Intel® SDK for UPnP™ Devices

Programming Guide

Intel® SDK for UPnP™ Devices Version 1.2.1

November 2002

Notice

Disclaimer

INTEL CORPORATION MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. INTEL CORPORATION ASSUMES NO RESPONSIBILITY FOR ANY ERRORS THAT MAY APPEAR IN THIS DOCUMENT. INTEL CORPORATION MAKES NO COMMITMENT TO UPDATE OR TO KEEP CURRENT THE INFORMATION CONTAINED IN THIS DOCUMENT.

THIS SPECIFICATION IS COPYRIGHTED BY AND SHALL REMAIN THE PROPERTY OF INTEL CORPORATION. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED HEREIN.

INTEL DISCLAIMS ALL LIABILITY, INCLUDING LIABILITY FOR INFRINGEMENT OF ANY PROPRIETARY RIGHTS, RELATING TO IMPLEMENTATION OF INFORMATION IN THIS SPECIFICATION. INTEL DOES NOT WARRANT OR REPRESENT THAT SUCH IMPLEMENTATIONS WILL NOT INFRINGE SUCH RIGHTS.

NO PART OF THIS DOCUMENT MAY BE COPIED OR REPRODUCED IN ANY FORM OR BY ANY MEANS WITHOUT PRIOR WRITTEN CONSENT OF INTEL CORPORATION.

INTEL CORPORATION RETAINS THE RIGHT TO MAKE CHANGES TO THESE SPECIFICATIONS AT ANY TIME, WITHOUT NOTICE.

Legal Notices

Intel UPnP software products are copyrighted by, and shall remain the property of, Intel Corporation. Use, duplication or disclosure is subject to restrictions stated in Intel's Software License Agreement, or in the case of software delivered to the government, in accordance with the software license agreement as defined in FAR 52.227-7013.

Intel Corporation, 5200 N.E. Elam Young Parkway, Hillsboro, OR 97124-6497

[§] Other brands and names are the property of their respective owners.

Contents

1 Overview

UPnP allows automatic discovery and control of services available on the network from other devices without user intervention. Devices that act as servers can advertise their services to clients. Client systems, known as control points, can search for specific services on the network. When they find the devices with the desired services, the control points can retrieve detailed descriptions of the devices and services and interact from that point on.

This document provides an overview of UPnP and provides examples of how to write a UPnP device and control point. For a complete description of the Intel® SDK for UPnP Devices API functions, refer to the *Intel® SDK for UPnP™ Devices v1.2 API Reference* included with the SDK.

The SDK also includes sample control point and device applications. For details on building and running the samples, see the README file in the sample directory of the SDK distribution.

1.1 UPnP Overview

This section provides a brief description of UPnP. For more information, refer to the document *Universal Plug and Play Device Architecture*, available from the UPnP Forum at http://www.upnp.org/resources/documents.asp.

UPnP includes five basic phases:

1. *Discovery*. In this first phase, control points search for devices and services. Similarly, devices multicast announcements of services they offer.
2. *Description*. Once a control point finds an interesting device or service, it requests from the device a complete description of the device, its component devices, and services.
3. *Control*. This phase allows control points to control one or more of the services contained in a device by enacting changes in the state of the device.
4. *Eventing*. This phase allows control points to keep in sync with the state of services in which it is interested. Control points subscribe to the event server for a particular service and receive event notifications when that service's state changes.
5. *Presentation*. The presentation phase allows a device to host a document, written in standard HTML, which can be a user interface for that device.

The following sections describe each of these phases.

1.1.1 Discovery

In the discovery phase control points find devices and services, and devices announce their presence to control points using the Simple Service Discovery Protocol (SSDP). SSDP uses a variant of HTTP that operates over multicast UDP for broadcasts and another variant of HTTP that operates over unicast UDP for replies.

A device may consist of other devices, each with its own services. Devices are identified both by type and by a unique identifier. Services are identified by their type.

To search for devices or services on the network, control points use the HTTP `M-SEARCH` command multicast to the address 239.255.255.250:1900 over UDP. Any device on the network that matches the criteria the control point is searching for issues a unicast UDP reply that includes the URL to its description document (see section 1.1.2). If a control point receives one or more acceptable replies, it moves into the description phase.

When a control point sends out a search request, it includes the amount of time it is willing to wait in an SSDP header. Matching devices will wait a random time between zero and the number of seconds the control point indicated before responding. If the control point does not receive any replies when its search time has expired, it can assume that there are currently no matching devices on the network.

Devices don't have to wait for a control point to search for their services. They can advertise their device availability by means of the SSDP `NOTIFY` command on the 239.255.255.250:1900 multicast address. When control points see this `NOTIFY` multicast, they can request the device's description document using a standard HTTP `GET` request to the URL in the `NOTIFY` message. Devices must send a notification when their services will no longer be available.

1.1.2 Description

When a control point locates a service it wants to know more about, it requests the description document. The description is an XML document describing the device, including:

- Manufacturer information, version, and so on.
- Any URLs to icons that can be used for the device.
- A list of embedded devices.
- A list of services supported by the device.

For more information on the format of the description document, refer to the document *Universal Plug and Play Device Architecture.*

The control point requests the description document using HTTP over TCP. The control point performs a standard HTTP `GET` command (similar to retrieving a Web page). On the server side, the device runs a standard HTTP server—either a full Web server such as Apache or a mini-server. Many of the elements in the description document are URLs. Those elements are also retrieved using HTTP/TCP.

1.1.3 Control

Once a control point discovers a device and retrieves its description document, it may want to control one or more of the services contained in the device. The Simple Object Access Protocol (SOAP) allows a control point to query or change elements in a service's state table. SOAP uses the `POST` or `M-POST` HTTP command transported over TCP.

SOAP uses XML to specify what actions to take. The control point creates the XML document and posts it to the control URL for the service, as specified in the description document. The control point can request current values and make changes to the service's state table.

On the server side, the control server waits for control requests. The control server is an HTTP-like server implementing the SOAP protocol. A device can operate more than one control server depending on the combination of services provided by the device.

1.1.4 Eventing

After a control point discovers a device and retrieves its description, it can stay informed of the state of a service offered by that device. Interested control points subscribe to the device's event notification service URL found in the description document for the particular service. An event notification is sent to the control point any time the state of the service changes, even if the control point causes the change.

Subscribe and unsubscribe requests use HTTP/TCP to connect to the event URL contained in the description document for the service. The control point specifies an URL where event notifications are made during subscription. Events arrive by means of HTTP/TCP to the URL registered with the service. The event notification includes a small XML document that describes the actual event, such as a change in the state table for the service.

On the server side, an event server waits for subscribe and unsubscribe requests. The event server is an HTTP-like server implementing the General Event Notification Architecture protocol (GENA). A device may have to operate more than one event server depending on the combination of services provided by the device.

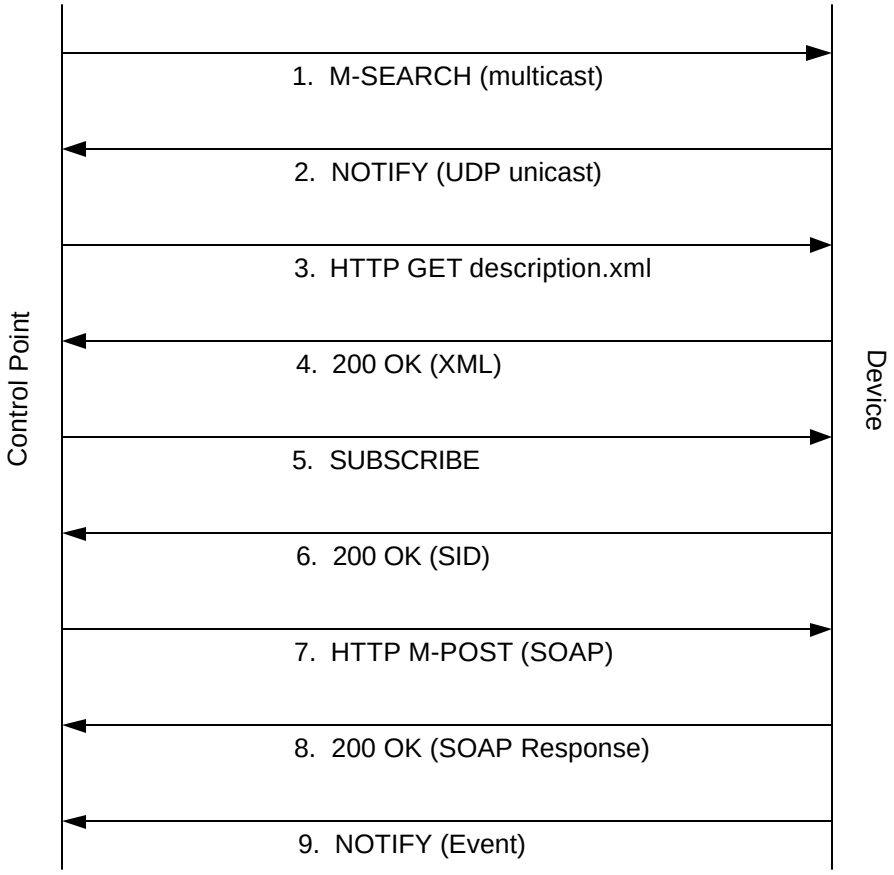
1.1.5 Presentation

For devices that need or support user interaction, in the presentation phase a control point can download an HTML document that represents the user interface for the device. This is a standard HTML document that can provide a means of control or status display.

The protocol for retrieving the presentation document, as with the description document, is HTTP over TCP. The control point can use the presentation URL contained in the description document to request the presentation document. Not all devices have a presentation document nor are all control points able to display a presentation document containing complex HTML objects such as frames, embedded Java[§] applets, and so on.

1.1.6 Control Point and Device Interaction

The following diagram shows the sequence of interactions between a control point and device during the UPnP phases described in the previous sections. A description of each step follows the diagram. The descriptions indicate the API calls that a control point or device uses and what network packets the SDK issues. It should be noted that due to the asynchronous nature of the sequence, the interactions do not necessarily happen in the order shown. The control and eventing steps can happen in any order.



1. The control point sends out a search request using the `UpnpSearchAsync()` API. The SDK for UPnP Devices issues a multicast `M-SEARCH` SSDP message onto the network.
2. If the device matches what the control point is searching for, the SDK issues a unicast UDP `NOTIFY` response with the URL to the device's description document. The SDK responds automatically from information contained in the device description document registered using `UpnpRegisterRootDevice()` or `UpnpRegisterRootDevice2()`.
3. If the control point wants more information on the device, it calls `UpnpDownloadXmlDoc()` with the URL with which the device responded. `UpnpDownloadXmlDoc()` downloads the XML description document using a standard HTTP `GET` request and returns a DOM document representing the device description. This step may be repeated to retrieve the service description documents for the device also.
4. The web server contained in the device responds to the request and returns the XML description document.
5. To receive automatic notifications of changes in the device, a control point subscribes to the services in which it is interested. Control points subscribe via `UpnpSubscribe()` or `UpnpSubscribeAsync()`. The control point extracts the subscription URL out of the device description document for the service or services to which it would like to subscribe, and calls one of the subscribe functions. For each subscribe call, the SDK sends a `SUBSCRIBE` message via HTTP along with a URL to which to send the events.
6. The device acknowledges the subscription request and returns a unique Subscription Identifier (SID).
7. The control point instructs the device to perform some action by changing one of the state variables contained inside the device. The URL to send control requests is contained in the device description document. The control point calls `UpnpSendAction()` or `UpnpSendActionAsync()` to change the state. The SDK issues a SOAP action via an HTTP `M-POST` command.
8. The device changes the state of the internal variable and issues a SOAP response message.
9. The device can notify clients of changes in its state because of either explicit actions, as in step 8, or implicit changes in the device itself. The device calls `UpnpNotify()` or `UpnpNotifyExt()` to send the updates. The SDK automatically notifies all subscribed control points via a unicast `NOTIFY` message over HTTP.

1.2 SDK Architecture

The following diagram shows the architecture of the SDK:

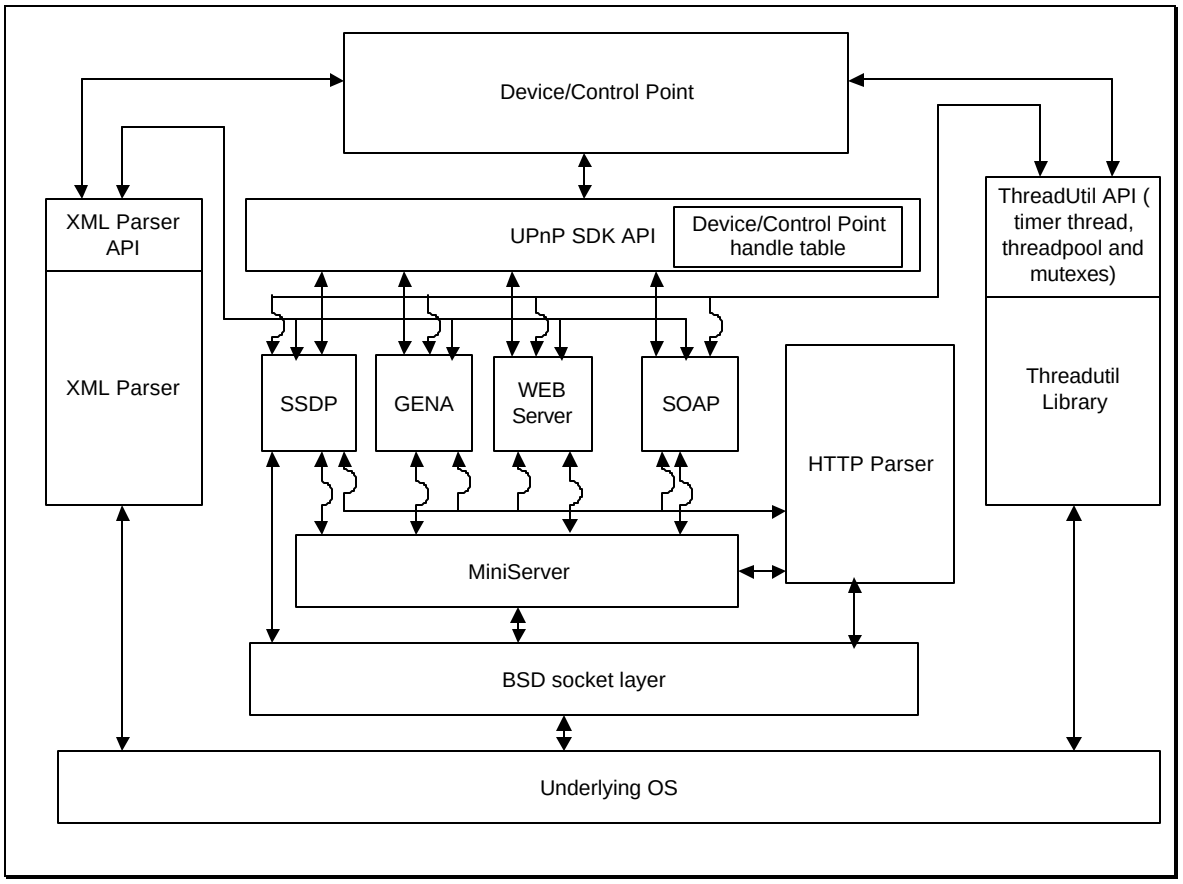


Figure 1: A high-level diagram of the SDK architecture

The following sections describe each part of the diagram. For more information on any of the protocols, refer to the document *Universal Plug and Play Device Architecture*.

1.2.1 Device/Control Point Application

The customer provides the client or server application software to run on top of the SDK. The client or server application implements the functionality of a specific service. For example, for a gateway service, the server software implements the "Enable Internet" functionality that the control point software can control using UPnP. A sample service and control point application is included as part of the SDK.

1.2.2 SDK API

The SDK API abstracts the details of the core UPnP protocols away from the control point or service application and gives applications access to the functionality in a unified interface. This frees the developers from concern about the details of the SSDP, GENA, and SOAP protocols in their code.

The API layer also maintains the table of control point and device handles registered with the SDK. On each call to an API function, the SDK will validate the handle is known by checking in the handle table. Currently, there is a limit of one control point and one device handle that can be allocated in one process at a time. In other words, an application can register once as a device and once as a control point. Any further attempts to register by that application will fail.

For information about the API, refer to the *Intel® SDK for UPnP™ Devices v1.2 API Reference*.

1.2.3 SSDP

The SSDP module implements the Simple Service Discovery Protocol, providing the discovery phase of UPnP. This module allows control points to send multicast searches for services and devices on the network and receives the replies to those searches. It also notifies them when new services are announced on the network.

This module also allows devices to multicast announcements of their services to the network.

1.2.4 Mini Web Server

The Mini Web Server module handles the standard HTTP `GET` requests. Many UPnP elements are requested using this basic HTTP service. This module manages the locations of documents that are available using the `GET` command and implements the actual streaming of the data using the HTTP protocol.

The Mini Web Server module implements the `RANGE` header from HTTP/1.1. This header allows a remote client to request a certain piece (or pieces) of a file rather than the entire file. An application for this is seeking to a particular track within a play list or jumping to an offset inside a media file.

The Mini Web Server also supports HTTP `POST` requests for Virtual Directories but not for any others. For a definition of Virtual Directories, refer to section 1.3.

1.2.5 GENA

The GENA module implements the General Event Notification Architecture, providing the eventing phase of UPnP. Control points use this module to subscribe or unsubscribe to services of interest. Service applications receive subscribe and unsubscribe notifications from this module and generate the appropriate events.

1.2.6 SOAP

The SOAP module implements the Simple Object Access Protocol, providing the control phase of UPnP. Control points use this module to generate the appropriate XML documents to retrieve or change the state tables of a service. The server uses this module to decode the control requests and generate the correct responses.

1.2.7 HTTP

The HTTP module parses the HTTP headers for incoming messages and aids in constructing the appropriate headers for outgoing messages. It understands a number of HTTP/1.0 and HTTP/1.1 headers. It also provides the parsing for HTTP/1.1 chunked encoding. The SDK does not generate chunked encoding by default on `GET` requests with HTTP/1.1. It does, however, fully support decoding it when it receives chunked encoded messages.

1.2.8 Mini Server

The Mini Server layer provides common functionality for GENA, SOAP, SSDP, and the mini-Web server. This layer accepts all network connections, determines which request is coming in, hands off the HTTP header to the HTTP module for parsing, and transfers the connection over to the appropriate protocol for processing. The first line of the HTTP header contains the request. The Mini Server layer minimally handles these commands:

- **GET**. Control points use the `GET` command to retrieve the description document and any of its sub-elements including the presentation document, the service description documents, and the icons

associated with the device. Using Virtual Directories, described in section 1.3, device applications can request the SDK to generate callbacks into the application to handle `GET` requests for certain directories. Refer to section 1.3 on page 8 for more information.

- **POST/M-POST**. A SOAP command is in the form of a `POST` or an `M-POST` command. All POST commands posted to one of the control URLs specified in the device description document are transferred to the SOAP module for further processing. `POST` commands are also allowed to Virtual Directories, generating a callback into the device application. Refer to section 1.3 on page 8 for more information.
- **SUBSCRIBE**. `SUBSCRIBE` commands are transferred to the GENA module for further processing. Control points use `SUBSCRIBE` requests to subscribe or renew a subscription to event notifications for a particular service.
- **UNSUBSCRIBE**. `UNSUBSCRIBE` commands are transferred to the GENA module for further processing. Control points use `UNSUBSCRIBE` commands to notify a server that they are no longer interested in receiving event notifications.
- **NOTIFY**. `NOTIFY` commands are transferred to other modules for further processing. In the case of TCP connections, they are transferred to the GENA module. In the case of UDP connections, they are transferred to the SSDP module. `NOTIFY` commands can be event notifications sent from servers to control points containing a description of the event, or notification of a device or service appearing on, or disappearing from, the network.

1.2.9 ThreadUtil Library

The Intel SDK for UPnP Devices utilizes threads extensively to parallelize processing of UPnP traffic as much as possible. The ThreadUtil library provides the SDK an abstraction of a POSIX-like thread API, thread management routes, and linked and free list management utilities. The thread manager creates a pool of threads that can be "borrowed" to perform a task and returned back into the pool to be used for other purposes. It maintains a ratio of how many jobs it has queued to the current number of threads in the pool. If this ratio gets too large, it will increase the number of threads in the pool automatically. Likewise, if threads sit in the pool idle for a long time, it will remove threads from the pool to decrease resource usage but always maintaining a configurable minimum number.

1.2.10 XML Parser

XML is used extensively in UPnP. The description documents are XML documents. GENA uses XML to describe change in a service's state. SOAP uses XML to format requests and responses. The SDK contains an XML parser used both by the core UPnP protocols and by the client or server software.

The interface to the XML Parser uses a subset of the Document Object Model (DOM) Level 2 recommendation written by the World Wide Web Consortium (W3C). The SDK provides a C interface. It implements the `Node`, `Attr`, `CDATASection`, `Document`, `Element`, `NamedNodeMap`, and `NodeList` interfaces, but excludes the `DOMImplementation`, `DocumentFragment`, `CharacterData`, `Text`, and `Comment` interfaces. See http://w3c.org/DOM/ for more information on DOM.

1.2.11 BSD Socket Layer

The SDK assumes the BSD Socket Layer (POSIX.1g) is provided by the operating system. Although not part of the SDK, it is included in Figure 1 to illustrate the relationship between the SDK and the underlying operating system.

1.3 Virtual Directories

The integrated Mini Web Server inside the SDK supports a concept known as Virtual Directories. A Virtual Directory is a path accessible to HTTP clients that does not match the physical structure of the Mini Web Server's root directory structure. Normally, the URLs sent to a web server correspond to the actual physical structure of the files hosted by the web server. The web server prefixes the URL passed from the client with its root directory and opens that file on the file system, feeding the data back to the client. With a Virtual Directory, a device application can register specific directories that it wishes to receive callbacks from when a makes a request. For example, assume a device has a directory structures such as this:

```
<webroot>
      index.html
      device.xml
      service.xml
```

To retrieve `index.html`, a client would make a standard HTTP `GET` request:

```
GET /index.html HTTP/1.0
```

Suppose the device application has registered a Virtual Directory called "media". A client would make a request such as this:

```
GET /media/MySong.mp3 HTTP/1.0
```

The device application gets a callback from the Mini Web Server requesting the device application to provide the data to return to the client. Where the data comes from does not matter to the Mini Web Server. It could be streamed from the Internet, read from a file that is outside the Mini Web Server root directory, or could be generated dynamically.

The API that a device application uses to receive these callbacks is very similar to a standard file interface consisting of a structure with six function pointers:

- `get_info()` is the first callback for a request. It passes a structure to the application with information about the URL a client is requesting. The application passes back information about the file, such as the size, to the Mini Web Server. This information becomes the basis of the HTTP response header.
- `open()` returns a handle back to the Mini Web Server for subsequent operations. What open actually does and what value the handle has is irrelevant to the Mini Web Server. It will simply pass this handle to any subsequent calls.
- `read()` retrieves a block of data. The Mini Web Server calls this function repeatedly on an HTTP `GET` requests until it returns no more data.
- `write()` writes a block of data. The Mini Web Server calls this function repeatedly on an HTTP `POST` request to a Virtual Directory.
- `seek()` changes the position in a file. The Mini Web Server mainly uses this function to satisfy HTTP `RANGE` requests which ask for particular offsets into a file.
- `close()` closes the handle.

A device application registers these callback functions via `UpnpSetVirtualDirCallbacks()`. `UpnpAddVirtualDir()` adds a new mapping to the list of Virtual Directories. Note that the directory passed to `UpnpAddVirtualDir()` becomes the prefix that the Mini Web Server uses to determine if it should generate a callback. `UpnpRemoveVirtualDir()` removes a single Virtual Directory mapping and

`UpnpRemoveAllVirtualDirs()` removes all mappings. For more detail on each of these functions, consult the *Intel® SDK for UPnP™ Devices v1.2 API Reference.*

2 Writing a UPnP Device

There are many ways to implement a UPnP device using the SDK for UPnP Devices. However, any implementation must perform some basic steps. Specifically, an application must:

1. Set up and initialize the device by following these basic steps:
    a. Initialize the SDK using `UpnpInit()`.
    b. Set a root directory for the Mini Web Server using `UpnpSetWebServerRootDir()`.
    c. Register the device description document using `UpnpRegisterRootDevice()` or `UpnpRegisterRootDevice2()`.
    d. Perform any device-specific initialization.
    e. Advertise the device on the network using `UpnpSendAdvertisement()`.
2. Handle asynchronous requests. The device needs to handle three different types of requests:
    a. Requests to subscribe to notifications of service state changes.
    b. Requests to retrieve the current value of a service state variable.
    c. Requests to change the value of a service state variable.
3. Keep control points up-to-date by sending events using `UpnpNotify()` or `UpnpNotifyExt()`.
4. Shut down the device following these steps:
    a. Send out SSDP "bye-bye" messages and unregister the device from the SDK using `UpnpUnRegisterRootDevice()`.
    b. Shut down the SDK using `UpnpFinish()`.

In the following discussion, the examples are drawn from the sample TV device implementation found in `upnp/sample/tvdevice/`. Parts of the sample code have been removed in this document to aid in clarity. For complete descriptions of the SDK for UPnP Devices API functions, refer to the *Intel® SDK for UPnP™ Devices v1.2 API Reference* included with the SDK.

2.1 Setup and Initialization

2.1.1 Initializing the SDK

Before starting a device implementation, it is important to write or obtain the device and service description documents that are going to be used. These specify the type and number of services that the device supports, as well as actions, parameters, and variables that each service supports. For more information refer to the *Universal Plug and Play Device Architecture* document or the relevant specification for the particular device.

The device and service description documents the sample TV device uses are in `upnp/sample/tvdevice/web`. It is important to note that the sample TV device is *not* a certified UPnP device.

The application must initialize the SDK before using any of the API functions.

```
if ((ret = UpnpInit( ip_address, port )) != UPNP_E_SUCCESS) {
      SampleUtil_Print( "Error with UpnpInit -- %d\n", ret );
      UpnpFinish();
      return ret;
}
```

The application can specify an IP address and port number during initialization. These are used to set the interface and port the server listens on for UPnP and HTTP requests. If the IP address is `NULL`, the address of the first non-null, non-loopback address is used. If the port number is 0, a random port number is used.[1] You can retrieve both the IP address and port number from the SDK after initialization by using `UpnpGetServerIpAddress()` and `UpnpGetServerPort()`.

2.1.2 Setting a Root Directory

Once the SDK has been initialized, the device application can specify the root directory of the web server. This is the local directory the web server searches in order to serve files in response to HTTP requests. Specifying the root directory is optional. If a root directory is not specified, then only requests for documents in the virtual directory list are served via the registered callbacks. It is up to the application to insure that the specified directory contains the necessary files (such as the description documents for the device and services). The web directory for the sample is `/upnp/sample/tvdevice/web`.

```
char web_dir_path[] = "./web";

if ((ret = UpnpSetWebServerRootDir( web_dir_path )) != UPNP_E_SUCCESS) {
      SampleUtil_Print( "Error specifying webserver root directory -- %s:
                        %d\n", web_dir_path, ret);
      UpnpFinish();
      return ret;
}
```

Note that once `UpnpInit()` has succeeded, it is very important to call `UpnpFinish()` if the device shuts down due to an error. This gives the SDK an opportunity to clean up the resources it has allocated.

2.1.3 Registering a Root Device

The next step in setting up the device is registering with the SDK. There are two functions for registering the device, `UpnpRegisterRootDevice()` and `UpnpRegisterRootDevice2()`. The first function takes a fully qualified description URL as input, and the second can take the description document in a variety of forms. The example shows only the first case.

```
if ((ret = UpnpRegisterRootDevice( desc_doc_url,
                                   TvDeviceCallbackEventHandler,
                                   &Cookie,
                                   &device_handle ))
            != UPNP_E_SUCCESS)
{
      SampleUtil_Print( "Error registering the rootdevice : %d\n", ret );
      UpnpFinish();
      return ret;
}
```

The first parameter to the function is the description document URL. It must point to the valid description document of the device. If the description document is a file being served by the web server included with the

[1] The current implementation listens to all interfaces. The IP address specified during initialization will only affect the IP address advertised during SSDP advertisements and search responses.

SDK, it should be located off of the directory specified in `UpnpSetWebServerRootDir()`. The second parameter registers a callback with the SDK. The prototype for the callback is:

```
int CallbackFxn( Upnp_EventType EventType, void* Event, void* Cookie );
```

Whenever a request is received for the device from the network, such as a subscription request, a get variable request, or an action request, this function is called on an independent thread with the appropriate parameters. `EventType` specifies the type of request received, the `Event` parameter is a structure whose actual type differs based on the `EventType`, and `Cookie` is the application-specific data specified in `UpnpRegisterRootDevice()`. For the TV sample, the callback registered is:

```
int TvDeviceCallbackEventHandler( Upnp_EventType EventType,
                                  void *Event,
                                  void *Cookie )
```

The third parameter to `UpnpRegisterRootDevice()` is a void pointer specified by the user. It can be anything the size of a pointer and can be used to point to application-specific data. It can also be `NULL`. The application is responsible for allocating and deallocating this pointer if it is used. The pointer is passed back to the application when it receives a request. The final parameter to `UpnpRegisterRootDevice()` is a pointer to space allocated by the application to store the device handle. The device handle is used as a parameter to other API functions.

2.1.4 Device-Specific Initialization

So far we have not discussed initialization specific to the device or application. Before the device is advertised, all device and application-specific initialization should be performed. Once the device has been advertised, it can start receiving requests immediately. The device is responsible for maintaining the values of the service state variables as well as manipulating them correctly in response to actions. In the TV device sample code, the function, `TvDeviceStateTableInit()` initializes the internal data structures used to store the state variables, service ids, action pointers, and so on.

2.1.5 Advertising the Device

The final step in setting up the UPnP device is sending out advertisements.

```
int default_advr_expire = 100;

if ((ret = UpnpSendAdvertisement( device_handle, default_advr_expire ))
            != UPNP_E_SUCCESS) {
      SampleUtil_Print( "Error sending advertisements : %d\n", ret );
      UpnpFinish();
      return ret;
}
```

The first parameter is the device handle returned during registration. The second parameter is the expiration time for the advertisement. During the lifetime of the device, the SDK automatically re-advertises the device before it expires.

The device is now ready to receive requests. The application must set itself in a loop or wait for some condition in order to shut down.

2.2 Handling Requests

During the lifetime of a device, its main purpose is to handle requests sent to it by control points. When requests are received by the SDK, they are forwarded to the application through the callback specified when the device registered. The callback is invoked on an independent thread with the appropriate parameters, depending on the type of the request.

There are three requests handled by a device:

- subscription requests
- get variable requests
- action requests

Note that the device does not need to handle any discovery requests. Since the SDK has the device description document for the device (passed during `UpnpRegisterRootDevice()`), it can automatically determine if the discovery search criteria matches the device. If so, it responds with the URL of the device description document.

The type of request is indicated in the callback through the `EventType` variable. The callback for the TV device sample is:

```
int TvDeviceCallbackEventHandler( Upnp_EventType EventType,
                                  void *Event,
                                  void *Cookie )
{
    switch ( EventType ) {

    case UPNP_EVENT_SUBSCRIPTION_REQUEST:

      TvDeviceHandleSubscriptionRequest(
          (struct Upnp_Subscription_Request *) Event);
      break;

    case UPNP_CONTROL_GET_VAR_REQUEST:
      TvDeviceHandleGetVarRequest(
          (struct Upnp_State_Var_Request *) Event);
      break;

    case UPNP_CONTROL_ACTION_REQUEST:
      TvDeviceHandleActionRequest(
          (struct Upnp_Action_Request *) Event);
      break;

    default:
          SampleUtil_Print( "Error in TvDeviceCallbackEventHandler: unknown
                            event type %d\n", EventType );
    }

    return 0;
}
```

2.2.1 Subscription Requests

When a control point makes a subscription request, the SDK invokes the registered callback with the `EventType` variable set to `UPNP_EVENT_SUBSCRIPTION_REQUEST`. The device is responsible for accepting the subscription by calling either `UpnpAcceptSubscription()` or `UpnpAcceptSubscriptionExt()`, thus sending the current state table to the control point. The difference between these two functions is simply how the application passes the state table to the SDK for sending to the subscribing control point. `UpnpAcceptSubscription()` takes a couple of arrays of strings for the variable/value pairs. `UpnpAcceptSubscriptionExt()` takes a DOM document for the current values of the variables. The format of the DOM document is given in section 4.3 of the *Universal Plug and Play Device Architecture* document. The TV sample uses `UpnpAcceptSubscription()`:

```
int TvDeviceHandleSubscriptionRequest( IN struct Upnp_Subscription_Request *
                                       sr_event )
{
      unsigned int I = 0;

      ithread_mutex_lock( &TVDevMutex );

      for ( i=0; I < TV_SERVICE_SERVCOUNT; i++ )
      {
            if ((strcmp(sr_event->UDN,tv_service_table[i].UDN) == 0) &&
                (strcmp(sr_event->ServiceId,tv_service_table[i].ServiceId) == 0))
            {

                  UpnpAcceptSubscription( device_handle,
                                          sr_event->UDN,
                                          sr_event->ServiceId,
                                          (const char **)
                                                tv_service_table[i].VariableName,
                                          (const char **)
                                                tv_service_table[i].VariableStrVal,
                                          tv_service_table[i].VariableCount,
                                          sr_event->Sid );

            }
      }
      ithread_mutex_unlock( &TVDevMutex );
      return 1;
}
```

In this case, the `Event` parameter passed to the callback is a pointer to a structure of type `struct Upnp_Subscription_Request`. The subscription request structure specifies the UDN and ServiceID of the service for which the subscription is requested, along with the subscription identifier (SID). In the above sample, once the service is identified, the state variables, which are stored in `tv_service_table[i]`, are sent to the control point using `UpnpAcceptSubscription()`. `UpnpAcceptSubscription()` takes the following:

- the device handle
- the UDN
- the ServiceID
- the variable names `(tv_service_table[i].VariableName)`
- the variable values `(tv_service_table[i].VariableStrVal)`
- the number of variables `(tv_service_table[i].VariableCount)`
- the SID

Once the subscription has been accepted, the control point receives future events sent by the device. The sample code uses the `TVDevMutex` mutual exclusion variable to protect access to global data handled by the thread. The SDK callbacks are multi-threaded, and it is up to the device implementation to insure that access to shared data is protected.

2.2.2 Get Variable Requests

When a control point requests the current state of a variable, the SDK invokes the registered callback with the `EventType` set to `UPNP_CONTROL_GET_VAR_REQUEST`.

Note that the UPnP Forum has deprecated this capability. For clients to access state variables, the preferable method is to have a specific action to retrieve the value. Having this support in a device is not strictly necessary. The discussion below illustrates how an application supports this on top of the Intel SDK for UPnP Devices.

The `Event` parameter passed to the callback is a pointer to a structure of type `struct Upnp_State_Var_Request`. The request structure specifies the UDN, ServiceID, and variable name of the requested variable. The device is responsible for setting the current value of the variable in the structure.

```
int TvDeviceHandleGetVarRequest( INOUT struct Upnp_State_Var_Request *
                                            cgv_event)
{
      unsigned int I = 0, j = 0;
      int getvar_succeeded = 0;

      cgv_event->CurrentVal = NULL;

      ithread_mutex_lock( &TVDevMutex );

      for ( i = 0; I < TV_SERVICE_SERVCOUNT; i++ )
      {
            //check udn and service id
            if ((strcmp( cgv_event->DevUDN, tv_service_table[i].UDN ) == 0 ) &&
                (strcmp( cgv_event->ServiceID, tv_service_table[i].ServiceId )
                   ==0 ))
            {
                  //check variable name
                  for ( j = 0; j < tv_service_table[i].VariableCount; j++ )
                  {
                        if (strcmp( cgv_event->StateVarName,
                                    tv_service_table[i].VariableName[j] ) == 0 )
                        {
                              getvar_succeeded = 1;
                              cgv_event->CurrentVal =
                                    ixmlCloneDOMString( tv_service_table[i].
                                                        VariableStrVal[j] );
                              break;
                        }
                  }
            }
      }

      if ( getvar_succeeded ) {
            cgv_event->ErrCode = UPNP_E_SUCCESS;
      } else {
            cgv_event->ErrCode = 404;
            strcpy( cgv_event->ErrStr, "Invalid Variable" );
      }

      ithread_mutex_unlock( &TVDevMutex );
      return( cgv_event->ErrCode == UPNP_E_SUCCESS );
}
```

The `CurrentVal` member of the structure must be set with a character string created with `ixmlCloneDOMString()`. This memory is released by the SDK once the value has been sent to the control point. Once again, the `TVDevMutex` mutual exclusion variable is used to protect data accessed by multiple threads.

2.2.3 Action Requests

When a control point sends an action to the device, the SDK invokes the registered callback with the `EventType` set to `UPNP_CONTROL_ACTION_REQUEST`.

The `Event` parameter passed to the callback is a pointer to a structure of type `struct Upnp_Action_Request`. The request structure specifies the UDN, ServiceID, Action Name, and the action request document (which contains the inbound parameters). The device is responsible for navigating the document, extracting the relevant parameters, performing the requested action, creating a response document with outbound parameters (if applicable), and sending any events (if applicable). In the sample TV device, the action request is handled by the function `TvDeviceHandleActionRequest()`. This function in turn dispatches the request by looking up the action name in a table and retrieving the appropriate function. Each UPnP action is handled by a separate function. The prototype for the function is:

```
int upnp_action( IN Document *in, OUT Document **out,
                 OUT char **errorString ).
```

The function is passed the XML document specifying the parameters for the action and is required to create the response document, as well as send back an error string if appropriate. The function is required to return `UPNP_E_SUCCESS` on success and a non-zero error code otherwise.

The TV device first sees which service the control point is invoking an action on:

```
if ((strcmp( ca_event->DevUDN,
             tv_service_table[TV_SERVICE_CONTROL].UDN) == 0 ) &&
    (strcmp( ca_event->ServiceID,tv_service_table[TV_SERVICE_CONTROL].
             ServiceId ) == 0 ))
{
      service = TV_SERVICE_CONTROL;
}
else if ((strcmp( ca_event->DevUDN,
                  tv_service_table[TV_SERVICE_PICTURE].UDN) == 0 ) &&
         (strcmp( ca_event->ServiceID,tv_service_table[TV_SERVICE_PICTURE].
                  ServiceId ) == 0 ))
{
      service = TV_SERVICE_PICTURE;
}
```

Based on that service, it looks up the specific action and dispatches that to the action handler:

```
for ( i = 0; ( (i < TV_MAXACTIONS) &&
             ( tv_service_table[service].ActionNames[i] != NULL ));
      i++)
{
      if (!strcmp( ca_event->ActionName,
                   tv_service_table[service].ActionNames[i] ))
      {
            retCode = tv_service_table[service].actions[i]
                  ( ca_event->ActionRequest,
                    &ca_event->ActionResult,
                    &errorString);
            action_found = 1;
            break;
      }
}
```

As an example, here is the function implementing the `SetChannel` action:

```
int TvDeviceSetChannel( IN Document *in, OUT Document **out,
                        OUT char **errorString )
{
      char *value = NULL;
      int  channel = 0;

      (*out) = NULL;
      (*errorString) = NULL;

      if (!(value = SampleUtil_GetFirstDocumentItem( in, "Channel" ))) {
            (*errorString) = "Invalid Channel";
            return UPNP_E_INVALID_PARAM;
      }

      channel = atoi(value);

      if (channel < MIN_CHANNEL  ||  channel > MAX_CHANNEL) {
            free( value );
            (*errorString) = "Invalid Channel";
            return UPNP_E_INVALID_PARAM;
      }

      /* Vendor-specific code to set the channel goes here */

      if (TvDeviceSetServiceTableVar( TV_SERVICE_CONTROL,
                                      TV_CONTROL_CHANNEL,
                                      value ))
      {
            if (UpnpAddToActionResponse( out,"SetChannel",
                                         TvServiceType[TV_SERVICE_CONTROL],
                                         "NewChannel",value )
                  ! = UPNP_E_SUCCESS)
            {
                  (*out)=NULL;
                  (*errorString) = "Internal Error";
                  free( value );
                  return UPNP_E_INTERNAL_ERROR;
            }
            free( value );
            return UPNP_E_SUCCESS;
      } else {
            free( value );
            (*errorString) = "Internal Error";
            return UPNP_E_INTERNAL_ERROR;
      }
}
```

The function uses the `UpnpAddToActionResponse()` utility function to build the action response.

2.3 Sending Events

Whenever an evented state variable is changed, the device is required to send a state table update event. This can be in response to an action, an outside event, user input, and so on. The device application is responsible for determining when an event should be sent. The SDK sends the event to all subscribed control points. In the TV device sample, events are sent in response to actions. Eventing is handled by the function `TvDeviceSetServiceTableVar()`.

```
int TvDeviceSetServiceTableVar( IN unsigned int service,
                                IN unsigned int variable,
                                IN char *value)
{
      ithread_mutex_lock( &TVDevMutex );

      strcpy( tv_service_table[service].VariableStrVal[variable], value );

      UpnpNotify(device_handle,
            tv_service_table[service].UDN,
            tv_service_table[service].ServiceId,
            (const char **)&tv_service_table[service].VariableName[variable],
            (const char **)&tv_service_table[service].VariableStrVal[variable],
            1);

      ithread_mutex_unlock( &TVDevMutex );

      return( 1 );

}
```

The function updates the application's internal state table and sends the event using `UpnpNotify()`. `UpnpNotify()` takes the following:

- the device_handle
- the device UDN
- the ServiceID
- the names of changed variables (`&tv_service_table[service].VariableName[variable]`)
- the values of changed variables (`&tv_service_table[service].VariableStrVal[variable]`)
- the number of changed variables (`1`)

2.4 Shutting Down

When a device is shut down, the SDK must be uninitialized. This is done by calling `UpnpUnRegisterRootDevice()` and `UpnpFinish()`. The TV sample performs all this in `TvDeviceStop()`.

```
int TvDeviceStop()
{
      UpnpUnRegisterRootDevice( device_handle ;
      UpnpFinish();
      SampleUtil_Finish();
      ithread_mutex_destroy( &TVDevMutex );
      return UPNP_E_SUCCESS;
}
```

3 Writing a UPnP Control Point

The Intel® SDK for UPnP™ Devices not only supports UPnP device applications but also UPnP control point applications as well. The basic steps for a control point application are:

1. Set up and initialize the control point following these basic steps:
    a. Initialize the SDK using `UpnpInit()`.
    b. Register a control point (also known as a client) callback function using `UpnpRegisterClient()`.
2. Find interesting devices using `UpnpSearchAsync()`.
3. Download the description documents using `UpnpDownloadXmlDoc()` or the `UpnpHttp()` family of functions.
4. Subscribe to interesting services using `UpnpSubscribe()` or `UpnpSubscribeAsync()`.
5. Have the device do something interesting by changing the state using `UpnpSendAction()` or `UpnpSendActionAsync()`.
6. Shut down the control point following these steps:
    a. Unregister the control point from the SDK using `UpnpUnRegisterClient()`.
    b. Shut down the SDK using `UpnpFinish()`.

In the following discussion, the examples are drawn from the sample TV control point implementation found in `upnp/sample/tvdevice/`. Parts of the sample code have been removed in this document to aid in clarity. For complete descriptions of the SDK for UPnP Devices API functions, refer to the *Intel® SDK for UPnP™ Devices v1.2 API Reference* included with the SDK.

3.1 Setup and Initialization

3.1.1 SDK Initialization

Just like a device, a control point application needs to initialize the SDK.

```
short int port = 0;
char *ip_address = NULL;

rc = UpnpInit( ip_address, port );
if (UPNP_E_SUCCESS != rc) {
      SampleUtil_Print( "UpnpInit() Error: %d", rc );
      UpnpFinish();
      return TV_ERROR;
}
```

The application can specify an IP address and port number during initialization. For control points, this sets the default IP address and port it will use to listen for events. If the IP address is `NULL`, the address of the first non-null, non-loopback address is used. If the port number is 0, a random port number is used. You can retrieve both the IP address and port number from the SDK after initialization by using `UpnpGetServerIpAddress()` and `UpnpGetServerPort()`. For control points, the only benefit for selecting an IP address is to listen on a particular interface in a multi-interface configuration. There is no real benefit to selecting a fixed port.

3.1.2 Control Point Application-Specific Initialization

Prior to registering the control point callback function with the SDK (see section 3.1.3), the control point application should perform any application-specific initialization. This is important because on the application registers the callback, it can immediately start receiving callbacks.

3.1.3 Control Point Registration

The next step is to register the client callback function with the SDK. This callback function is the default notification method the SDK will use. Some functions, such as `UpnpSendActionAsync()`, allow a different callback function for each call. The same function may be specified for all asynchronous operations since the prototype is the same. As with devices, the callback function has a prototype like this:

```
int CallbackFxn( Upnp_EventType EventType, void* Event, void* Cookie );
```

All searching operations will use the default callback registered via `UpnpRegisterClient()`.

```
rc = UpnpRegisterClient( TvCtrlPointCallbackEventHandler, &ctrlpt_handle,
                         &ctrlpt_handle );
if (UPNP_E_SUCCESS != rc) {
      SampleUtil_Print( "Error registering CP: %d", rc );
      UpnpFinish();
      return TV_ERROR;
}
```

The first parameter is the callback function the client wishes the SDK to use. The TV sample uses `TvCtrlPointCallbackEventHandler()` for this purpose. The second parameter is a pointer to a cookie that will be passed to the callback function when invoked. The TV sample passes the control point handle so that it may make SDK calls during the callbacks. The final parameter is a pointer to store the actual control point handle itself. This handle will be necessary for making any subsequent SDK calls.

The TV sample uses the `TvCtrlPointCallbackEventHandler()` function for all asynchronous operations. Because of this, the function is long and will not be included here in its entirety. In the following sections, pieces of this function will be included pertaining to the specific topic to illustrate how the sample handles the callbacks the SDK generates.

Once the control point application calls `UpnpRegisterClient()`, any device advertisement traffic on the network will immediate generate callbacks to the application. The application needs to be ready to handle these.

3.2 Searching for Something Interesting

Once the control point application is completely initialized, it can start searching for interesting devices on the network. The TV sample is a very simple control point and searches for only one device: the TV Sample Device.

```
rc = UpnpSearchAsync( ctrlpt_handle, 5, TvDeviceType, NULL );
if (UPNP_E_SUCCESS != rc) {
      SampleUtil_Print( "Error sending search request%d", rc );
      return TV_ERROR;
}
```

The sample starts looking for the TV device in `TvCtrlPointRefresh()`. `UpnpSearchAsync()` starts the process of finding devices. It takes the following parameters:

- the control point handle
- the number of seconds the control point is willing to wait for responses (5)
- the target for the search

- an optional cookie to pass to the callback function when invoked (`NULL`)

The search target can specify something as specific as a particular device to devices or services of a particular type. Section 1.2.2 of the *Universal Plug and Play Device Architecture* discusses search targets in detail. In summary, a search target must match one of the following:

- `ssdp:all` – searches for all UPnP devices and services on the network
- `upnp:rootdevice` – searches for only the root devices on the network
- `uuid:`*`device-UUID`* – searches for a specific device on the network matching *device-UUID*
- `urn:schemas-upnp-org:device:`*`deviceType:v`* – searches for a particular device type, *deviceType*, with a specification version matching *v*
- `urn:schemas-upnp-org:service:`*`serviceType:v`* – searches for a particular service type, *serviceType*, with a specification version matching *v*

For the TV device, the search target is defined as this:

```
char TvDeviceType[] = "urn:schemas-upnp-org:device:tvdevice:1";
```

The search time, called MX in the *Universal Plug and Play Device Architecture*, specifies the maximum time a control point is willing to wait for matching responses to a search. Devices will wait a random time between 0 and the MX value specified by the control point before responding to avoid discovery storms on searches. The Intel SDK generates a special callback, `UPNP_DISCOVERY_SEARCH_TIMEOUT`, when this value expires. The SDK will generate no further callbacks for the particular search, although device advertisements will continue to generate callbacks.

The TV sample control point handles all these discovery messages in the `TvCtrlPointEventHandler()` function:

```
int TvCtrlPointCallbackEventHandler( Upnp_EventType EventType,
                                     void *Event,
                                     void *Cookie)
{
      switch ( EventType ) {
            case UPNP_DISCOVERY_ADVERTISEMENT_ALIVE:
            case UPNP_DISCOVERY_SEARCH_RESULT:
            {
                  struct Upnp_Discovery *d_event = (
                        struct Upnp_Discovery * ) Event;
                  IXML_Document *DescDoc=NULL;
                  int ret;

                  if ((ret = UpnpDownloadXmlDoc( d_event->Location,
                                                 &DescDoc ))
                        != UPNP_E_SUCCESS) {
                        /* … */
                  } else {
                        TvCtrlPointAddDevice( DescDoc, d_event->Location,
                                              d_event->Expires );
                  }

                  if (DescDoc) ixmlDocument_free( DescDoc );
                  TvCtrlPointPrintList();
                  break;
            }

            case UPNP_DISCOVERY_SEARCH_TIMEOUT:
                  /* Nothing to do here... */
                  break;

            case UPNP_DISCOVERY_ADVERTISEMENT_BYEBYE:
            {
                  struct Upnp_Discovery *d_event =
                        (struct Upnp_Discovery * ) Event;
                  TvCtrlPointRemoveDevice(d_event->DeviceId);
                  TvCtrlPointPrintList();
                  break;
            }
      }
}
```

Devices need to be removed from the list of known devices when the advertisements expire or when the device explicitly sends out a "bye-bye" message. "Bye-bye" messages are handled as shown. Advertisement expirations are handled by calling `TvCtrlPointTimerLoop()` every 30 seconds, calling `TvCtrlPointVerifyTimeouts()` to check if any advertisements have expired. The sample does an automatic search for devices that are about to expire by searching for the expiring device's UDN. Normally, this is not required because the device should refresh its advertisement prior to expiration.

3.3 Retrieving Descriptions

The sample handles advertisements and search results in the same manner: it uses `UpnpDownloadXmlDoc()` to retrieve the description document and adds that device to its list of known devices. It is very important to destroy the description document returned from `UpnpDownloadXmlDoc()` when it is no longer necessary. The sample does not keep this document around but other control point applications might want to do this.

`UpnpDownloadXmlDoc()` returns a completely parsed DOM document of the description document, ready to be consumed by the control point application. This function can take a lot of memory if the description document is large, since it is downloaded, parsed, and returned in one chunk. An alternate API that the SDK offers allows much larger HTTP transfers by breaking the transfer down into chunks. Unlike `UpnpDownloadXmlDoc()`, it requires multiple calls: `UpnpOpenHttpGet()` creates a new HTTP transfer, `UpnpReadHttpGet()` to transfer a piece of the file, and `UpnpCloseHttpGet()` to complete the connection. Refer to the *Intel® SDK for UPnP™ Devices v1.2 API Guide* for more information on these functions.

3.4 Watching for Events

Each time a state variable is changed on a device, the device sends out event notifications to all control points that have registered to receive these events. The SDK has two functions to subscribe to a service: `UpnpSubscribe()` and `UpnpSubscribeAsync()`. Both functions perform the same operation, the latter generating a callback when the subscription request completes. It is important to note that when subscribing, the control point subscribes for all events for a particular *service.* Subscriptions for particular events are not currently supported by the UPnP 1.0 architecture. Also, the control point needs to subscribe to each service it is interested in separately. UPnP 1.0 also does not allow "bulk" subscriptions to all services a device offers.

The sample subscribes to the TV service in `TvCtrlPointAddDevice()` in response to a search request or when a device sends out an advertisement.

```
ret = UpnpSubscribe( ctrlpt_handle,
                     eventURL[service],
                     &TimeOut[service],
                     eventSID[service] );

if(ret == UPNP_E_SUCCESS) {
      SampleUtil_Print( "Subscribed to EventURL with SID=%s", eventSID[service] );
} else {
      SampleUtil_Print( "Error Subscribing to EventURL -- %d", ret );
      strcpy( eventSID[service], "" );
}
```

`UpnpSubscribe()` takes the follow parameters:

- the control point handle
- the URL of the service in which to subscribe
- a pointer to a requested timeout value. Upon return, this will contain the actual life of the subscription if the device did not like the value the control point passed
- a pointer in which to store the Subscription ID

`UpnpSubscribeAsync()` takes similar values on input, but the SID and the actual timeout value are given to the application during the callback rather than when the function returns.

The SDK sends events to the default callback function registered via `UpnpRegisterClient()`. For the TV sample, that function is `TvCtrlPointCallbackEventHandler()`.

```
int TvCtrlPointCallbackEventHandler( Upnp_EventType EventType,
                                     void *Event,
                                     void *Cookie )
{
      /* … */

      switch ( EventType ) {

      /* … */

      case UPNP_EVENT_RECEIVED:
      {
            struct Upnp_Event *e_event = (struct Upnp_Event * ) Event;

            TvCtrlPointHandleEvent( e_event->Sid,
                                    e_event->EventKey,
                                    e_event->ChangedVariables );
            break;
      }
      /* … */
      return 0;
}
```

The `UPNP_EVENT_RECEIVED` callback is an actual event received from a device. The `Event` parameter contains a `Upnp_Event` structure describing the actual event. The sample dispatches these events to `TvCtrlPointHandleEvent()` to handle the event.

Once a control point subscribes to a service, the SDK will automatically renew the subscription until the device explicitly unsubscribes.

3.5 Invoking Actions

Control points cause a device to do something by changing the internal state of the device. It does this by sending actions to the device. The SDK has two functions for changing state variables inside of a device: `UpnpSendAction()` and `UpnpSendActionAsync()`. Both of these functions accomplish the same thing except the latter operates asynchronously. Each of these functions takes a DOM document that describes the action the control point wishes to execute. Section 3.2.1 in the *Universal Plug and Play Device Architecture* discusses the format of these messages. The SDK has utility functions to assemble these DOM documents: `UpnpMakeAction()` and `UpnpAddToAction()`. The TV sample takes full advantage of these utility functions in `TvCtrlPointSendAction()`:

```
IXML_Document *actionNode = NULL;

if (0 == param_count) {
      actionNode = UpnpMakeAction(actionname, TvServiceType[service], 0, NULL);
} else {
      for (param = 0; param < param_count; param++) {
            if (UpnpAddToAction( &actionNode, actionname,
                                 TvServiceType[service],
                                 param_name[param],
                                 param_val[param]) != UPNP_E_SUCCESS) {
                  /* Handle error… */
            }
      }
}

rc = UpnpSendActionAsync( ctrlpt_handle,
                          devnode->device.TvService[service].
                          ControlURL, TvServiceType[service],
                          NULL, actionNode,
                          TvCtrlPointCallbackEventHandler, NULL );

if (rc != UPNP_E_SUCCESS) {
      SampleUtil_Print( "Error in UpnpSendActionAsync -- %d", rc );
      rc = TV_ERROR;
}
```

If the action does not require any parameters, `UpnpMakeAction()` is all that is necessary to build up the correct `actionNode` for the action. Otherwise, the sample calls `UpnpAddToAction()` repeatedly to add all the necessary parameters and values to the `actionNode` document. Finally, it calls `UpnpSendActionAsync()` to send the action message to the device.

`TvSendCtrlPointAction()` is a generic function for sending actions used in the sample. An example of calling this function for turning the TV power on is:

```
int TvCtrlPointSendPowerOn(int devnum)
{
      return TvCtrlPointSendAction( TV_SERVICE_CONTROL,
                                    devnum,
                                    "PowerOn",
                                    NULL,
                                    NULL,
                                    0 );
}
```

When the asynchronous action completes, the callback is generated to the callback handler passed either through `UpnpSendActionAsync()` or registered with `UpnpRegisterClient()`. The sample prefers to use the same callback handler for everything so the action complete message ends up in `TvCtrlPointCallbackEventHandler()`:

```
case UPNP_CONTROL_ACTION_COMPLETE:
{
      struct Upnp_Action_Complete *a_event =
            (struct Upnp_Action_Complete *) Event;

      if (a_event->ErrCode != UPNP_E_SUCCESS) {
            SampleUtil_Print( "Error in  Action Complete Callback -- %d",
                              a_event->ErrCode );
      }

      /* No need for any processing here, just print out results.  Service state
         table updates are handled by events. */

      break;
}
```

3.6 Shutting Down

When the control point application shuts down, it needs to unregister itself from the SDK using `UpnpUnRegisterClient()` and shut down the SDK using `UpnpDeInit()`.

```
int TvCtrlPointStop( void )
{
      TvCtrlPointRemoveAll();
      UpnpUnRegisterClient( ctrlpt_handle );
      UpnpFinish();
      SampleUtil_Finish();

      return TV_SUCCESS;
}
```