented variable and its value in XML format.

DESCRIPTION

Sends an asynchronous (non blocking) event notification message to all control points which are subscribed to service with supplied service ID on this device.

RETURNS

0	Operation was a success
-1	Operation failed

SEE ALSO

UPnP_DeviceNotify ( )

2.15 UPnP_AcceptSubscription

FUNCTION

Accept a new subscription request.

SUMMARY

int UPnP_AcceptSubscription (UPnPSubscriptionRequest* subReq, const GENA_CHAR* subscriptionId,
UPNP_INT32 timeoutSec, IXML_Document* propertySet,
UPNP_INT32 firstNotifyDelayMsec)

UPnPSubscriptionRequest* subReq	Address of structure containing subscription request information
GENA_CHAR* subscriptionId	Subscription identifier for the subscriber
UPNP_INT32 timeoutSec	Duration in seconds for which the subscription is valid
IXML_Document* propertySet	Address of response message in XML format
UPNP_INT32 firstNotifyDelayMsec	Delay in milliseconds before sending the first event notification to the new subscriber

DESCRIPTION

This function adds a new subscriber device's internal subscriber's list, generates a unique subscription Id for this subscriber, sets a duration in seconds for this subscription to be valid and sends a subscription response indicating success or failure to subscription request.

RETURNS

0	Operation was a success
-1	Operation failed

SEE ALSO

UPnP_AcceptSubscriptionAsync ( )

2.16 UPnP_AcceptSubscriptionAsync

FUNCTION

Accept a new subscription request in Asynchronous (non blocking) mode.

SUMMARY

int UPnP_AcceptSubscriptionAsync (UPnPSubscriptionRequest* subReq,
const GENA_CHAR* subscriptionId, UPNP_INT32 timeoutSec,
IXML_Document* propertySet, UPNP_INT32 firstNotifyDelayMsec)

UPnPSubscriptionRequest* subReq	Address of structure containing subscription request information
GENA_CHAR* subscriptionId	Alternate subscription identifier for the subscriber (Optional)
UPNP_INT32 timeoutSec	Duration in seconds for which the subscription is valid (Optional)
IXML_Document* propertySet	Address of response message in XML format
UPNP_INT32 firstNotifyDelayMsec	Delay in milliseconds before sending the first event notification to the new subscriber

DESCRIPTION

This function asynchronously adds a new subscriber device's internal subscriber's list. Optional parameters may be given a value of zero to indicate use default.

RETURNS

0	Operation was a success
-1	Operation failed

SEE ALSO

UPnP_AcceptSubscription ( )

2.17 UPnP_GetRequestedDeviceName

FUNCTION

Extracts Unique Device Name (UDN) from an action/subscription request.

SUMMARY

const UPNP_CHAR* UPnP_GetRequestedDeviceName (void* eventStruct,
enum e_UPnPDeviceEventType eventType)

void* eventStruct	Pointer to a UPnPActionRequest or a UPnPSubscriptionRequest structure depending on the type of request.
enum e_UPnPDeviceEventType eventType	Indicated the type of control point's request to handle. If eventType == UPNP_DEVICE_EVENT_ACTION_REQUEST, it indicates an action request If eventType == UPNP_DEVICE_EVENT_SUBSCRIPTION_REQUEST, it indicated a subscription request type.

DESCRIPTION

This function is used by application's event handler (device callback) to handle an action or a subscription request invoked by a control point on this device. This function extracts unique device name (UDN) for the device targetted by control point's action or subscription request.

RETURNS

const UPNP_CHAR*	Pointer to string containing unique device name
NULL	Operation failed

SEE ALSO

UPnP_GetRequestedServiceId ( ), UPnP_GetRequestedActionName ( ), UPnP_GetArgValue ( )

2.18 UPnP_GetRequestedServiceId

FUNCTION

Extracts service identifier from an action/subscription request.

SUMMARY

const UPNP_CHAR* UPnP_GetRequestedServiceId (void* eventStruct,
enum e_UPnPDeviceEventType eventType)

void* eventStruct	Pointer to a UPnPActionRequest or a UPnPSubscriptionRequest structure depending on the type of request.
enum e_UPnPDeviceEventType eventType	Indicated the type of control point's request to handle. If eventType == UPNP_DEVICE_EVENT_ACTION_REQUEST, it indicates an action request If eventType == UPNP_DEVICE_EVENT_SUBSCRIPTION_REQUEST, it indicated a subscription request type.

DESCRIPTION

This function is used by application's event handler (device callback) to handle an action or a subscription request invoked by a control point on this device. This function extracts service identifier for a service targetted by control point's action or subscription request.

RETURNS

const UPNP_CHAR*	Pointer to string containing service identifier
NULL	Operation failed

SEE ALSO

UPnP_GetRequestedDeviceName ( ), UPnP_GetRequestedActionName ( ), UPnP_GetArgValue ( )

2.19 UPnP_GetRequestedActionName

FUNCTION

Extracts name of targetted action from an action request.

SUMMARY

const UPNP_CHAR* UPnP_GetRequestedActionName (void* eventStruct,
enum e_UPnPDeviceEventType eventType)

void* eventStruct	Pointer to a UPnPActionRequest or a UPnPSubscriptionRequest structure depending on the type of request.
enum e_UPnPDeviceEventType eventType	Indicated the type of control point's request to handle. If eventType == UPNP_DEVICE_EVENT_ACTION_REQUEST, it indicates an action request If eventType == UPNP_DEVICE_EVENT_SUBSCRIPTION_REQUEST, it indicated a subscription request type.

DESCRIPTION

This function is used by application's event handler (device callback) to handle an action or a subscription request invoked by a control point on this device. This function extracts the supplied action name from control point's action request.

RETURNS

const UPNP_CHAR*	Pointer to string containing action name
NULL	Operation failed

SEE ALSO

UPnP_GetRequestedDeviceName ( ), UPnP_GetRequestedServiceId, UPnP_GetArgValue ( )

2.20 UPnP_SetActionErrorResponse

FUNCTION

Sets error code and error description as response to an action request.

SUMMARY

void UPnP_SetActionErrorResponse (UPnPActionRequest* request, UPNP_CHAR* description,
UPNP_INT32 value)

UPnPActionRequest* request	Pointer to structure containing action request.
UPNP_CHAR* description	Pointer to string providing error description.
UPNP_INT32 value	Error code value.

DESCRIPTION

This function is used by application's event handler (device callback). This function sets error code and error description as response to an action request.

RETURNS

None

2.21 UPnP_GetArgValue

FUNCTION

Extracts the value of an argument from an action request.

SUMMARY

const UPNP_CHAR* UPnP_GetArgValue (UPnPActionRequest* request, const UPNP_CHAR* argName)

UPnPActionRequest* request	Pointer to structure containing action request.
UPNP_CHAR* argName	Name of action's argument whose value is to be extracted.

DESCRIPTION

Extracts the value of an argument from an action request. Action information is stored in form of IXML element.

RETURNS

const UPNP_CHAR*	Pointer to string containing argument's value
NULL	Operation failed

SEE ALSO

UPnP_GetRequestedActionName ( ), UPnP_SetActionResponseArg ( ), UPnP_CreateActionResponse ( )

2.22 UPnP_CreateActionResponse

FUNCTION

Creates a message wrapper for SOAP action response.

SUMMARY

int UPnP_CreateActionResponse (UPnPActionRequest* request)

UPnPActionRequest* request	Pointer to structure containing action request.

DESCRIPTION

Creates a response message skeleton for the supplied SOAP action request.

RETURNS

0	Operation was a success
-1	Operation failed

SEE ALSO

UPnP_GetRequestedActionName ( ), UPnP_GetArgValue ( ), UPnP_SetActionResponseArg ( )

2.23 UPnP_SetActionResponseArg

FUNCTION

Inserts name and value of an argument to an action response message.

SUMMARY

int UPnP_SetActionResponseArg (UPnPActionRequest* request, const UPNP_CHAR* name,
const UPNP_CHAR* value)

UPnPActionRequest* request	Pointer to structure containing action request.
UPNP_CHAR* name	Name of action's argument whose value is to be added.
UPNP_CHAR* value	Pointer to string containing argument value.

DESCRIPTION

Adds an argument name and its value to response message created for an action request.

RETURNS

0	Operation was a success
-1	Operation failed

SEE ALSO

UPnP_GetRequestedActionName ( ), UPnP_GetArgValue ( ), UPnP_CreateActionResponse ( )

2.24 UPnP_CreateAction

FUNCTION

Create a SOAP action request.

SUMMARY

IXML_Document* UPnP_CreateAction (const UPNP_CHAR* serviceTypeURI,
const UPNP_CHAR* actionName)

UPNP_CHAR* serviceTypeURI	String containing service type of the target service.
const UPNP_CHAR* actionName	Name on action on the target service.

DESCRIPTION

Creates an XML document which will hold the SOAP action request message. This function returns the address of newly formed XML document. After finishing the process of sending action request the application must release this xml document.

RETURNS

IXML_Document*	pointer to newly created IXML_Document, which can be passed into UPnP_SetActionArg to set the action arguments
NULL	Operation failed

SEE ALSO

UPnP_ SetActionResponseArg ( )

2.25 UPnP_SetActionArg

FUNCTION

Set an argument for a SOAP action response/request..

SUMMARY

int UPnP_SetActionArg (IXML_Document* actionDoc, const UPNP_CHAR* name,
const UPNP_CHAR* value)

IXML_Document* actionDoc	Pointer to action respose message.
UPNP_CHAR* name	Pointer to string containing argument name.
UPNP_CHAR* value	Pointer to string containing argument value.

DESCRIPTION

This function can be used on an IXML_Document created by either UPnP_CreateActionResponse or UPnP_CreateAction to set either the input or output arguments for a SOAP action.

RETURNS

0	Operation was a success
-1	Operation failed

SEE ALSO

UPnP_CreateAction ( )

2.26 UPnP_AddToPropertySet

FUNCTION

Add name and value pair to a message property set.

SUMMARY

int UPnP_AddToPropertySet (IXML_Document** doc, const UPNP_CHAR* name,
const UPNP_CHAR* value)

IXML_Document** doc	Address of property set.
UPNP_CHAR* name	Pointer to name for new entry.
UPNP_CHAR* value	Address of value of for the new entry

DESCRIPTION

Add a new name value pair entry to the property set. A in SOAP property set is an xml document which hold the body of a response / request message.

RETURNS

UPNP_CHAR*	Pointer to string containing value
NULL	Property was not found

EXAMPLE

```
IXML_Document *propertySet = 0;

UPnP_AddToPropertySet (&propertySet, “Status”, value);

ixmlDocument_free(propertySet);
```

SEE ALSO

UPnP_ SetActionResponseArg ( )

2.27 UPnP_GetPropertyValueByName

FUNCTION

Get the value of a named property in a message property set.

SUMMARY

const UPNP_CHAR* UPnP_GetPropertyValueByName (IXML_Document* propertySet,
const UPNP_CHAR* name)

IXML_Document* propertySet	Address of xml property set.
UPNP_CHAR* name	Name of the property element.

DESCRIPTION

The string returned must not be modified in any way. The string containing value is only valid until the IXML_Document is deleted.

RETURNS

UPNP_CHAR*	Pointer to string containing value
NULL	Property was not found

SEE ALSO

UPnP_ GetPropertyValueByIndex ( ), UPnP_ GetPropertyNameByIndex ( )

2.28 UPnP_GetPropertyNameByIndex

FUNCTION

Get the name of the nth property.

SUMMARY

const UPNP_CHAR* UPnP_GetPropertyNameByIndex (IXML_Document* propertySet, int index)

IXML_Document* propertySet	Address of xml property set.
int index	Name of the property element.

DESCRIPTION

The string returned must not be modified in any way. The string containing value is only valid until the IXML_Document is deleted.

RETURNS

UPNP_CHAR*	Pointer to string containing name
NULL	Property was not found

SEE ALSO

UPnP_ GetPropertyValueByIndex ( ), UPnP_ GetPropertyValueByName ( )

2.29 UPnP_GetPropertyValueByIndex

FUNCTION

Get the value of the nth property.

SUMMARY

const UPNP_CHAR* UPnP_GetPropertyValueByIndex (IXML_Document* propertySet, int index)

IXML_Document* propertySet	Address of xml property set.
int index	Index in property for value.

DESCRIPTION

The string returned must not be modified in any way. The string containing value is only valid until the IXML_Document is deleted.

RETURNS

UPNP_CHAR*	Pointer to string containing value
NULL	Property was not found

SEE ALSO

UPnP_ GetPropertyNameByIndex ( ), UPnP_ GetPropertyValueByName ( )

Appendix I

UPnP Device Initialization Example

Setting up a UPnP Device

This example code demonstrates in brief the necessary steps to set up a UPnP device for discovery, description, control, and eventing.

```
int main (void)
{
	int result;
	IXML_Document *xmlDevice;
	UPnPRuntime rt;
	UPnPRootDeviceHandle rootDevice;

	// UPnP maintains a runtime structure; The first step is to
	// initialize UPnPRuntime struct. UPnP_RuntimeInit takes a
	// pointer to an uninitialized UPnPRuntime struct and other
	// necessary necessary data to initialize and populate upnp
	// the engine.

	result = UPnP_RuntimeInit (
			&rt,
			0,                          // serverAddr: IP_ANY_ADDR
			0,                          // serverPort: any port
			RTP_NET_TYPE_IPV4,  // ipv4
			"c:\\www-root\\",           // web server root dir
			10,                         // maxConnections
			5                           // maxHelperThreads
		);

	if (result < 0)
	{
		return (-1);
	}

	// Next, we need a UPnPDeviceRuntime; UPnP_DeviceInit takes
	//  a pointer to an uninitialized UPnPDeviceRuntime struct
	//  and does all necessary initialization.

	result = UPnP_DeviceInit (
				&deviceRuntime,
				&rt
			);

	if (result < 0)
	{
		return (-1);
	}

	// Load the root device description page into memory.
	xmlDevice = ixmlLoadDocument("c:\\www-root\\device.xml");
	if (!xmlDevice)
	{
		return (-1);
	}
```

```
        result = UPnP_RegisterRootDevice (
                        &deviceRuntime,
                        "device.xml",
                        xmlDevice,
                        1,              // auto address resolution
                        testDeviceCallback,
                        0,              // userData for callback
                        &rootDevice,
                        1               // advertise
                );

        if (result < 0)
        {
                return (-1);
        }

        UPnP_DeviceAdvertise(rootDevice, ANNOUNCE_FREQUENCY_SEC,
                                                REMOTE_CACHE_TIMEOUT_SEC);

        // start the UPnP daemon thread
        UPnP_StartDaemon(&rt);

        // for polled mode, use this

        //while (1)
        //{
        //UPnP_ProcessState (&rt,1000);
                //printf(".");
        //}

}
```

Appendix II

Sample Device Callback

Here is an example of an application callback implementing network light. The following section will demostrate code to handle events generated as a result of a control (action) or event (subscription) request received by upnp device.

```
#define ANNOUNCE_FREQUENCY_SEC    10
#define REMOTE_CACHE_TIMEOUT_SEC  1800


UPnPDeviceRuntime deviceRuntime;

int testDeviceCallback (
                        void *userData,
                        struct s_UPnPDeviceRuntime *deviceRuntime,
                        UPnPRootDeviceHandle rootDevice,
                        enum e_UPnPDeviceEventType eventType,
                        void *eventStruct);

RTP_MUTEX lightMutex;
int lightStatus = 0;
int levelStatus = 50; // default light level
#define MINLEVEL  0
#define MAXLEVEL  100

int main (void)
{
   // UPnP device Application
   return (0);
}

int testDeviceCallback (
                        void *userData,
                        struct s_UPnPDeviceRuntime *deviceRuntime,
                        UPnPRootDeviceHandle rootDevice,
                        enum e_UPnPDeviceEventType eventType,
                        void *eventStruct)
{
          const UPNP_CHAR* targetDeviceName;
          const UPNP_CHAR* targetServiceId;

          //extracty the UDN for the request
          targetDeviceName = UPnP_GetRequestedDeviceName (eventStruct, eventType);

          //extract the serviceId for a particular service on the device
          targetServiceId = UPnP_GetRequestedServiceId (eventStruct, eventType);

          //if the callback is invoked to handle action request or a subscription request
          switch (eventType)
          {
                    case UPNP_DEVICE_EVENT_ACTION_REQUEST:
                    {
                              const UPNP_CHAR* targetActionName;
                              // To handle action requests, the cookie (eventStruct) is to be cast to be
                              //  of UPnPActionRequest type
                              UPnPActionRequest *request = (UPnPActionRequest *) eventStruct;

                              // request structure holds the all the action request information
                              // start by determining if the action request is meant for this device
                              //  by compairing the unique device name supplied in the request with the
                              //  UDN of this device
```

```
if (!rtp_strcmp(targetDeviceName, "uuid:9de82eea-b4a2-41ae-b182-058befd73af8"))
{
	// if the action request is intended for this device, check to see which service on
	 // this device the action will be performed upon

	// extract the actionName for a particular service on the device
	targetActionName = UPnP_GetRequestedActionName (eventStruct, eventType);

	// SERVICE : Switch Power
	if (!rtp_strcmp(targetServiceId, "urn:upnp-org:serviceId:SwitchPower.0001"))
	{
		// a service may offer multiple actions, next step is to determine the
		// target action
		if (!rtp_strcmp(targetActionName, "GetStatus"))
		{
			// create a response to acknowledge the request
			if(UPnP_CreateActionResponse(request) >=0 )
			{
				UPNP_CHAR temp[5];

				rtp_sig_mutex_claim(lightMutex);
				// get the value of lightStatus into temp variable
				rtp_itoa(lightStatus, temp, 10);
				rtp_sig_mutex_release(lightMutex);

				// Since this action is a out action, this action is
				// invoked by the control point to query the value of
				// state variable. For such actions with direction of
				// variables as out, the response needs
				// to contain the name and current value of action's
				// out variable.
				if(UPnP_SetActionResponseArg(request,
"ResultStatus", temp) < 0)
				{
					return(-1);
				}
			}
			else
			{
				return (-1); // create action response failed
			}

		}
		else if (!rtp_strcmp(targetActionName, "SetTarget"))
		{
			// If the action contains argument having direction 'in', this
			// means that control point
			// will send an action request with a new value of argument
			// that will replace the current
			// value of the argument
			// The following step extracts the value of the argument if
			 // supplied
			const UPNP_CHAR *newTargetValue =
UPnP_GetArgValue(request, "newTargetValue");
			if (newTargetValue)
			{
				int i = rtp_atoi(newTargetValue);
				int changed = 0;

				if (i)
				{
					printf("Light turned on.\n");
				}
				else
				{
					printf("Light turned off.\n");
				}

				rtp_sig_mutex_claim(lightMutex);
```

```
                                        if (i != lightStatus)
                                        {
                                                changed = 1;
                                        }

                                        lightStatus = i;

                                        rtp_sig_mutex_release(lightMutex);
                                        // if this actions causes the value of a state variable to
                                        // change a notification to all the subscribed devices
                                        // will be send
                                        if (changed)
                                        {
                                                // must be initialized to zero
                                                 IXML_Document *propertySet = 0;
                                                UPNP_CHAR temp[15];

                                                rtp_itoa(i, temp, 10);

                                                // add name and vaule of the changed
                                                // variable to the property set
                                                // this property set is sent to the subscribers
                                                // as the event notification

                                                UPnP_AddToPropertySet(&propertySet,
"Status", temp);


                                                // send all the subscribers a notification of
                                                // change of value event
                                                UPnP_DeviceNotifyAsync(
                                                        deviceRuntime,
rootDevice,
                                                        targetDeviceName,
                                                        targetServiceId,
                                                        propertySet);

                                                ixmlDocument_free(propertySet);
                                        }
                                }

                                // create a response to acknowledge the request
                                if(UPnP_CreateActionResponse(request) < 0 )
                                {
                                        return(-1);
                                }
                        }
                        // unknown action name
                        else
                        {
                           UPnP_SetActionErrorResponse(request, "Invalid Action", 401);
                        }
                }
                else
                // SERVICE : Dimming Service
                if (!rtp_strcmp(targetServiceId, "urn:upnp-org:serviceId:DimmingService.0001"))
                {
                        if (!rtp_strcmp(targetActionName, "GetLoadLevelStatus"))
                        {
                                // create a response to acknowledge the request
                                if(UPnP_CreateActionResponse(request) >=0 )
                                {
                                        UPNP_CHAR temp[15];

                                        rtp_sig_mutex_claim(lightMutex);
                                        // get the value of lightStatus into temp variable
                                        rtp_itoa(levelStatus, temp, 10);
                                        rtp_sig_mutex_release(lightMutex);
```

```
                                                // Since this action is a out action, this action is
                                                // invoked by the control point to query the value of
                                                // state variable
                                                // For such actions with direction of variables as out,
                                                // the response needs to contain the name and
                                                // current value of action's out variable.
                                                UPnP_SetActionResponseArg(request,
"RetLoadLevelStatus", temp);
                                        }
                                }

                                else if (!rtp_strcmp(targetActionName, "GetMinLevel"))
                                {
                                        // create a response to acknowledge the request
                                        if(UPnP_CreateActionResponse(request) >=0 )
                                        {
                                                // Since this action is a out action, this action is
                                                // invoked by the control point to query the value of
                                                // state variable.
                                                // For such actions with direction of variables as out,
                                                // the response needs to contain the name and
                                                // current value of action's out variable.
                                                UPnP_SetActionResponseArg(request, "MinLevel",
"0");
                                        }

                                }

                                else if (!rtp_strcmp(targetActionName, "SetLoadLevelTarget"))
                                {

                                        // If the action contains argument having direction 'in', this
                                        // means that control point will send an action request with a
                                        // new value of argument that will replace the current
                                        // value of the argument
                                        // The following step extracts the value of the argument if
                                        // supplied
                                        const UPNP_CHAR *newTargetValue =
UPnP_GetArgValue(request, "NewLoadLevelTarget");
                                        if (newTargetValue)
                                        {
                                                int i = rtp_atoi(newTargetValue);
                                                int changed = 0;

                                                if(i < MINLEVEL || i > MAXLEVEL)
                                                {
                                                   printf("Error: New lightLevel value out of range\n");
                                                   UPnP_SetActionErrorResponse(request, "Invalid
Action", 402);
                                                   break;
                                                }

                                                printf("New Light Level set to :%d\n", i);

                                                rtp_sig_mutex_claim(lightMutex);

                                                if (i != levelStatus)
                                                {
                                                        changed = 1;
                                                }

                                                levelStatus = i;

                                                rtp_sig_mutex_release(lightMutex);

                                                // if this actions causes the value of a state variable to
                                                // change a notification to all the subscribed devices
                                                // will be send
                                                if (changed)
```

```
                                                {
                                                        // must be initialized to zero
                                                        IXML_Document *propertySet = 0;
                                                        UPNP_CHAR temp[5];

                                                        rtp_itoa(i, temp, 10);

                                                        // add name and vaule of the changed
                                                        // variable to the property set
                                                        // this property set is sent to the subscribers
                                                        // as the event notification
                                                        UPnP_AddToPropertySet(&propertySet,
"LoadLevelStatus", temp);

                                                        // send all the subscribers a notification of
                                                        // change of value event
                                                        UPnP_DeviceNotifyAsync(
                                                                deviceRuntime,
rootDevice,
                                                                targetDeviceName,
                                                                targetServiceId,
                                                                propertySet);

                                                        ixmlDocument_free(propertySet);
                                                }
                                        }

                                        // create a response to acknowledge the request
                                        if(UPnP_CreateActionResponse(request) < 0 )
                                        {
                                                return(-1);
                                        }
                                }
                                // unknown action name
                                else
                                {
                                   UPnP_SetActionErrorResponse(request, "Invalid Action", 401);
                                }
                        }
                }

                break;
        }

        case UPNP_DEVICE_EVENT_SUBSCRIPTION_REQUEST:
        {
                UPnPSubscriptionRequest *request = (UPnPSubscriptionRequest *) eventStruct;

                if (!rtp_strcmp(targetDeviceName, "uuid:9de82eea-b4a2-41ae-b182-058befd73af8"))
                {
                        // SERVICE : Switch Power
                        if (!rtp_strcmp(targetServiceId, "urn:upnp-org:serviceId:SwitchPower.0001"))
                        {
                                IXML_Document *propertySet = 0; // must be initialized to zero
                                UPNP_CHAR temp[5];

                                rtp_sig_mutex_claim(lightMutex);
                                rtp_itoa(lightStatus, temp, 10);
                                rtp_sig_mutex_release(lightMutex);

                                UPnP_AddToPropertySet(&propertySet, "Status", temp);
                                UPnP_AcceptSubscription(request, 0, 0, propertySet, 100);
                                ixmlDocument_free(propertySet);
                        }
                        else
                        // SERVICE : Dimming Service
                        if (!rtp_strcmp(targetServiceId, "urn:upnp-org:serviceId:DimmingService.0001"))
                        {
                                IXML_Document *propertySet = 0; // must be initialized to zero
```

```
                                        UPNP_CHAR temp[5];

                                        rtp_sig_mutex_claim(lightMutex);
                                        rtp_itoa(levelStatus, temp, 10);
                                        rtp_sig_mutex_release(lightMutex);

                                        UPnP_AddToPropertySet(&propertySet, "LoadLevelStatus", temp);
                                        UPnP_AcceptSubscription(request, 0, 0, propertySet, 100);
                                        ixmlDocument_free(propertySet);
                                }
                        }

                        break;
                }
        }

        return (0);
}
```