

---

---

# EBS UPnP Control Point Software Development Kit (SDK)

## User Manual

*Revised Nov 2006*

---

---

Copyright © 2006 EBS Inc.





## Table Of Contents

<b>PART I - EBS UPnP Control Point SDK User Guide</b>	<b>1</b>
<b>Section 1: Introduction</b>	<b>1</b>
<b>Section 2: UPnP Phases</b>	<b>3</b>
<b>Section 3: Server and Client Interaction Model</b>	<b>5</b>
<b>Section 4: Software Development Kit (SDK) Architecture</b>	<b>7</b>
4.1 RTPLATFORM	7
4.2 Application	8
4.3 IXML Library	8
4.4 UPnP.DOM	8
4.5 HTTP Library (HTTP Server and HTTP Client)	8
4.6 UPnP.c and UPnPcli.c	8
4.7 ControlAction.c and SoapCli.c	8
4.8 ControlEvent.c and GenaCli.c	8
4.9 ControlDescribe.c	8
4.10 ControlDiscover.c, SsdpSrv.c and SsdpCli.c	8
<b>Section 5: Getting Started</b>	<b>9</b>
5.1 Writing your application	9
5.1.1 Initializing and Setting Up UPnP Runtime	10
5.1.2 Initializing UPnP Control Point	10
5.1.3 Active Searches by the Control Point	11
5.1.4 Getting Device and Service Information ( Description Phase )	12
5.1.5 Controlling a Remote Device (Control Phase)	16
5.1.6 Subscribing and Unsubscribing to a Service on a Device (Event Phase)	18
5.1.7 Listening for Passive Events and Asynchronously Generated Active Events	19
5.1.8 Shutting Down	19
<b>Section 6. Writing an Application Callback</b>	<b>21</b>
<b>PART II – Porting and Configuration Guide</b>	<b>29</b>
UPnP SDK Source Code Structure	31
Configuring UPnP SDK	35
Porting UPnP SDK	35
<b>PART III - EBS UPnP Control Point API Reference Manual</b>	<b>37</b>
<b>Section 1: Introduction</b>	<b>39</b>
<b>Section 2: EBS UPnP Control Point API</b>	<b>41</b>
2.1 UPnP_RuntimeInit	43
2.2 UPnP_RuntimeDestroy	44
2.3 UPnP_AddVirtualFile	45
2.4 UPnP_RemoveVirtualFile	46
2.5 UPnP_ProcessState	47
2.6 UPnP_StartDaemon	48

2.7 UPnP_StopDaemon	49
2.8 UPnP_ControlPointInit	50
2.9 UPnP_ControlPointDestroy	51
2.10 UPnP_ControlFindAll	52
2.11 UPnP_ControlFindAllDevices	53
2.12 UPnP_ControlFindDevicesByType	54
2.13 UPnP_ControlFindDevicesByUUID	55
2.14 UPnP_ControlFindDevicesByService	56
2.15 UPnP_AcceptSubscription	57
2.16 UPnP_AcceptSubscriptionAsync	58
2.17 UPnP_GetRequestedDeviceName	59
2.18 UPnP_GetRequestedServiceId	60
2.19 UPnP_GetRequestedActionName	61
2.20 UPnP_SetActionErrorResponse	62
2.21 UPnP_GetArgValue	63
2.22 UPnP_CreateActionResponse	64
2.23 UPnP_SetActionResponseArg	65
2.24 UPnP_CreateAction	66
2.25 UPnP_SetActionArg	67
2.26 UPnP_AddToPropertySet	68
2.27 UPnP_GetPropertyValueByName	69
2.28 UPnP_GetPropertyNameByIndex	70
2.29 UPnP_GetPropertyValueByIndex	71
<b>Appendix I</b>	<b>73</b>
UPnP Control Point Example	73
<b>Appendix II</b>	<b>83</b>
Sample Control Point Callback	83

## PART I - EBS UPnP Control Point SDK 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 contains 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 messages 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.

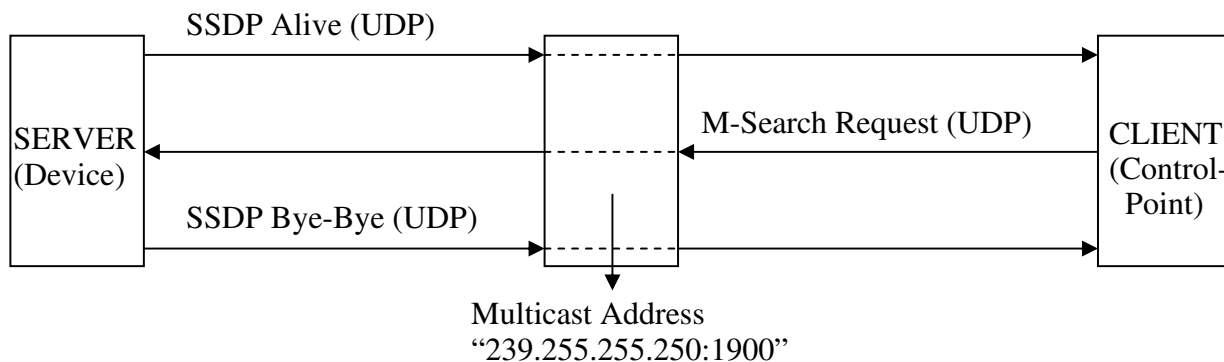


Figure 1: UPnP client server multicast communication

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)

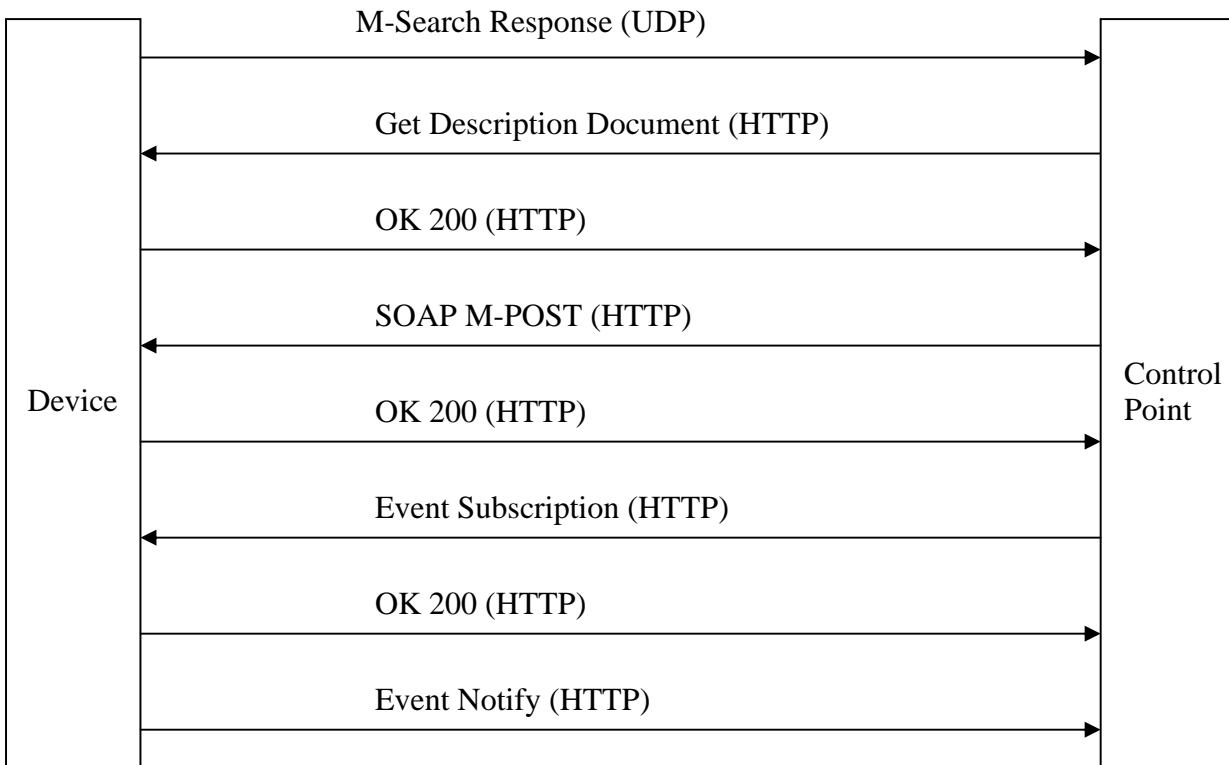


Figure 2: UPnP client server unicast communication

- 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

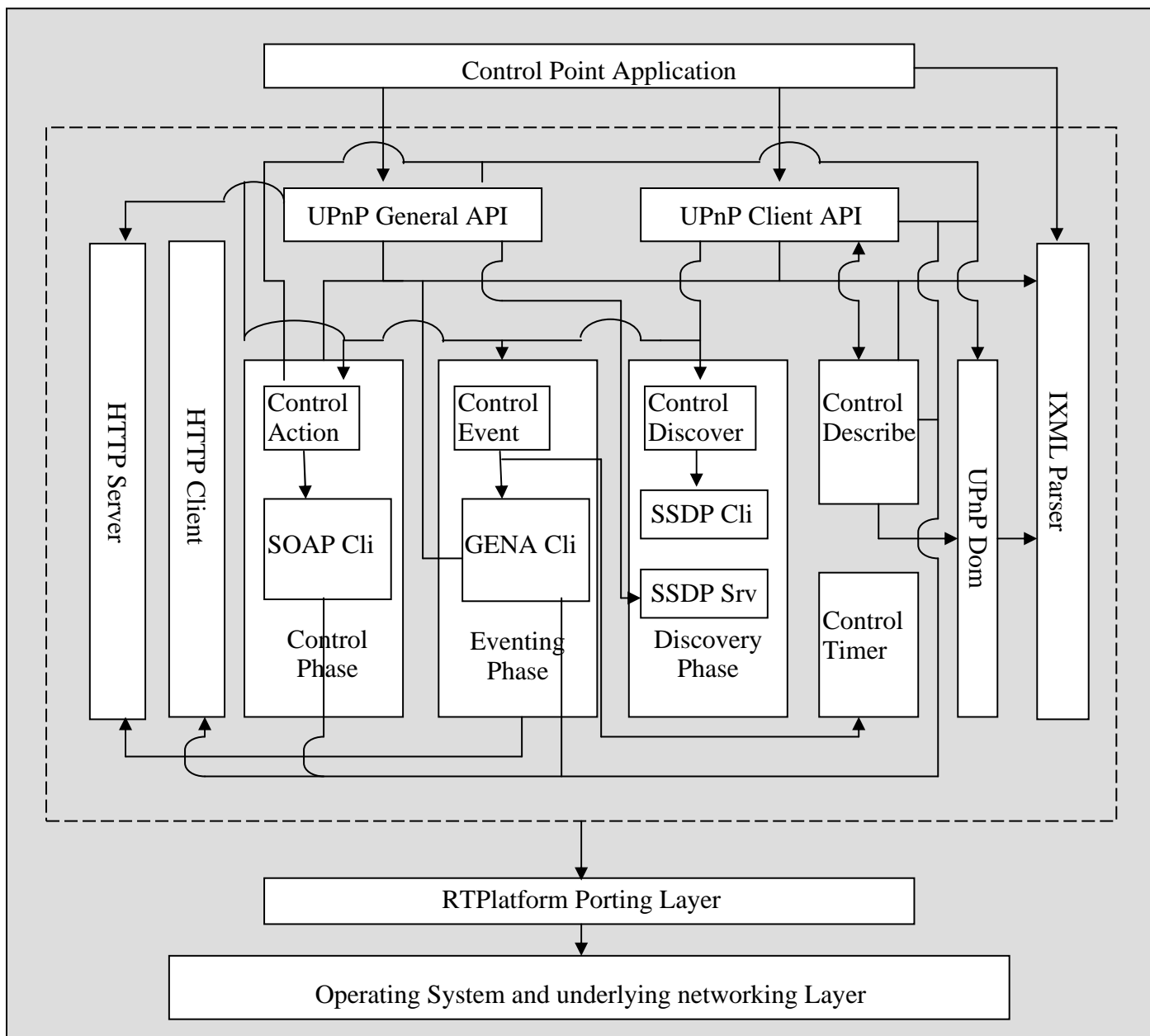


Figure 3: EBS UPnP Control Point Architecture

Figure 3, shows where the different modules of control point 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 Client)

This library contains the web server and http client used by UPnP control point SDK. 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 UPnPCLI.c

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

## 4.7 ControlAction.c and SoapCli.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 ControlEvent.c and GenaCli.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 ControlDescribe.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 ControlDiscover.c, SsdpSrv.c and SsdpCli.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 Writing your application

The UPnP stack needs to access to its state at all times during its lifetime. UPnPRuntime structure shown below is designed to hold the runtime information of the stack. Although this structure is used internally it is owned by the application and the application developer will need to initialize this structure using supplied API. Figure 4 shown below shows a sample of control point's application body

```
int controlPointCallback (
    UPnPControlPoint* cp,
    UPnPControlEvent* event,
    void *perControl,
    void *perRequest
);

int main (int* argc, char* agrv[])
{
    int result;
    UpnPRuntime* rt;
    UpnPControlPoint* cp;

    rt = (UpnPRuntime*) malloc (sizeof (UpnPRuntime)); ++
    cp = (UpnPControlPoint *) malloc (sizeof (UpnPControlPoint)); ++

    rtp_net_init();
    rtp_threads_init();

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

    if (result >= 0)
    {
        /* Initialize the control point */
        if (UPnP_ControlPointInit(cp, rt, controlPointCallback, 0) >= 0)
        {
            .....

            UPnP_ControlPointDestroy (cp, 0);
        }
        UPnP_RuntimeDestroy(rt);
    }
    rtp_threads_shutdown();
    rtp_net_exit();
    return(0);
}
```

```

int controlPointCallback (
    UPnPControlPoint* cp,
    UPnPControlEvent* event,
    void *perControl,
    void *perRequest )
{
    .....Callback body.....
}
++ : malloc used in this sample to demonstrate that the two structures are owned by the application.

```

Figure 4: Sample framework of a control point application

### 5.1.1 Initializing and Setting Up UPnP Runtime

UPnP control point application must start by initializing and setting up UPnPRuntime structure which holds runtime information for the stack, using UPnP\_RuntimeInit ( ) API, for example,

```

result = UPnP_RuntimeInit ( &upnpRuntime, serverAddr, serverPort, "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) IP address. 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.

### 5.1.2 Initializing UPnP Control Point

The application must next initialize and setup control point structure. The control point needs to know the address of the stack's runtime structure, which was initialized in above step, the address of application callback which as explained in later section is used to handle events. Use API UPnP\_ControlPointInit ( ) in order to initialize and setup the control point, for example:

```

result = UPnP_ControlPointInit (&controlPoint, &upnpRuntime, appCallback, userData);

```

As first parameter, supply address of an uninitialized buffer of type UPnPControlPoint, this will hold the information needed by the control point throughout its life. Next parameter is address of UpnPRuntime structure which was initialized in the previous step, third parameter is the address of the callback function which is invoked internally by the stack in case any control point event is generated. Section 6 of the manual explains callback in detail. The last parameter to this API is address of userdata that would be passed into the callback.

### 5.1.3 Active Searches by the Control Point

The control point application can also perform active searches on the network. EBS UPnP stack gives developer the option of conducting following searches on the network

- a. Search all devices and services on the network
- b. Search all root devices on the network
- c. Search all devices of a given device type (a device type is defined by UPnP forum)
- d. Search for a particular device by the device's unique ID (UUID, which is supplied by the device vendor)
- e. Search all services of a given service type

All the above searches are conducted for a given time period. If the searched target is detected during this period the control point stack internally invokes the application callback, with UPNP\_CONTROL\_EVENT\_DEVICE\_FOUND event, to handle the discovery. When the duration expires the application callback is invoked with a state UPNP\_CONTROL\_EVENT\_SEARCH\_COMPLETE to indicate and handle completion of the search.

#### 5.1.3a Search for All Devices and Services Available on the Network

In order for the control point to search for all available devices and services, synchronously or asynchronously, the application developer can use UPnP\_ControlFindAll ( ) API in the following fashion,

```
UPnP_ControlFindAll (&controlPoint, timeoutMsec, userData, waitForCompletion);
```

where, first argument is the address to structure holding control point information, the second argument specifies the duration of time in milliseconds for which this search will be performed, the third parameter is userData (cookie) which will be passed to the application callback and finally the last argument is a boolean select value which decides whether the search will be conducted synchronously(no waiting) or asynchronously (wait for completion).

#### 5.1.3b Search for All Root Devices on the Network

The control point application can search for all the root devices on the network using

UPnP\_ControlFindAllDevices API, here is an example

```
UPnP_ControlFindAllDevices (&controlPoint, timeoutMsec, userData, waitForCompletion)
```

where, arguments have same meaning as in above section 5.1.5.a.

#### 5.1.3c Search for All Devices given Device Type on the Network:

A device type for a device is one of the standard types defined by the working committee of UPnP forum. This type is also listed in the device description document under element 'deviceType'. A search can be initiated to locate all devices of a given device type using API UPnP\_ControlFindDevicesByType ( ) for example,

```
UPnP_ControlFindDevicesByType (&controlPoint, "InternetGatewayDevice:1", timeoutMsec, userData, 1)
```

where, the first argument to this API is address to structure holding control point information, the second argument is the string containing the target device type, timeoutMsec is the duration of time in milliseconds for which this search will be performed, 4th argument is the userData which will be passed to the application callback and finally the last argument is a boolean select value which decides whether the search will be conducted synchronously(no waiting) or asynchronously (wait for completion).

#### 5.1.3d Search for a Particular Device by its Unique ID:

Each device vendor specifies a unique device identifier to their device, this unique identifier does not change even across device reboots and is present in the 'UDN' field of the device's description document. For sending a search for a devices containing the specified unique id ( uuid ) use UPnP\_ControlFindDevicesByUUID ( ) API, a usage sample is provided below,

```
UPnP_ControlFindDevicesByUUID (&controlPoint, "upnp-WANDevice-1_0-0090a2777777", 5000, userData, 0)
```

where, the first argument is address to structure holding control point information, the second argument is the uuid which is the target of this search, timeoutMsec is the duration of time in milliseconds for which this search will be performed, 4th argument is the userData which will be passed to the application callback and finally the last argument is a boolean select value which decides whether the search will be conducted synchronously(no waiting) or asynchronously (wait for completion).

### 5.1.3e Search for All Services of a given Service Type:

A service type for a service is one of the standard types defined by the working committee of UPnP forum. This type is also listed in the device description document for each associated service under element 'serviceType'. A search can be initiated to locate all services of a given type using API `UPnP_ControlFindDevicesByService ( )` for example,

```
UPnP_ControlFindDevicesByService (&controlPoint, "SwitchPower:1", timeoutMsec, userData, waitForCompletion)
```

where, the first argument to this API is address to structure holding control point information, the second argument contains the target type, `timeoutMsec` is the duration of time in milliseconds for which this search will be performed, 4th argument is the `userData` which will be passed to the application callback and finally the last argument is a boolean select value which decides whether the search will be conducted synchronously (no waiting) or asynchronously (wait for completion).

### 5.1.4 Getting Device and Service Information ( Description Phase )

At this point the application can request device and service description documents and start extracting useful information from them. The developer will need to use three new types for this purpose: `UPnPControlDeviceHandle`, `UPnPControlDeviceInfo` and `UPnPControlServiceIterator`.

Here is the description for these types,

- a. *UPnPControlDeviceHandle* – Unique handler to a device. It is a pointer to `UPnPControlDevice` structure which is opaque to the developer, filled internally by the stack for each device.
- b. *UPnPControlDeviceInfo* – This structure is meant to be used by the application developer in order to read device information contained in device description document. This read only structure is used to pass the output from various API calls, all values are valid until control point is closed using `UPnP_ControlCloseDevice` API. The structure is shown below,

```
typedef struct s_UPnPControlDeviceInfo
{
    UPNP_CHAR*      deviceType;
    UPNP_CHAR*      friendlyName;
    UPNP_CHAR*      manufacturer;
    UPNP_CHAR*      manufacturerURL;
    UPNP_CHAR*      modelDescription;
    UPNP_CHAR*      modelName;
    UPNP_CHAR*      modelNumber;
    UPNP_CHAR*      modelURL;
    UPNP_CHAR*      serialNumber;
    UPNP_CHAR*      UDN;
    UPNP_CHAR*      UPC;
    UPNP_CHAR*      presentationURL;
}
UPnPDeviceInfo;
```

All the fields in this structure are filled by corresponding information contained in device's description document.

- c. *UPnPControlServiceIterator* – This structure is used internally by the stack for the purpose of enumerating and sorting the different services contained in a device. This structure is opaque to the developer and its internal fields are not to be accessed for use in the application. This structure is by API's to access and iterated the services.

The application needs to declare variable to hold the above the three types. For example,

```
UPnPControlDeviceHandle dh;
UPnPControlServiceIterator iterator;
UPnPControlDeviceInfo deviceInfo;
```

See figure 5 for sample.

```

void deviceOpenTest (UPnPRuntime *rt, UPnPControlPoint *cp)
{
    UPnPControlDeviceHandle dh;

    printf("Opening device at http://192.168.0.6:50989/...");
    dh = UPnP_ControlOpenDevice(cp, "http://192.168.0.6:50989/");
    if (dh != 0)
    {
        UPnPControlDeviceInfo info;
        UPnPControlServiceIterator i;

        UPnP_ControlGetDeviceInfo(dh, &info);
        printf ( " deviceType:      %.60s\n"
                " manufacturer:      %.60s\n"
                " manufacturerURL:  %.60s\n"
                " modelName:        %.60s\n"
                " modelURL:         %.60s\n"
                " UDN:              %.60s\n"
                " UPC:              %.60s\n",
                info.deviceType,
                info.manufacturer,
                info.manufacturerURL,
                info.modelName,
                info.modelURL,
                info.UDN,
                info.UPC);

        if (UPnP_ControlGetServices(dh, &i) >= 0)
        {
            UPNP_CHAR* serviceId = UPnP_ControlNextService(&i);

            while (serviceId)
            {
                printf ("Found Service: %.60s\n", serviceId);
                printf (" (type: %.65s)\n", UPnP_ControlGetServiceType(dh, serviceId));
                serviceId = UPnP_ControlNextService(&i);
            }

            UPnP_ControlServiceIteratorDone(&i);
        }

        UPnP_ControlCloseDevice(dh);
    }
    else
    {
        printf (" open failed.\n");
    }
}

```

Figure 5: Demonstrates description phase

These will be used in all API's of this section.

#### 5.1.4a Accessing and Opening a Device Located at a Given URL Synchronously:

The first step that the developer needs to take is to access a device at a desired URL and request its information. The desired url is either url of a device that the developer already knows exists on the network or one which is obtained at realtime as a result of one of the searches in above sections ( 5.1.4, 5.1.5). Upon success the developer gets a unique device handle to the newly opened device.

API, UPnP\_ControlOpenDevice ( ) synchronously (blocking) requests the remote device's description document and makes the device available to be used by the control point, shown below is a sample usage:

```
dh = UPnP_ControlOpenDevice(&cp, "http://192.168.0.6:1575/light.xml");
```

This api returns a handle, of type *UPnPControlDeviceHandle*, to the opened device. It takes address of the control point structure and url of the remote device as its arguments.

#### 5.1.4b Asynchronously Accessing and Opening a Device Located at a Given URL:

If the developer needs to use asynchronous (non blocking) version of the above call, API *UPnP\_ControlOpenDeviceAsync* provides the developer this option. This API makes asynchronous (non blocking) requests for remote device's description document and makes the device available to be used by the control point, for example:

```
UPnP_ControlOpenDeviceAsync(&cp, deviceUrl, userData);
```

In case of success the SDK internally invokes the application callback with event *UPNP\_CONTROL\_EVENT\_DEVICE\_OPEN*. In case, the control point fails in opening the device *UPNP\_CONTROL\_EVENT\_DEVICE\_OPEN\_FAILED* event in the callback reflects the failure. The developer needs to catch the events in the callback and handle them accordingly. See more details in callback section of the manual. This API takes following arguments address of the control point structure, url of the remote device and *userData* to be passed to the callback as a result of this event.

#### 5.1.4c Extracting Device Information:

After successfully making the remote device available for use, using the above API, application can now extract the device information if needed. This information is the one that the device maintains in its description document and is made available through fields of *UPnPControlDeviceInfo* structure. The API *UPnP\_ControlGetDeviceInfo* ( ) takes in handle to the device, and address to a *UPnPControlDeviceInfo* structure which will hold device information after a successful call to this API. Here is an example, *UPnP\_ControlGetDeviceInfo*(dh, &info);

#### Services Available in a Device

Having access to the remote device and all its information, the developer is now in a position to determine all the services contained in the device. The developer can enumerate the services available on the device, obtain their service identifier and service types. The application can then request the service description document for any service of interest. Below are detailed instructions to perform such operation.

#### 5.1.4d Setup Device for Extracting Service Information:

Once the application has a handle to the remote device and the application requires to determine information on the services contained in the device it must setup the device to provide information about the services contained in it. API *UPnP\_ControlGetServices* ( ) sets up the device so that the application can extract service information in the next steps, here is the usage:

```
UPnP_ControlGetServices(dh, &iterator)
```

where, *dh* is handle to the device and second parameter is address of *UPnPControlServiceIterator* structure. This API sets up and initializes service iterator internally. This iterator is used by the stack to extract service information.

#### 5.1.4e Get Service Identifier (serviceld) of the Next Service on the Device

At this point the application can search for available services on the device. API *UPnP\_ControlNextService* ( ) obtains the service id of the next service in the device description document, a null is returned in case no next service is available. Here is an example,

```
UPNP_CHAR* serviceld = UPnP_ControlNextService(&i);
```

Where, address of the serviceIterator initialized in the previous step is supplied as parameter.

#### 5.1.4f Obtain the Service Type of a Service on a Device with known Service Id

If the service Id for the service is available the developer can obtain the service type for this service on a given device. The API *UPnP\_ControlGetServiceType* ( ) returns a pointer to a string containing the service type, here is a sample usage:

```
UPnP_ControlGetServiceType(dh, serviceld)
```

Here *dh* is the device handle and *serviceld* is the service identifier for the service on the device.

#### 5.1.4g Obtain the Service Identifier for a Service on a device with known Service Type

An application may need to find a service of a known service type on a device. The API `UPnP_ControlGetServicesByType ( )` searches for the service of a given type on the device. Upon success the service identifier of the service is returned a null is returned on failure, here is an example,

```
UPNP_CHAR* serviceId = UPnP_ControlGetServicesByType(dh, &i, "SwitchPower:1")
```

This Api is provide with device handle to the device containing this service, address of the service iterator and the target serviceType

#### 5.1.4h Release the service Iterator

At this point the service iterator can be freed using API `UPnP_ControlServiceIteratorDone ( )` as it is not required by the control point stack, example,

```
UPnP_ControlServiceIteratorDone(&i);
```

the address to the service Iterator initialized in step (5.1.5d). This api releases the resources used by the service iterator.

#### 5.1.4i Get the information about the device owning the service

The application developer needs to keep in mind that a device may contain one or more embedded devices within itself. A service of interest may be obtained on such a device using one of the above steps. But in case this service is owned by one the embedded device within the known root device, it will not provide the developer with information about the device owning the service. The application can obtain the information about the device owning the service with a known service Id using `UPnP_ControlGetServiceOwnerDeviceInfo ( )` api, here is a sample.

```
UPnP_ControlGetServiceOwnerDeviceInfo (dh, serviceId, &deviceInfo)
```

Where, dh is the device handle of the known root device, serviceId is the service identifier of the target service, and like in section 5.1.5c deviceInfo is address of structure of type `UPnPControlDeviceInfo` which will contain the device information on success.

#### 5.1.4j Obtaining Information in a Service

At this point the application has determined the service of interest on a device and has access to its service identifier. The next step is to obtain the description document that the device maintains for this service. API `UPnP_ControlGetServiceInfo ( )` can be used to synchronously (blocking) obtain an xml document containing service information, for example,

```
IXML_Document* xmlDoc;
```

```
xmlDoc = UPnP_ControlGetServiceInfo(dh, serviceId);
```

An asynchronous (non-blocking) version of this api is also available, `UPnP_ControlGetServiceInfoAsync ( )`. If the asynchronous version of the api is used, the application will have to catch the events of success or failure in obtaining service information in the application callback which is invoked internally by the stack in case a success or failure. In case of success a `UPNP_CONTROL_EVENT_SERVICE_INFO_READ` event is generated while a `UPNP_CONTROL_EVENT_SERVICE_GET_INFO_FAILED` is generated in case of failure.

The developer can use xml parser's `ixmlPrintDocument ( )` API to read the service descriptor as a string. This API takes in the dom (data object representation) tree representation of the xml document obtained from `UPnP_ControlGetServiceInfo ( )` and `UPnP_ControlGetServiceInfoAsync ( )` API and returns a buffer containing service descriptor. Here is a sample,

```
xmlDoc = UPnP_ControlGetServiceInfo(dh, serviceId);
if (xmlDoc)
{
    DOMString str = ixmlPrintDocument(xmlDoc);
    if (str)
    {
        printf("%s\n\n", str);
        ixmlFreeDOMString(str);
    }
    ixmlDocument_free(xmlDoc);
}
```

### 5.1.5 Controlling a Remote Device (Control Phase)

Control point is now ready to invoke actions to control services on a remote device. A UPnP device capable of being controlled by a UPnP control point contains one or more services, each of these services offer one or more actions which can be invoked by the control point.

It is important to keep in mind that each such 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.

At this point, the application knows about the following,

- Device handle of the device containing the service of interest.
- Service Id and service type of the service to be controlled on the device
- The target action and its arguments available in the service. This is obtained from examining `actionList` field of the service descriptor document obtained in step 5.1.5j. A service can offer one or more actions, invoking an action on a service of a remote device consists of sending the device an action request containing action name of one of the action and desired values of all its 'in' arguments.

A target action is controlled through its arguments which have 'in' direction. In order to invoke this action on the remote service, the application will create an action request in which it needs to assign a desired value to each of its arguments having 'in' direction. An important point to note here is UPnP specification requires the request to contain every 'in' argument in the definition of the action in the service description.

Here is the procedure to create an action request, see figure 6 for sample implementation, the contents of an action request are stored in xml format contained inside an envelope which is send to the remote service. The first step is to create this xml envelope, API `UPnP_CreateAction ( )` is used to constructs the request. Here is a sample,

```
IXML_Document* action;
action = UPnP_CreateAction(serviceType, "SetTarget");
```

This Api needs the service type of the target service as its first parameter and the name of the action to be invoked as its second parameter. The Api returns address of the xml dom tree holding the request.

The next step is to supply a desired value to each of the 'in' argument of the action, this argument and its supplied value is inserted in the action request body. The application should call `UPnP_SetActionArg ( )` API to perform this. Here is an example,

```
UPnP_SetActionArg (action, "newTargetValue", "true");
```

In the above example the API assigns a value "true" to an argument named "newTargetValue", this name value pair is then stored in the action request pointed by `IXML_Document` variable called `action`. The above API is used to store the name value pair of an argument to the action request.

This API should be called for each argument of an action having 'in' direction. As mentioned previously in this section, UPnP specification requires the request to contain a value for every 'in' argument that the action contains.

At this point the application has completed constructing the action request and is ready to send it to the remote device containing target service using `UPnP_ControlInvokeAction ( )` API, here is an example,

```
UPnP_ControlInvokeAction (dh, serviceId, actionName, action, userData, 1)
```

This API will invoke action on a service at a remote device, handle to this device is supplied by first parameter `dh`. The service on this device which contains this action is located by its service Identifier, `serviceId` which is the second argument to the API. The name of the action is supplied as the third argument, this is the target action on this service. The xml action request body is passed as the fourth argument, in the above example variable `action` holds address of the xml action request body. The next argument is the pointer to user data that will be passed to the application callback. The final parameter provides the application option of sending the action request synchronously (blocking) or asynchronously (non blocking), this boolean value should be set to true to make this a blocking call and false to make it non blocking.

If the device successfully invokes this action, the application callback is internally invoked and with a `UPnP_CONTROL_EVENT_ACTION_COMPLETE` event to indicate action completion.

Now, that the developer is done invoking the action on a remote device he must free up the xml message body for the request. Recall from previous step, UPnP\_CreateAction ( ) had created the xml message body and returned its address. To free it up use `ixmlDocument_free ( )`, here is a sample usage

```
ixmlDocument_free ( action );
```

Where, `action` holds the address of the xml message.

```
void controlTest (UPnPRuntime *rt, UPnPControlPoint *cp)
{
    UPnPControlDeviceHandle dh;
    char* deviceUrl = 0;
    int done;

    printf("\nSearching for services of type SwitchPower:1...\n");
    UPnP_ControlFindDevicesByService(cp, "SwitchPower:1", 2500, &deviceUrl, 1);

    if (deviceUrl)
    {
        printf("\nOpening device at %s...", deviceUrl);
        dh = UPnP_ControlOpenDevice(cp, deviceUrl);
        if (dh != 0)
        {
            UPnPControlServiceIterator i;

            printf("device open.\n\nLooking for SwitchPower:1 service...");

            /* find the SwitchPower:1 service and invoke an action on it */
            if (UPnP_ControlGetServicesByType(dh, &i, "SwitchPower:1") >= 0)
            {
                UPNP_CHAR* serviceId = UPnP_ControlNextService(&i);

                if (serviceId)
                {
                    IXML_Document* action;

                    printf("\n Found (serviceId: %.40s)\n\nInvoking action: SetTarget(true)...", serviceId);

                    action = UPnP_CreateAction(UPnP_ControlGetServiceType(dh, serviceId), "SetTarget");
                    if (action)
                    {
                        UPnP_SetActionArg (action, "newTargetValue", "true");

                        done = 0;

                        if (UPnP_ControlInvokeAction (dh, serviceId, "SetTarget", action, &done, 0) >= 0)
                        {
                            while (!done)
                            {
                                UPnP_ProcessState(rt, 200);
                            }
                            printf ("\nControl Test PASSED.\n");
                        }
                        /* free the xml document */
                        ixmlDocument_free (action);
                    }
                    else
                    {
                        printf (" could not create action!\n");
                    }
                }
            }
            else
            {

```

```

        printf (" service not found!\n");
    }
    UPnP_ControlServiceIteratorDone(&i);
}
else
{
    printf (" search init failed!\n");
}
UPnP_ControlCloseDevice(dh);
}
else
{
    printf (" open failed.\n");
}
rtp_strfree(deviceUrl);
}
}

```

Figure 6: Sample demonstration of Control Phase

### 5.1.6 Subscribing and Unsubscribing to a Service on a Device (Event Phase)

The application can subscribe to a service on a UPnP device. Once subscribed the control point receives event messages from the service notifying any change that may occur to its state.

This section describes the steps necessary for subscribing and unsubscribing to a service on the remote UPnP device.

Subscribing to a service is easily accomplished by using `UPnP_ControlSubscribe ( )` API. The application must possess the service identifier of the target service and device handle of the device containing the service using any of the steps described in section 5.1.5. Shown below is a sample sending subscription request,

```
UPnP_ControlSubscribe (dh, serviceId, timeoutMsec, userData, 0 ) ;
```

The first argument to this API is device handle, `dh`, to the UPnP device which contains the target service. The target service on this device is located with its service identifier supplied as `serviceId`, which is the second argument. Third argument is the timeout value in milliseconds, this is the period for which the subscription is valid. After the timeout period expires the device will unsubscribe this control point. The next argument is the pointer to user data that will be passed to the application callback. This subscription request can be sent synchronously (blocking) or asynchronously (non blocking), the final boolean argument determines if this call will be blocking(1) or non blocking(0) in nature.

The application callback is invoked internally to indicate success or failure of subscription request. In case this request is received by the device successfully `UPNP_CONTROL_EVENT_SUBSCRIPTION_ACCEPTED` event is passed to the callback. In case where the subscription request is rejected, callback is invoked with `UPNP_CONTROL_EVENT_SUBSCRIPTION_REJECTED` event to handle the case.

Keep in mind, a subscription is only valid for a given period (supplied as third argument in the above API). Upon expiration of this period UPnP specification requires the device to remove the control point from its subscriber's list. To take care of this issue EBS's control point stack provides the developer with an internal event which notifies the application that subscription period is about to expire. This is done by internally invoking the application callback with an `UPNP_CONTROL_EVENT_SUBSCRIPTION_NEAR_EXPIRATION` event. If application wants to be continuously subscribed to a service, it should call `UPnP_ControlSubscribe ( )` API on that service in order to refresh its subscription.

In case the subscription period expires and the device removes the control point from its subscriber list, this event is notified to the application through the application callback which is invoked with `UPNP_CONTROL_EVENT_SUBSCRIPTION_EXPIRED` event.

Similar to subscribing, a control point can easily unsubscribe to a service using `UPnP_ControlUnsubscribe ( )` API, for example

```
UPnP_ControlUnsubscribe (dh, serviceId, userData, 0 ) ;
```

The first argument to this API is device handle, `dh`, to the UPnP device which contains target service. The target service on this device is located with its service identifier supplied as `serviceId`, second argument. The

next argument is the pointer to user data that will be passed to the application callback. This subscription request can be sent synchronously (blocking) or asynchronously(non blocking), the final boolean argument determines if this call will a blocking(1) or non blocking(0) in nature.

The application callback is invoked internally with

UPNP\_CONTROL\_EVENT\_SUBSCRIPTION\_CANCELLED event to indicate unsubscription to the service.

### 5.1.7 Listening for Passive Events and Asynchronously Generated Active Events

Control point should always listen for all presence (alive) and bye-bye announcements. These are announcements which a UPnP device sends out to the multicast channel announcing its presence or departure from the network. Also, the application needs to catch and process any events generated as a result of an asynchronous operation.

Events such as presence or bye-bye announcements which are not generated as a result of a synchronous or asynchronous call made by the application are called passive events while events which are generated as a result of an API call made by the application are called active events.

API UPnP\_ProcessState ( ) is designed to block for at most a supplied duration of time, processing any asynchronous operations that may be in progress on the control point. This API must be called in order to receive events if an application is running with the UPnP SDK in single-threaded mode. UPnP\_ProcessState ( ) sets up the control point to listen for any passive events occurring within supplied time period. Shown below is its sample usage,

```
UPnP_ProcessState ( &upnpRuntime, 1000);
```

where, address of upnpRuntime structure is the first argument, and time in milliseconds for which this function blocks is the second argument. For example, if an alive announcement is detected during this period the control point invokes the application callback with an UPNP\_CONTROL\_EVENT\_DEVICE\_ALIVE event, in case an bye-bye announcement is detected the control point invokes the callback with a UPNP\_CONTROL\_EVENT\_DEVICE\_BYE\_BYE event.

### 5. 1. 8 Shutting Down

API UPnP\_ControlCloseDevice ( ) is designed to free up and release all the resources used up in storing information of a UPnP device, referenced by device handle dh, and all its services. Here is an example,

```
UPnP_ControlCloseDevice(dh);
```

This API takes the device handle to the device to be closed.

Next step, is to close the control point using UPnP\_ControlPointDestroy ( ) API. Here is how,

```
UPnP_ControlPointDestroy (&cp, gracefulTimeoutMsec);
```

where, the first argument is address to structure holding control point information and if, gracefulTimeoutMsec, the second argument, is nonzero, then this API will block if needed for at max gracefulTimeoutMsec waiting for any outstanding asynchronous operations to complete before closing down the control point.

The last step is to turn of the UPnP engine by calling UPnP\_RuntimeDestroy ( ) API. This is used as shown here,

```
UPnP_RuntimeDestroy (&upnpRuntime);
```

Address of UpnpRuntime structure which stores the run time information of the UPnP stack is supplied as the parameter.



## Section 6. Writing an Application Callback

### *Why an application callback?*

Application callback is the mechanism used by UPnP control point stack to notify the control point application of any event or state change that may occur on it during its lifetime. The developer needs to create a callback in order to handle the events which may be generate when the control point is operating.

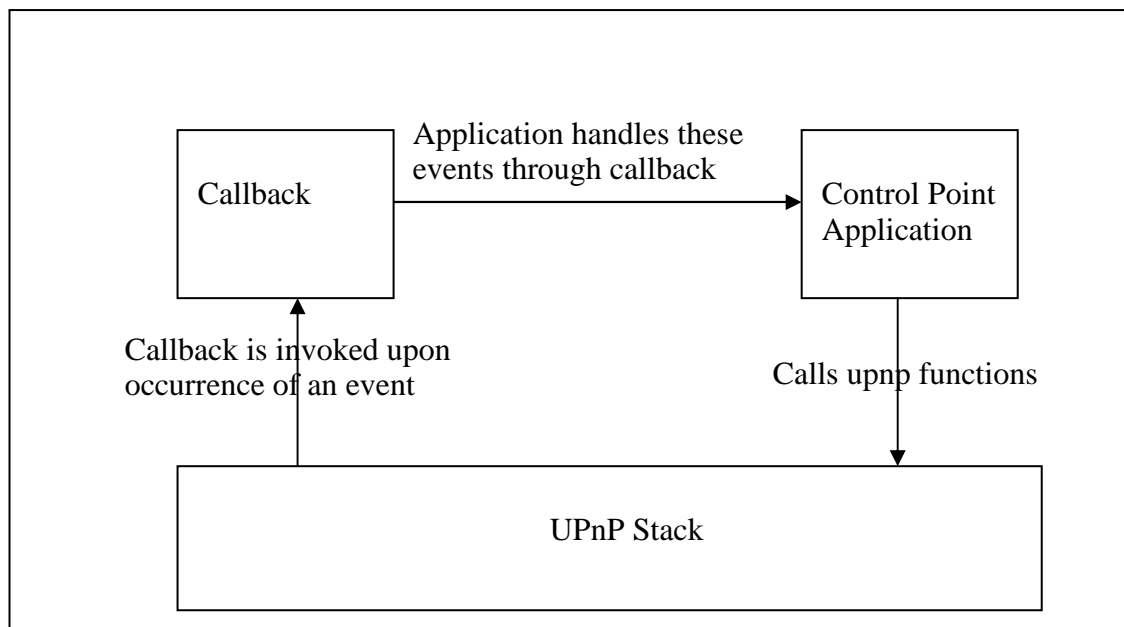


Figure 7: Shows interactions between control point application, UPnP stack and callback.

Figure 7 depicts the interactions between UPnP stack, application and the callback, figure 8. shows a sample implementation of the application callback

```

int controlPointCallback (
    UPnPControlPoint* cp,
    UPnPControlEvent* event,
    void *perControl,
    void *perRequest
)
{
    int result = 0;
    switch (event->type)
    {
        case UPNP_CONTROL_EVENT_DEVICE_FOUND:
        {
            char** str = (char**) perRequest;

            printf (" Device found at %.50s\n", event->data.discover.url);
            if(str) /* if callback data is passed */
            {
                *str = rtp_strdup(event->data.discover.url);
            }
            result = 1; // if 1 is returned no further searches will be performed
            break;
        }
    }
}
  
```

```

case UPNP_CONTROL_EVENT_SEARCH_COMPLETE:
{
    printf (" Search Complete.\n\n");
    result = 1; // perform no further searches
    break;
}

case UPNP_CONTROL_EVENT_DEVICE_ALIVE:
{
    printf ("\n\n Device alive at %.50s"
            "\n\n ST: %.55s"
            "\n USN: %.55s", event->data.discover.url, event->data.discover.type, event-
>data.discover.usn);
    break;
}

case UPNP_CONTROL_EVENT_DEVICE_BYE_BYE:
{
    printf ("\n\n Device bye-bye at %.50s"
            "\n\n ST: %.55s"
            "\n USN: %.55s", event->data.discover.url, event->data.discover.type, event-
>data.discover.usn);
    break;
}

case UPNP_CONTROL_EVENT_ACTION_COMPLETE:
{
    int* done = (int*) perRequest;
    UPNP_CHAR* outValue;
    if(event->data.action.success == 0)
    {
        printf("Action cannot be executed...\n");
        printf("Description: %s\n", event->data.action.errorDescription);
    }
    printf ("Action Complete.\n");
    if(done)
    {
        if(*done == -1)
        {
            *done = 1;
        }
    }
    /* Response to action 'GetMinLevel' contains output value for
    variable "MinLevel". */
    outValue = UPnP_GetPropertyValueByName(event->data.action.response, "MinLevel");
    if (outValue)
    {
        *done = rtp_atoi(outValue);
    }
    break;
}

case UPNP_CONTROL_EVENT_SERVICE_GET_INFO_FAILED:
{
    int* done = (int*) perRequest;
    printf ("UPnP_ControlGetServiceInfoAsync failed!");
    if(done)
    {
        *done = 1;
    }
    break;
}

```

```

}

case UPNP_CONTROL_EVENT_SERVICE_INFO_READ:
{
    int* done = (int*) perRequest;
    DOMString str = ixmlPrintDocument(event->data.service.scpdDoc);

    printf ("service info read:\n\n");

    if (str)
    {
        printf("%s\n\n", str);
        ixmlFreeDOMString(str);
    }
    ixmlDocument_free(event->data.service.scpdDoc);
    if(done)
    {
        *done = 1;
    }
    break;
}

case UPNP_CONTROL_EVENT_SUBSCRIPTION_ACCEPTED:
    printf("subscription accepted\n\n");
    printf("subscription id is: %s\n", event->data.subscription.serviceld);
    break;

case UPNP_CONTROL_EVENT_SUBSCRIPTION_CANCELLED:
    printf("service Unsubscribed\n\n");
    break;

case UPNP_CONTROL_EVENT_SUBSCRIPTION_REJECTED:
    printf("subscription rejected\n\n");
    break;

case UPNP_CONTROL_EVENT_SERVICE_STATE_UPDATE:
{
    const UPNP_CHAR* str = UPnP_GetPropertyValueByName(event->data.notify.stateVars,
"LoadLevelStatus");
    if (!str)
    {
        str = "[unknown]";
    }
    printf ("Notification: Load level is =%s\n", str);
    break;
}

case UPNP_CONTROL_EVENT_SUBSCRIPTION_NEAR_EXPIRATION:
    printf ("Subscription Expiration Warning!\n");
    break;

case UPNP_CONTROL_EVENT_SUBSCRIPTION_EXPIRED:
    printf ("Subscription Expired.\n");
    break;
}
return (result);
}

```

Figure 8. Sample of an application callback

Here is how the callback function is defined

```
typedef int (*UPnPControlPointCallback) (
    UPnPControlPoint* cp,
    UPnPControlEvent* event,
    void* userControlPointData,
    void* userRequestData
);
```

Take a note of the new structure introduced at this point which is UPnPControlEvent, this is passed as the second argument to the callback. Structure UPnPControlEvent holds the information needed to handle the event responsible for invoking the callback. In other words, whenever an event which may need to be handled by the application occurs in the stack, it invokes the callback passing information needed to handle it in UPnPControlEvent structure. Let us now study the internal fields of this structure,

```
struct s_UPnPControlEvent
{
    UPnPControlEventType    type;
    union
    {
        struct
        {
            const UPNP_CHAR*    url;        // valid until event handler returns
            const UPNP_CHAR*    type;      // valid until event handler returns
            const UPNP_CHAR*    usn;       // valid until event handler returns
        }
        discover;
        struct
        {
            UPnPControlDeviceHandle handle; // valid until UPnP_ControlCloseDevice is called
            const UPNP_CHAR*    url;        // valid until event handler returns
        }
        device;
        struct
        {
            UPnPControlDeviceHandle deviceHandle;
            const UPNP_CHAR*    id;
            IXML_Document*    scpDoc;    // now owned by the application
        }
        service;
        struct
        {
            UPNP_BOOL        success;
            IXML_Document*    response;    // valid until event handler returns
            UPNP_INT32        errorCode;
            UPNP_CHAR*        errorDescription; // valid until event handler returns
        }
        action;
        struct
        {
            UPnPControlDeviceHandle deviceHandle; // valid until event handler returns
            const UPNP_CHAR*    serviceId;    // valid until event handler returns
            UPNP_INT32        timeoutSec;
        }
        subscription;
        struct
        {
            IXML_Document*    stateVars;    // now owned by the event handler
            UPnPControlDeviceHandle deviceHandle; // valid until event handler returns
            const UPNP_CHAR*    serviceId;    // valid until event handler returns
        }
    }
};
```

```

    }
    notify;
}
data;
};

```

It can be observed above that this structure contains two main fields. One is the member 'type' which is of UPnPControlEventType, any one of the possible events enumerated by UPnPControlEventType may have occurred. The member 'type' holds the event type which is responsible for callback invocation. All the possible event types are enumerated by UPnPControlEventType which is defined as follows,

```

enum e_UPnPControlEventType
{
    UPNP_CONTROL_EVENT_ACTION_COMPLETE = 0,
    UPNP_CONTROL_EVENT_DEVICE_ALIVE,
    UPNP_CONTROL_EVENT_DEVICE_BYE_BYE,
    UPNP_CONTROL_EVENT_DEVICE_FOUND,
    UPNP_CONTROL_EVENT_DEVICE_OPEN,
    UPNP_CONTROL_EVENT_DEVICE_OPEN_FAILED,
    UPNP_CONTROL_EVENT_NONE,
    UPNP_CONTROL_EVENT_SEARCH_COMPLETE,
    UPNP_CONTROL_EVENT_SERVICE_GET_INFO_FAILED,
    UPNP_CONTROL_EVENT_SERVICE_INFO_READ,
    UPNP_CONTROL_EVENT_SERVICE_STATE_UPDATE,
    UPNP_CONTROL_EVENT_SUBSCRIPTION_ACCEPTED,
    UPNP_CONTROL_EVENT_SUBSCRIPTION_CANCELLED,
    UPNP_CONTROL_EVENT_SUBSCRIPTION_EXPIRED,
    UPNP_CONTROL_EVENT_SYNCHRONIZE_FAILED,
    UPNP_CONTROL_EVENT_SUBSCRIPTION_NEAR_EXPIRATION,
    UPNP_CONTROL_EVENT_SUBSCRIPTION_OUT_OF_SYNC,
    UPNP_CONTROL_EVENT_SUBSCRIPTION_REJECTED,
    UPNP_NUM_CONTROL_EVENT_TYPES
};

```

The second important element of the UPnPControlEvent structure is the union named 'data'. The idea behind this union is that at any time it only holds data specific to handle the type of event contained in field 'type'. To elaborate this point notice that the union 'data' contains 6 structures, which are discover, device, service, action, subscription and notify. When a callback is invoked one out these 6 structures holds information to handle the event type specified by 'type' field.

We will now take a detailed look at individual events and the structures holding their information. We use two terms when describing occurrence of an event, active and passive. An event may occur actively when it is result of a command issued by the application. An event is passive when it is not caused due to a locally initiated action or command.

The fields of structure 'discover' hold information in case any one of the following 4 events occur

1. UPNP\_CONTROL\_EVENT\_DEVICE\_ALIVE
2. UPNP\_CONTROL\_EVENT\_DEVICE\_BYE\_BYE
3. UPNP\_CONTROL\_EVENT\_DEVICE\_FOUND
4. UPNP\_CONTROL\_EVENT\_SEARCH\_COMPLETE

Event 1 occur passively when ever the control point stack receives alive message sent by a device announcing its arrival on the network bye-bye messages, similarly event 2 passively occurs when a control point stack receives bye-bye messages sent by a devices leaving the network. Event 3. is an active event which occurs as a result of a receiving a response from a device which matched the search criteria of a search initiated by the control point. Event 4 occurs actively when the time period specified for a search

which was initiated by the control point application elapses. This event indicates that the search is complete.

Structure 'device' holds information when the callback contain event of types

- 5. UPNP\_CONTROL\_EVENT\_DEVICE\_OPEN
- 6. UPNP\_CONTROL\_EVENT\_DEVICE\_OPEN\_FAILED

Event 5 occurs actively indicating the success of a device open operation was initiated by the control point application. Event 6 is an active event indicating failure of device open operation.

The fields of structure service contain information when events of following type occurs,

- 7. UPNP\_CONTROL\_EVENT\_SERVICE\_INFO\_READ
- 8. UPNP\_CONTROL\_EVENT\_SERVICE\_GET\_INFO\_FAILED

Where, event 7 occurs actively and indicates success in acquiring the service description document for a service on which a service get info operation was initiated by the control point. This xml description document is stored in the scpdDoc field of service structure.

Event 8 indicates that service get info operation initiated by the control point has failed in obtaining the service description document from the service.

Next structure is 'action' which holds information for event of type

- 9. UPNP\_CONTROL\_EVENT\_ACTION\_COMPLETE

This is an active event which occurs when a device sends a response indicating success or failure of an action requested on it. In case of success the boolean success field in the 'action' structure is set to true, in case the request fails fields errorCode and errorDescription contain error information.

If the action, on the device side, contained any state variable in 'OUT' direction, then the response will contain argument value for all such 'OUT' variables. To extract the value of 'OUT' variable from action response (event->data.action.response), use API UPnP\_GetPropertyValueByName ( ) or UPnP\_GetPropertyValueByIndex ( ), for example

```
Const UPNP_CHAR* value = UPnP_GetPropertyValueByName(
    event->data.action.response,
    "MinLevel");
```

where, first parameter is the address of xml document containing the action response, stored in data.action.response field in this case, second parameter is the name of the state variable.

Structure 'subscribe' holds information for subscribe phase events of type

- 10. UPNP\_CONTROL\_EVENT\_SUBSCRIPTION\_ACCEPTED
- 11. UPNP\_CONTROL\_EVENT\_SUBSCRIPTION\_REJECTED
- 12. UPNP\_CONTROL\_EVENT\_SUBSCRIPTION\_NEAR\_EXPIRATION
- 13. UPNP\_CONTROL\_EVENT\_SUBSCRIPTION\_EXPIRED
- 14. UPNP\_CONTROL\_EVENT\_SUBSCRIPTION\_CANCELLED
- 15. UPNP\_CONTROL\_EVENT\_SYNCHRONIZE\_FAILED
- 16. UPNP\_CONTROL\_EVENT\_SUBSCRIPTION\_OUT\_OF\_SYNC

Event 10 occurs actively as a result of a subscription request by the control point application. This event indicates successful establishment of subscription with the target service. After a subscription is established state update events will begin to arrive from the subscribed service.

Event 11 occurs as a result of subscription request to a service. This indicates that the request to subscribe on a service has failed. Event 12 is a passive event which is generated internally by EBS's control point stack to inform the control point application that the duration of its subscription to a service is about to expire. If application wants to be continuously subscribed to a service it can do so by subscribing again to this service at this point.

Event 13 occurs passively to indicate that the a expiration period of subscription to a service has been reached and the control point has been unsubscribed by the device. Event 14 occurs actively when an operation to unsubscribe from a service initiated by control point is completed successfully. Event 15 is an active event which occurs when an out-of-sync event generated by device prompts efforts from the control point stack to try to re-synchronize the subscription; this event indicates the re-synchronization has failed and the subscription to the service is no longer valid. Event number 16 is a passive event which informs the control point that the sequence numbers for events from a subscribed service are out of order. To handle this control point must choose to a) ignore the error, b) cancel the subscription, or c) allow UPnP stack to automatically attempt to re-synchronize.

Final structure is 'notify' which contains information received from subscribed service whenever next event occurs,

#### 17. UPNP\_CONTROL\_EVENT\_SERVICE\_STATE\_UPDATE

This event is passively generated whenever the control point receives an event notification message indicating a change in state of a service that this control point is subscribed to.

In this case, the stateVars field of the notify structure holds the name and changed value of the state variable send by the service.

To extract the value of a state variable, use API UPnP\_GetPropertyValueByName ( ) or UPnP\_GetPropertyValueByIndex ( ) , for example

```
const UPNP_CHAR* str = UPnP_GetPropertyValueByName (
    event->data.notify.stateVars,
    "LoadLevelStatus" );
```

where, first parameter is the address of xml document containing the notification, stored in data.notify.stateVars field in this case, second parameter is the name of the state variable which is "LoadLevelStatus" in the above sample. If the notification does not contains the state variable which is supplied as second parameter, the API return a null string.

Let us now come back to the defination of callback, shown below is sample callback function declaration

```
int controlPointCallback (
    UPnPControlPoint* controlPoint,
    UPnPControlEvent* event,
    void *perControl,
    void *perRequest )
{
    ..... callback body.....
}
```

Here, the first argument is address to structure holding control point information, second argument holds address to controlEvent structure which is explained above in this section. The third argument 'perControl' is address of control point cookie (this is the user data which was earlier supplied as 4th parameter while initializing the control point using UPnP\_ControlPointInit ( ) API ). Last parameter perRequest holds the address of the cookie which is passed by control point application when making any aysnchnorous calls. The name of this variable is chosen to reflect the nature of this cookie, unlike 'perControl' which is valid through out the life of control point, 'perRequest' user data is only valid when the callback is invoked as a result of an active event which is generated due to a call which supplied this user data. For example, if UPnP\_ControlFindAllDevices (&controlPoint, timeoutMsec, userData, waitForCompletion) is called, the userData supplied as third parameter to this call will be passed to 'perRequest' when the callback is invoked as a result of UPNP\_CONTROL\_EVENT



## PART II – Porting and Configuration Guide

---

---

### EBS UPnP SDK

### Porting and Configuration

*Revised July 2006*

---

---

Copyright © 2006 EBS Inc.



## 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. http.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 Http.c Httpsrv.c  Http files used by upnp control point: Fileext.c Filetype.c Http.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.

			<p>Two special subdirectories are available</p> <ol style="list-style-type: none"> <li>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</li> <li>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.</li> </ol> <p>Following RTPlatform source files are used by UPnP SDK</p> <p>Rtpchar.c Rtpdate.c Rtpdebug.c Rtpdobj.c Rtpdutil.c Rtpfile.c Rtphelper.c Rtpmem.c Rtpnet.c Rtpnetism.c Rtpprint.c Rtptrand.c Rtpscnv.c Rtpsignl.c Rtpstdup.c Rtpstr.c Rtpthrd.c Rtptime.c</p>
	doc	RTPlatform reference documentation in html format	RTPlatform reference documentation in html format can be accessed through <a href="doc/html/index.html">doc/html/index.html</a>



## 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

Rtpnetism.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 Control Point API Reference Manual

---

---

# EBS UPnP Control Point SDK

## API Reference Manual

*Revised July 2006*

---

---

Copyright © 2006 EBS Inc.



## 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 Control Point 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_ControlPointInit	Initialize a UPnP control point context
UPnP_ControlPointDestroy	Destroy a UpnPControlPoint
UPnP_ControlFindAll	Search for UPnP devices
UPnP_ControlFindAllDevices	Search for UPnP devices
UPnP_ControlFindDevicesByType	Search for UPnP devices of a certain type
UPnP_ControlFindDevicesByUUID	Search for a particular UPnP device
UPnP_ControlFindDevicesByService	Search for UPnP devices that offer a certain service
UPnP_ControlOpenDevice	Open a remote device for description, control, and eventing
UPnP_ControlOpenDeviceAsync	Asynchronously open a remote device for description, control, and eventing.
UPnP_ControlCloseDevice	Close an open device handle
UPnP_ControlGetDeviceInfo	Get information for an open device
UPnP_ControlGetServiceOwnerDeviceInfo	Get information for the parent device of a service
UPnP_ControlGetServiceType	Get the type of a service
UPnP_ControlGetServices	Enumerate the services on a device
UPnP_ControlGetServicesByType	Enumerate the services of a certain type on a device
UPnP_ControlNextService	Enumerate the next service on the device
UPnP_ControlServiceIteratorDone	Clean up when done enumerating services
UPnP_ControlGetServiceInfo	Get detailed information about a service
UPnP_ControlGetServiceInfoAsync	Asynchronous get detailed information about a service
UPnP_ControlInvokeAction	Invoke an action on a remote service

UPnP_ControlSubscribe	Subscribe to a service or renew a subscription
UPnP_ControlSubscribedToService	Return whether or not the control point is subscribed to the given service
UPnP_ControlSetServiceSubscriptionExpirationWarning	Sets the time offset before subscription expiration at which a warning event will be generated
UPnP_ControlUnsubscribe	Cancel a subscribed service
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

## 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\_ControlPointInit

### **FUNCTION**

Initialize a UPnP Control Point context.

### **SUMMARY**

```
int UPnP_ControlPointInit ( UPnPControlPoint* cp, UPnPRuntime* rt, UPnPControlPointCallback callbackFn,
                           void* callbackData )
```

UPnPControlPoint* cp	Pointer to control point context instance
UPnPRuntime* rt	UPnP runtime to associate with this control point
UPnPControlPointCallback callbackFn	Callback function for this control point
void* callbackData	Application-specific data which will be passed into the callback

### **DESCRIPTION**

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

### **RETURNS**

0	Operation was a success
-1	Operation failed

### **SEE ALSO**

UPnP\_ControlPointDestroy ( )

## 2.9 UPnP\_ControlPointDestroy

### **FUNCTION**

Destroy a UPnP control point's runtime context.

### **SUMMARY**

```
void UPnP_ControlPointDestroy ( UPnPControlPoint* cp, UPNP_INT32 gracefulTimeoutMsec )
```

UPnPControlPoint* cp	Address of control point to destroy
UPNP_INT32 gracefulTimeoutMsec	If this control point has any outstanding operations, wait for this many milliseconds to allow them to complete gracefully. After timeout expires, do hard close.

### **DESCRIPTION**

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

### **RETURNS**

None

### **SEE ALSO**

UPnP\_ControlPointInit ( )

## 2.10 UPnP\_ControlFindAll

### **FUNCTION**

Search for UPnP devices.

### **SUMMARY**

```
int UPnP_ControlFindAll ( UPnPControlPoint* cp, UPNP_INT32 timeoutMsec, void* userData,
                          UPNP_BOOL waitForCompletion )
```

UPnPControlPoint* cp	Control point, initialized by UPnP_ControlPointInit
UPNP_INT32 timeoutMsec	Total duration of time, in milliseconds , for the search
void* userData	Passed to callback as userRequestData for UPNP_CONTROL_EVENT_DEVICE_FOUND event
UPNP_BOOL waitForCompletion	If UPNP_TRUE, this function will not return until the search completes; else the function will return immediately after sending the multicast search request

### **DESCRIPTION**

Sends a request for all UPnP devices on the network to make their presence known to the control point. If this search method is used, then separate UPNP\_CONTROL\_EVENT\_DEVICE\_FOUND events will be generated for each root device, embedded device, and service that responds.

When the timeout has been reached, a UPNP\_CONTROL\_EVENT\_SEARCH\_COMPLETE event will be sent to the control point.

Generates: UPNP\_CONTROL\_EVENT\_DEVICE\_FOUND,  
UPNP\_CONTROL\_EVENT\_SEARCH\_COMPLETE

### **RETURNS**

0	Operation was a success
-1	Operation failed

### **SEE ALSO**

UPnP\_ControlFindAllDevices ( )

## 2.11 UPnP\_ControlFindAllDevices

### **FUNCTION**

Search for UPnP devices.

### **SUMMARY**

int UPnP\_ControlFindAllDevices ( UPnPControlPoint\* cp, UPNP\_INT32 timeoutMsec, void\* userData, UPNP\_BOOL waitForCompletion )

UPnPControlPoint* cp	Control point, initialized by UPnP_ControlPointInit
UPNP_INT32 timeoutMsec	Total duration of time, in milliseconds , for the search
void* userData	Passed to callback as userRequestData for UPNP_CONTROL_EVENT_DEVICE_FOUND event
UPNP_BOOL waitForCompletion	If UPNP_TRUE, this function will not return until the search completes; else the function will return immediately after sending the multicast search request

### **DESCRIPTION**

Sends a request for all UPnP devices on the network to make their presence known to the control point. Only one UPNP\_CONTROL\_EVENT\_DEVICE\_FOUND event per root device will be generated if this function is used.

When the timeout has been reached, a UPNP\_CONTROL\_EVENT\_SEARCH\_COMPLETE event will be sent to the control point.

Generates: UPNP\_CONTROL\_EVENT\_DEVICE\_FOUND,  
UPNP\_CONTROL\_EVENT\_SEARCH\_COMPLETE

### **RETURNS**

0	Operation was a success
-1	Operation failed

### **SEE ALSO**

UPnP\_ControlFindAll ( )

## 2.12 UPnP\_ControlFindDevicesByType

### **FUNCTION**

Search for UPnP devices of a certain type.

### **SUMMARY**

```
int UPnP_ControlFindDevicesByType ( UPnPControlPoint* cp, UPNP_CHAR* deviceType,
                                     UPNP_INT32 timeoutMsec, void* userData,
                                     UPNP_BOOL waitForCompletion )
```

UPnPControlPoint* cp	Control point, initialized by UPnP_ControlPointInit
UPNP_CHAR* deviceType	Device type to search for, as specified by the UPnP Forum
UPNP_INT32 timeoutMsec	Total duration of time, in milliseconds , for the search
void* userData	Passed to callback as userRequestData for UPNP_CONTROL_EVENT_DEVICE_FOUND event
UPNP_BOOL waitForCompletion	If UPNP_TRUE, this function will not return until the search completes; else the function will return immediately after sending the multicast search request

### **DESCRIPTION**

Sends a request for all UPnP devices of a certain type on the network to make their presence known to the control point. One UPNP\_CONTROL\_EVENT\_DEVICE\_FOUND event per device that matches the search type will be generated if this function is used.

When the timeout has been reached, a UPNP\_CONTROL\_EVENT\_SEARCH\_COMPLETE event will be sent to the control point.

Generates: UPNP\_CONTROL\_EVENT\_DEVICE\_FOUND,  
UPNP\_CONTROL\_EVENT\_SEARCH\_COMPLETE

### **RETURNS**

0	Operation was a success
-1	Operation failed

### **SEE ALSO**

UPnP\_ControlFindDevicesByUUID ( ), UPnP\_ControlFindDevicesByService ( )

## 2.13 UPnP\_ControlFindDevicesByUUID

### **FUNCTION**

Search for a particular UPnP device.

### **SUMMARY**

```
int UPnP_ControlFindDevicesByUUID ( UPnPControlPoint* cp, UPNP_CHAR* uuid,
                                     UPNP_INT32 timeoutMsec, void* userData,
                                     UPNP_BOOL waitForCompletion )
```

UPnPControlPoint* cp	Control point, initialized by UPnP_ControlPointInit
UPNP_CHAR* uuid	UUID to search for, as specified by the UPnP Forum
UPNP_INT32 timeoutMsec	Total duration of time, in milliseconds , for the search
void* userData	Passed to callback as userRequestData for UPNP_CONTROL_EVENT_DEVICE_FOUND event
UPNP_BOOL waitForCompletion	If UPNP_TRUE, this function will not return until the search completes; else the function will return immediately after sending the multicast search request

### **DESCRIPTION**

Sends a request for UPnP devices with the given UUID(unique identifier) to make their presence known to the control point. One UPNP\_CONTROL\_EVENT\_DEVICE\_FOUND event per device that matches the search type will be generated if this function is used.

When the timeout has been reached, a UPNP\_CONTROL\_EVENT\_SEARCH\_COMPLETE event will be sent to the control point.

Generates: UPNP\_CONTROL\_EVENT\_DEVICE\_FOUND,  
UPNP\_CONTROL\_EVENT\_SEARCH\_COMPLETE

### **RETURNS**

0	Operation was a success
-1	Operation failed

### **SEE ALSO**

UPnP\_ControlFindDevicesByType ( ), UPnP\_ControlFindDevicesByService ( )

## 2.14 UPnP\_ControlFindDevicesByService

### FUNCTION

Search for UPnP devices that offer a certain service.

### SUMMARY

```
int UPnP_ControlFindDevicesByService ( UPnPControlPoint* cp, UPNP_CHAR* serviceType,
                                       UPNP_INT32 timeoutMsec, void* userData,
                                       UPNP_BOOL waitForCompletion )
```

UPnPControlPoint* cp	Control point, initialized by UPnP_ControlPointInit
UPNP_CHAR* serviceType	Service type to search for, as specified by the UPnP Forum
UPNP_INT32 timeoutMsec	Total duration of time, in milliseconds , for the search
void* userData	Passed to callback as userRequestData for UPNP_CONTROL_EVENT_DEVICE_FOUND event
UPNP_BOOL waitForCompletion	If UPNP_TRUE, this function will not return until the search completes; else the function will return immediately after sending the multicast search request

### DESCRIPTION

Sends a request for UPnP devices with one or more services of the given type to make their presence known to the control point. One UPNP\_CONTROL\_EVENT\_DEVICE\_FOUND event per device that matches the search type will be generated if this function is used.

When the timeout has been reached, a UPNP\_CONTROL\_EVENT\_SEARCH\_COMPLETE event will be sent to the control point.

Generates: UPNP\_CONTROL\_EVENT\_DEVICE\_FOUND,  
UPNP\_CONTROL\_EVENT\_SEARCH\_COMPLETE

### RETURNS

0	Operation was a success
-1	Operation failed

### SEE ALSO

UPnP\_ControlFindDevicesByType ( ), UPnP\_ControlFindDevicesByUUID ( )

## 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 targeted 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 response 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 Control Point Example

#### Setting up a UPnP Control Point

This example code demonstrates in brief the steps necessary to set up EBS's UPnP Control Point and use it for discovery, description, control, and eventing.

```

int main (int* argc, char* agrv[])
{
    int result;
    UPnPRuntime rt;
    UPnPControlPoint cp;

    GLOBAL_FPTR = fopen("memory_log", "w+");
    rtp_net_init();
    rtp_threads_init();

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

    if (result >= 0)
    {
        /* Initialize the control point */
        if (UPnP_ControlPointInit(&cp, &rt, controlPointCallback, 0) >= 0)
        {
            // discoverTest (&rt, &cp);
            /* listen passively for any announcements. does not actively search for any device. */
            // passiveDiscoverTest (&rt, &cp);
            // deviceOpenTest (&rt, &cp);
            // deviceOpenTestAsync (&rt, &cp);
            // controlTest (&rt, &cp);
            // controlTest2 (&rt, &cp);
            genaTest (&rt, &cp);
            // serviceDescribeTest (&rt, &cp);
            UPnP_ControlPointDestroy (&cp, 0);
        }

        UPnP_RuntimeDestroy(&rt);
    }
    printf("hit any key to exit this demo..\n");
    getch();
    rtp_threads_shutdown();
    rtp_net_exit();
    fclose(GLOBAL_FPTR);
    return(0);
}

/*-----*/

```

```

void discoverTest (UPnPRuntime *rt, UPnPControlPoint *cp)
{
    RTP_UINT32 startTime;

    printf("Searching for all...\n\n");
    UPnP_ControlFindAll(cp, 5000, 0, 1);

    printf("\nSearching for all UPnP devices...\n\n");
    UPnP_ControlFindAllDevices(cp, 5000, 0, 1);

    printf("\nSearching for devices of type InternetGatewayDevice:1...\n\n");
    UPnP_ControlFindDevicesByType(cp, "InternetGatewayDevice:1", 5000, 0, 1);

    printf("\nSearching for services of type SwitchPower:1...\n\n");
    UPnP_ControlFindDevicesByService(cp, "SwitchPower:1", 5000, 0, 1);

    printf("\nSearching for devices with uuid: upnp-WANDevice-1_0-0090a2777777...\n\n");
    UPnP_ControlFindDevicesByUUID(cp, "upnp-WANDevice-1_0-0090a2777777", 5000, 0, 1);

    printf("\nSearching Asynchronously for all...\n\n");
    // UPnP_ControlFindAll(cp, 5000, &done, 0);

    startTime = rtp_get_system_msec();

    while (rtp_get_system_msec() - startTime < 5000)
    {
        printf("(time: %dms)\n", rtp_get_system_msec() - startTime);
        UPnP_ProcessState(rt, 100);
    }
}

/*-----*/
void passiveDiscoverTest (UPnPRuntime *rt, UPnPControlPoint *cp)
{
    RTP_UINT32 startTime;

    printf ("Listening passively for device announcements.");

    startTime = rtp_get_system_msec();

    while (rtp_get_system_msec() - startTime < 60000)
    {
        printf(".");
        UPnP_ProcessState(rt, 100);
    }
}

/*-----*/
void deviceOpenTest (UPnPRuntime *rt, UPnPControlPoint *cp)
{
    UPnPControlDeviceHandle dh;
    printf("Opening device at http://[fe80::20b:dbff:fe2f:c162]...");
    dh = UPnP_ControlOpenDevice (cp, "http://[fe80::20b:dbff:fe2f:c162]:4087/device.xml");
    if (dh != 0)
    {
        UPnPControlDeviceInfo info;
        UPnPControlServiceIterator i;

        UPnP_ControlGetDeviceInfo(dh, &info);

        printf (" success!\n\n")

```

```

    " deviceType:    %.60s\n"
    " friendlyName:  %.60s\n"
    " manufacturer:  %.60s\n"
    " manufacturerURL: %.60s\n"
    " modelDescription: %.60s\n"
    " modelName:     %.60s\n"
    " modelNumber:   %.60s\n"
    " modelURL:      %.60s\n"
    " serialNumber:  %.60s\n"
    " UDN:           %.60s\n"
    " UPC:           %.60s\n"
    " presentationURL: %.60s\n\n",
    info.deviceType,
    info.friendlyName,
    info.manufacturer,
    info.manufacturerURL,
    info.modelDescription,
    info.modelName,
    info.modelNumber,
    info.modelURL,
    info.serialNumber,
    info.UDN,
    info.UPC,
    info.presentationURL);

if (UPnP_ControlGetServices(dh, &i) >= 0)
{
    UPNP_CHAR* serviceId = UPnP_ControlNextService(&i);

    while (serviceId)
    {
        printf ("Found Service: %.60s\n", serviceId);
        printf (" (type: %.65s)\n", UPnP_ControlGetServiceType(dh, serviceId));
        serviceId = UPnP_ControlNextService(&i);
    }

    UPnP_ControlServiceIteratorDone(&i);
}

UPnP_ControlCloseDevice(dh);
}
else
{
    printf (" open failed.\n");
}

printf("\n");
genaTest (rt, cp);
}

/*-----*/
void deviceOpenTestAsync (UPnPRuntime *rt, UPnPControlPoint *cp)
{
    int done = 0;
    printf("Asynchronously Opening device at http://192.168.0.6:4754/...");
    UPnP_ControlOpenDeviceAsync(cp, "http://192.168.0.6:4754/device.xml", &done);

    while (!done)
    {
        printf(".");
        UPnP_ProcessState(rt, 100);
    }
}

```

```

}
}
/*-----*/
void controlTest2(UPnPRuntime *rt, UPnPControlPoint *cp)
{
    UPnPControlDeviceHandle dh;
    char* deviceUrl = 0;
    int done = -1;

    printf("\nSearching for services of type DimmingService:1...\n");
    UPnP_ControlFindDevicesByService(cp, "DimmingService:1", 2500, &deviceUrl, 1);

    if (deviceUrl)
    {
        printf("\nOpening device at %s...", deviceUrl);
        dh = UPnP_ControlOpenDevice(cp, deviceUrl);
        if (dh != 0)
        {
            UPnPControlServiceIterator i;
            printf("device open.\n\nLooking for DimmingService:1 service...");

            /* find the DimmingService:1 service and invoke an action on it */
            if (UPnP_ControlGetServicesByType(dh, &i, "DimmingService:1") >= 0)
            {
                UPNP_CHAR* serviceId = UPnP_ControlNextService(&i);
                if (serviceId)
                {
                    IXML_Document* action;
                    /* Action 'GetMinLevel' on DimmingService:1 does not
                    contain any 'in' arguments. Invoking 'GetMinLevel' action
                    in order to get the value of MinLevel ('out') variable */

                    action = UPnP_CreateAction(
                        UPnP_ControlGetServiceType(dh,serviceId),"GetMinLevel");
                    if (action)
                    {
                        /* Since action 'GetMinLevel' does not contain any 'in'
                        argument, there is no need to set any argument value in
                        this action request
                        i.e. no need for UPnP_SetActionArg ( )*/

                        if (UPnP_ControlInvokeAction (dh, serviceId, "GetMinLevel", action, &done, 1) >= 0)
                        {
                            while (done < 0)
                            {
                                UPnP_ProcessState(rt, 200);
                            }
                            printf ("\nControl Test PASSED.\n");
                        }
                        /* free the xml document */
                        ixmlDocument_free (action);
                    }
                    else
                    {
                        printf (" could not create action!\n");
                    }
                }
            }
        }
    }
}
}
}
}
}

```

```

/*-----*/
void controlTest (UPnPRuntime *rt, UPnPControlPoint *cp)
{
    UPnPControlDeviceHandle dh;
    char* deviceUrl = 0;
    int done;

    printf("\nSearching for services of type SwitchPower:1...\n");
    UPnP_ControlFindDevicesByService(cp, "SwitchPower:1", 2500, &deviceUrl, 1);

    if (deviceUrl)
    {
        printf("\nOpening device at %s...", deviceUrl);
        dh = UPnP_ControlOpenDevice(cp, deviceUrl);
        if (dh != 0)
        {
            UPnPControlServiceIterator i;

            printf("device open.\n\nLooking for SwitchPower:1 service...");

            /* find the SwitchPower:1 service and invoke an action on it */
            if (UPnP_ControlGetServicesByType(dh, &i, "SwitchPower:1") >= 0)
            {
                UPNP_CHAR* serviceId = UPnP_ControlNextService(&i);

                if (serviceId)
                {
                    IXML_Document* action;

                    printf("\n Found (serviceId: %.40s)\n\nInvoking action: SetTarget(true)...", serviceId);

                    action = UPnP_CreateAction(UPnP_ControlGetServiceType(dh, serviceId), "SetTarget");

                    if (action)
                    {
                        UPnP_SetActionArg (action, "newTargetValue", "1");

                        done = 0;
                        if (UPnP_ControlInvokeAction (dh, serviceId, "SetTarget", action, &done, 1) >= 0)
                        {
                            while (!done)
                            {
                                UPnP_ProcessState(rt, 200);
                            }
                            printf ("\nControl Test PASSED.\n");
                        }
                        /* free the xml document */
                        ixmIDocument_free (action);
                    }
                    else
                    {
                        printf (" could not create action!\n");
                    }
                }
                else
                {
                    printf (" service not found!\n");
                }

                UPnP_ControlServiceIteratorDone(&i);
            }
        }
    }
}

```

```

    else
    {
        printf (" search init failed!\n");
    }

    UPnP_ControlCloseDevice(dh);
}
else
{
    printf (" open failed.\n");
}

    rtp_strfree(deviceUrl);
}
}

/*-----*/
void genaTest (UPnPRuntime *rt, UPnPControlPoint *cp)
{
    int done = 0;
    UPnPControlDeviceHandle dh;
    char* deviceUrl = 0;
    printf("\nSearching for services of type SwitchPower:1...\n");
    UPnP_ControlFindDevicesByService(cp, "SwitchPower:1", 2500, &deviceUrl, 1);
    if (deviceUrl)
    {
        printf("\nOpening device at %s...", deviceUrl);
        dh = UPnP_ControlOpenDevice(cp, deviceUrl);
        if (dh != 0)
        {
            UPnPControlServiceIterator i;

            printf("device open.\n\nLooking for SwitchPower:1 service...");

            /* find the tvcontrol:1 service and subscribe it */
            if (UPnP_ControlGetServicesByType(dh, &i, "SwitchPower:1") >= 0)
            {
                UPNP_CHAR* serviceId = UPnP_ControlNextService(&i);

                if (serviceId)
                {
                    printf("found.\n\nSubscribing to %s...\n\n", serviceId);
                    if (UPnP_ControlSubscribe (
                        dh,
                        serviceId,
                        500,
                        0,
                        0
                    ) >=0)
                    {
                        while(!done)
                        {
                            printf("\t hit u to unsubscribe\n\n");
                            printf("\t hit s to subscribe\n\n");
                            printf("\t hit x to exit\n\n");
                            while (!kbhit())
                            {
                                UPnP_ProcessState(rt, 100);
                            }
                            switch(getchar())
                            {
                                case 85:

```

```

        case 117:
            /* unsubscribe test */
            printf("processing Unsubscription .....\\n\\n");
            if (UPnP_ControlUnsubscribe (
                dh,
                serviceId,
                0,
                0
            ) >=0)
            {
                printf ("unsubscribed successfully ...\\n\\n");
            }
            else
            {
                printf("UPnP_Control Unsubscribe failed.\\n");
            }

            break;

        case 83:
        case 115:
            /* subscribe test */
            printf("processing subscription .....\\n\\n");
            if (UPnP_ControlSubscribe (
                dh,
                serviceId,
                500,
                0,
                0
            ) >=0)
            {
                printf ("Waiting for events...\\n\\n");
            }
            else
            {
                printf("UPnP_Control subscribe failed.\\n");
                done = 1;
            }
            break;

        case 88:
        case 120:
            done = 1;
            /* exit test */
            break;
    }

}

}
else
{
    printf("UPnP_ControlSubscribe failed.\\n");
}
}
else
{
    printf (" service not found!\\n");
}

UPnP_ControlServiceIteratorDone(&i);
}
else

```

```

    {
        printf (" search init failed!\n");
    }

    UPnP_ControlCloseDevice(dh);
}
else
{
    printf (" open failed.\n");
}

    rtp_strfree(deviceUrl);
}
else
{
    printf (" no device found.\n");
}
}

/*-----*/
void serviceDescribeTest (UPnPRuntime *rt, UPnPControlPoint *cp)
{
    UPnPControlDeviceHandle dh;
    char* deviceUrl = 0;

    printf("\nSearching for any device...\n");
    UPnP_ControlFindAllDevices(cp, 2500, &deviceUrl, 1);

    if (deviceUrl)
    {
        // In case you want to open a specific device, you can use the 2 lines below
        //printf("\nOpening device at %s...", "http://192.168.0.6:1085");
        //dh = UPnP_ControlOpenDevice(cp, "http://192.168.0.6:1085/light.xml");
        dh = UPnP_ControlOpenDevice(cp, deviceUrl);
        if (dh != 0)
        {
            UPnPControlServiceIterator i;

            printf("device open.\n\nLooking for services...");

            /* find the SwitchPower:1 service and invoke an action on it */
            if (UPnP_ControlGetServices(dh, &i) >= 0)
            {
                UPNP_CHAR* serviceId = UPnP_ControlNextService(&i);

                if (serviceId)
                {
                    IXML_Document *xmlDoc;
                    int done = 0;
                    UPnPControlDeviceInfo deviceInfo;

                    printf("\n  Found (serviceId: %.40s)\n\n", serviceId);

                    if (UPnP_ControlGetServiceOwnerDeviceInfo (dh, serviceId, &deviceInfo) >= 0)
                    {
                        printf (" Owner Device info for %.40s\n\n"
                                " deviceType:    %.60s\n"
                                " friendlyName:  %.60s\n"
                                " manufacturer:  %.60s\n"
                                " manufacturerURL: %.60s\n"
                                " modelDescription: %.60s\n"

```

```

        " modelName:      %.60s\n"
        " modelNumber:   %.60s\n"
        " modelURL:      %.60s\n"
        " serialNumber:  %.60s\n"
        " UDN:           %.60s\n"
        " UPC:           %.60s\n"
        " presentationURL: %.60s\n\n",
        servid,
        deviceInfo.deviceType,
        deviceInfo.friendlyName,
        deviceInfo.manufacturer,
        deviceInfo.manufacturerURL,
        deviceInfo.modelDescription,
        deviceInfo.modelName,
        deviceInfo.modelNumber,
        deviceInfo.modelURL,
        deviceInfo.serialNumber,
        deviceInfo.UDN,
        deviceInfo.UPC,
        deviceInfo.presentationURL);
    }

    printf("Getting Service Description...", servid);

    xmlDoc = UPnP_ControlGetServiceInfo(dh, servid);
    if (xmlDoc)
    {
        DOMString str = ixmlPrintDocument(xmlDoc);

        printf("done.\n\n");

        if (str)
        {
            printf("%s\n\n", str);
            ixmlFreeDOMString(str);
        }
        ixmlDocument_free(xmlDoc);
    }

    printf("Getting Service Description (Asynchronous call).", servid);
    UPnP_ControlGetServiceInfoAsync(dh, servid, &done);
    while (!done)
    {
        printf(".");
        UPnP_ProcessState(rt, 500);
    }
}
else
{
    printf (" service not found!\n");
}

UPnP_ControlServiceIteratorDone(&i);
}
else
{
    printf (" search init failed!\n");
}

UPnP_ControlCloseDevice(dh);
}

```

```
else
{
    printf (" open failed.\n");
}

rtp_strfree(deviceUrl);
}
}
```

## Appendix II

### Sample Control Point Callback

Here is an example of an control point application callback to handle asynchronous API events.

```
int controlPointCallback (
    UPnPControlPoint* cp,
    UPnPControlEvent* event,
    void *perControl,
    void *perRequest
)
{
    int result = 0;
    switch (event->type)
    {
        case UPNP_CONTROL_EVENT_DEVICE_FOUND:
        {
            char** str = (char**) perRequest;

            printf (" Device found at %.50s\n", event->data.discover.url);
            if(str) /* if callback data is passed */
            {
                *str = rtp_strdup(event->data.discover.url);
            }
            result = 1; // if 1 is returned no further searches will be performed
            break;
        }

        case UPNP_CONTROL_EVENT_SEARCH_COMPLETE:
        {
            printf (" Search Complete.\n\n");
            result = 1; // perform no further searches
            break;
        }

        case UPNP_CONTROL_EVENT_DEVICE_ALIVE:
        {
            printf ("\n\n Device alive at %.50s"
                "\n\n ST: %.55s"
                "\n\n USN: %.55s", event->data.discover.url, event->data.discover.type, event->data.discover.usn);
            break;
        }

        case UPNP_CONTROL_EVENT_DEVICE_BYE_BYE:
        {
            printf ("\n\n Device bye-bye "
                "\n\n ST: %.55s"
                "\n\n USN: %.55s", event->data.discover.type, event->data.discover.usn);
            break;
        }

        case UPNP_CONTROL_EVENT_ACTION_COMPLETE:
        {
            int* done = (int*) perRequest;
            if(event->data.action.success == 0)
            {
                printf("Action cannot be executed...\n");
                printf("Description: %s\n", event->data.action.errorDescription);
            }
            printf ("Action Complete.\n");
            if(done)
            {
                if(*done == -1)
            }
        }
    }
}
```

```

    {
        *done = 1;
    }
}
/* Response to action 'GetMinLevel' contains output value for
variable "MinLevel". */
outValue = UPnP_GetPropertyValueByName(event->data.action.response, "MinLevel");
if (outValue)
{
    *done = rtp_atoi(outValue);
}
break;
}

case UPNP_CONTROL_EVENT_SERVICE_GET_INFO_FAILED:
{
    int* done = (int*) perRequest;
    printf("UPnP_ControlGetServiceInfoAsync failed!");
    if(done)
    {
        *done = 1;
    }
    break;
}

case UPNP_CONTROL_EVENT_SERVICE_INFO_READ:
{
    int* done = (int*) perRequest;
    DOMString str = ixmlPrintDocument(event->data.service.scpdDoc);

    printf("service info read:\n\n");

    if (str)
    {
        printf("%s\n\n", str);
        ixmlFreeDOMString(str);
    }
    ixmlDocument_free(event->data.service.scpdDoc);
    if(done)
    {
        *done = 1;
    }
    break;
}

case UPNP_CONTROL_EVENT_SUBSCRIPTION_ACCEPTED:
    printf("subscription accepted\n\n");
    printf("subscription id is: %s\n", event->data.subscription.serviceld);
    break;

case UPNP_CONTROL_EVENT_SUBSCRIPTION_CANCELLED:
    printf("service Unsubscribed\n\n");
    break;

case UPNP_CONTROL_EVENT_SUBSCRIPTION_REJECTED:
    printf("subscription rejected\n\n");
    break;

case UPNP_CONTROL_EVENT_SERVICE_STATE_UPDATE:
{
    const UPNP_CHAR* str = UPnP_GetPropertyValuebyName(event->data.notify.stateVars,
"LoadLevelStatus");
    if (!str)
    {
        str = "[unknown]";
    }
    printf("Notification: Load level is =%s\n", str);
    break;
}
}

```

```

case UPNP_CONTROL_EVENT_SUBSCRIPTION_NEAR_EXPIRATION:
    printf ("Subscription Expiration Warning!\n");
    break;

case UPNP_CONTROL_EVENT_SUBSCRIPTION_EXPIRED:
    printf ("Subscription Expired.\n");
    break;

case UPNP_CONTROL_EVENT_DEVICE_OPEN:
{
    int* done = (int*) perRequest;

    if(event->data.device.handle)
    {
        UPnPControlDeviceInfo info;
        UPnPControlServiceIterator i;

        UPnP_ControlGetDeviceInfo(event->data.device.handle, &info);

        printf (" success!\n\n"
            " deviceType:   %.60s\n"
            " friendlyName:  %.60s\n"
            " manufacturer:   %.60s\n"
            " manufacturerURL: %.60s\n"
            " modelDescription: %.60s\n"
            " modelName:      %.60s\n"
            " modelNumber:    %.60s\n"
            " modelURL:       %.60s\n"
            " serialNumber:   %.60s\n"
            " UDN:           %.60s\n"
            " UPC:           %.60s\n"
            " presentationURL: %.60s\n\n",
            info.deviceType,
            info.friendlyName,
            info.manufacturer,
            info.manufacturerURL,
            info.modelDescription,
            info.modelName,
            info.modelNumber,
            info.modelURL,
            info.serialNumber,
            info.UDN,
            info.UPC,
            info.presentationURL);

        if (UPnP_ControlGetServices(event->data.device.handle, &i) >= 0)
        {
            UPNP_CHAR* servcId = UPnP_ControlNextService(&i);

            while (servcId)
            {
                printf ("Found Service: %.60s\n", servcId);
                printf (" (type: %.65s)\n", UPnP_ControlGetServiceType
                    (event->data.device.handle,
                    servcId)
                );
                servcId = UPnP_ControlNextService(&i);
            }

            UPnP_ControlServiceIteratorDone(&i);
        }

        UPnP_ControlCloseDevice(event->data.device.handle);
    }
    printf("\n");
    if(done)
    {
        *done = 1;
    }
}

```

```
    }  
    break;  
}  
case UPNP_CONTROL_EVENT_DEVICE_OPEN_FAILED:  
{  
    int* done = (int*) perRequest;  
    printf("Device Open Failed\n");  
    if(done)  
    {  
        *done = 1;  
    }  
    break;  
}  
}  
return (result);  
}
```