Intel® SDK for UPnP™ Devices Programming Guide

Intel® SDK for UPnP™ Devices Version 1.2.1

November 2002

Intel® SDK for UPnP™ Devices 编程指南

Intel® SDK for UPnP™ Devices Version 1.2.1

November 2002

翻译说明：由于要研究 UPnP 协议并在 Linux 下实现其部分设备的功能，在阅读 Intel SDK 自带的英文文档时，顺便把该篇入门文章翻译出来，本翻译是本人的第一份完整的技术翻译文档，其中有些词的翻译恐有不妥之处，有疑问的地方以英文文档为准，英文原文在$(LIBUPNP)/upnp/doc/UPnP_Programming_Guide.PDF.本文取自 libupnp-SDK-1.3.1 版本下的文档，文档中标记还是 1.2.1，估计是开发包升级而文档没有升级吧，暂且以此学习之。

本文档翻译为个人兴趣爱好，尚未联系原作者取得许可，也不对文中任何内容和翻译的问题负责，只作学习交流之用，如有好的建议或意见可与我联系，本文档可以任意的复制，分发，传播和改进。详情参照 GPL。

2006 年 7 月 1 日　　简体中文第一版　　译者：Chaoshua [chaoshua@ieee.org]

目 录

1、概述

UpnP 允许不经过用户干预而自动发现和控制网络上其他设备提供的可用服务。设备（Devices）作为服务器向客户端发布自己的服务。客户端系统，称为控制点（Control Points),能够搜索网络上特定的服务。当他们发现拥有他所期望的服务的设备时，他们就能够检索关于设备的服务的更详细的描述信息并在这个控制点上与其进行交互。

本文档给出 UPnP 的概述并给出如何编写设备和控制点的例子，完整的描述请参见 Intel® SDK for UPnP Devices API 函数，参考 SDK 中包含的《Intel® SDK for UPnP™ Devices v1.2 API Reference》。

开发包（SDK）中也提供了控制点和设备的示例程序，对于如何编译和运行这些示例程序，请参看开发包对应 sample 目录下的 README 文件。

1.1、UPnP 概述

本节简要描述 UPnP，对于更详细的信息，请参见《Universal Plug and Play Device Architecture》，该文档可在 UPnP 论坛找到：

http://www.upnp.org/resources/documents.asp 。

UpnP 包含一下五个基本部分／阶段（Phases):

1、发现（Discovery)：这是第一阶段，控制点搜索设备和服务，类似的，设备广播他所能提供的服务通告。

2、描述(Description)：一旦控制点发现了他感兴趣的设备或服务，他将请求该设备提供该设备本身、其他组件成员设备以及他们提供的服务的完整描述。

3、控制(Control)：本阶段允许控制点通过改变设备的状态来控制设备上提供的一个或多个服务。

4、事件(Eventing)：本阶段允许控制点与其感兴趣的服务状态保持同步，控制点向事件服务器订阅一个特定的服务，当该服务状态改变时，接收事件通告。

5、表示(Presentation)：表示阶段允许设备保持一份文档，该文档采用标准 HTML 语言编写，他可以是该设备的一个用户界面。

接下来的部分将对这五个阶段分别进行描述。

1.1.1、发现（Discovery）

在发现阶段，控制点采用 SSDP（简单服务发现协议：Simple Service Discovery Protocol)发现设备和服务而设备采用 SSDP 向控制点宣告他们的存在。SSDP 采用 HTTP 的一种变体以 UDP 多播的方式来进行广播，并采用另一种 HTTP 变体通过 UDP 单播来进行应答。

一个设备可能包含其他设备，每个设备都有他自己的服务。设备采用其类型和一个唯一的标志符来进行标识。服务则用他们的类型来标识。

为了搜索网络上的设备或服务，控制点使用 UDP 多播包向地址 239.255.255.250:1900 发送 HTTP 的 M－SEARCH 命令。任何网络上服务控制点搜索条件的设备发回一个 UDP 单播进行

应答，该应答中包含了指向其描述文档（参加 1.1.2 节）的 URL 地址。如果一个控制点收到一个或多个可接受的应答，他将转入描述阶段(description phase).

一个控制点发出一个搜索请求时，该请求在 SSDP 头中包含了他愿意等待的时间长度。匹配的设备将在响应之前随机等待一段时间，该时间介于 0 和控制点指示的时间之间。如果控制点在他的搜索时间超时之前没有收到任何应答，他就认为当前网络上没有匹配的设备。

设备没必要等待控制点来搜索他们的服务。他们可以采用向 239.255.255.250:1900 多播地址发送 SSDP 的 NOTIFY 命令来宣告他们的设备可用性。当控制点获得该 NOTIFY 多播，他们就可以使用标准的 HTTP GET 命令来向 NOTIFY 消息中提供的 URL 地址发出请求以获得设备的描述文档。设备必须在他们的服务不可用时发出一个通告信息。

1.1.2、描述（Description）

当控制点定位一个服务后他希望了解更多，因而他将请求描述文档。描述是一个 XML 文档用来描述一个设备，包括:

- 制造商信息，版本，其他。
- 可被设备采用的图标的 URL 地址。
- 嵌入式设备列表。
- 设备提供的服务列表。

想得到关于描述文档的格式的更多信息，请参见文档《Universal Plug and Play Device Architecture》（《通用即插即用设备架构》）。

控制点采用基于TCP的HTTP来请求描述文档。控制点执行标准的HTTP GET命令（与检索Web页面类似）。在服务器端，设备运行一个标准的HTTP服务－－可以是完全的Web服务器如Apache也可以是迷你服务器。描述文档中的很多条目都是URL地址。这些条目也使用HTTP/TCP检索。

1.1.3、控制（Control）

一旦一个控制点发现了一个设备并且检索到他的描述文档，他可能需要控制该设备包含的一个或多个服务。简单对象访问协议（SOAP: Simple Object Access Protocol）允许一个访问点查询或改变服务状态表中的元素。SOAP使用基于TCP传输HTTP的POST和M－POST命令。

SOAP使用XML来说明采取的活动。控制点如描述文档里指定的那样为服务创建XML文档并将其提交给控制URL。控制点能请求服务状态表中的当前值并改变他们。

在服务器端，控制服务器等待控制请求。控制服务器是一个实现了SOAP协议的类似于HTTP服务器的服务器。一个设备能运行多于一个控制服务器，这取决于设备提供的服务的组合。

1.1.4、事件（Eventing）

一个控制点发现一个设备并检索到他的描述后，他能保持设备提供的服务的状态信息。感兴趣的控制点为特定服务订阅了从描述文档中发现的设备事件提醒服务URL。一个事件提醒在任何服务状态改变的时候发送给控制点，即使这次改变是由该控制点产生的。

订阅和退订请求使用HTTP/TCP连接到事件URL，该URL包含在服务的描述文档中。在订阅时，控制点指定一个事件提醒的URL。事件以HTTP/TCP方式到达为服务注册的URL。事件提醒包含

一个小型的XML文档，该文档描述了实际的事件，如服务状态表的改变。

在服务器端，一个事件服务器等待订阅和退订请求。事件服务器是一个类HTTP服务器的实现通用事件提醒架构协议（GENA: General Event Notification Architecture）的服务器。一个设备将可能根据设备提供的服务的组合（情况）必须运行多于一个的事件服务器。

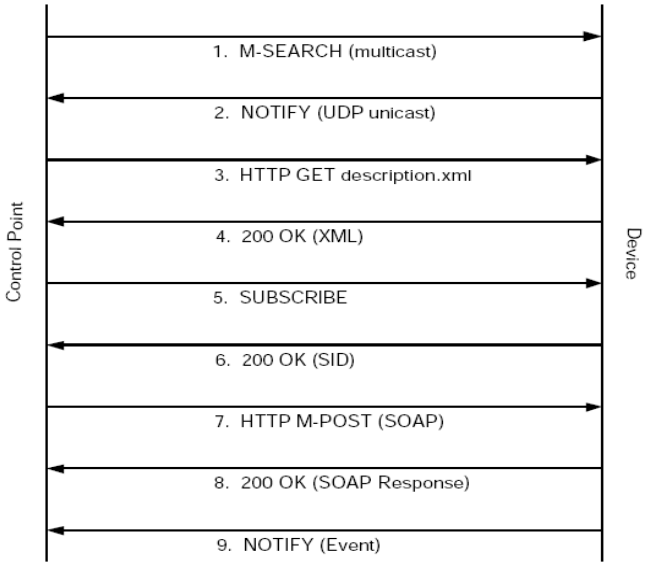
1.1.5、表示（Presentation）

由于设备需要或支持用户交互，在表示阶段一个控制点能下载一个为设备描述用户界面的HTML文档。这是一个能提供一种控制或状态显示的标准HTML文档。

如检索描述文档一样，检索表示文档的协议也是基于TCP的HTTP协议。控制点使用描述文档中包含的表示URL来请求表示文档。不是所有设备都拥有表示文档也不是所有控制点能够显示包含复杂HTML对象如框架，嵌入式Java applets等的表示文档。

1.1.6、控制点和设备的交互

下图显示了前面章节描述的在UPnP各阶段控制点和设备之间进行交互的（动作）序列。图后对每一步进行了描述。这些描述指示出了控制点和设备采用的API调用和SDK中产生的网络包类型。需要注意的是，由于该序列的异步本性，交互本身并不是必须按显示的顺序发生，控制和事件步骤能以任意顺序发生。



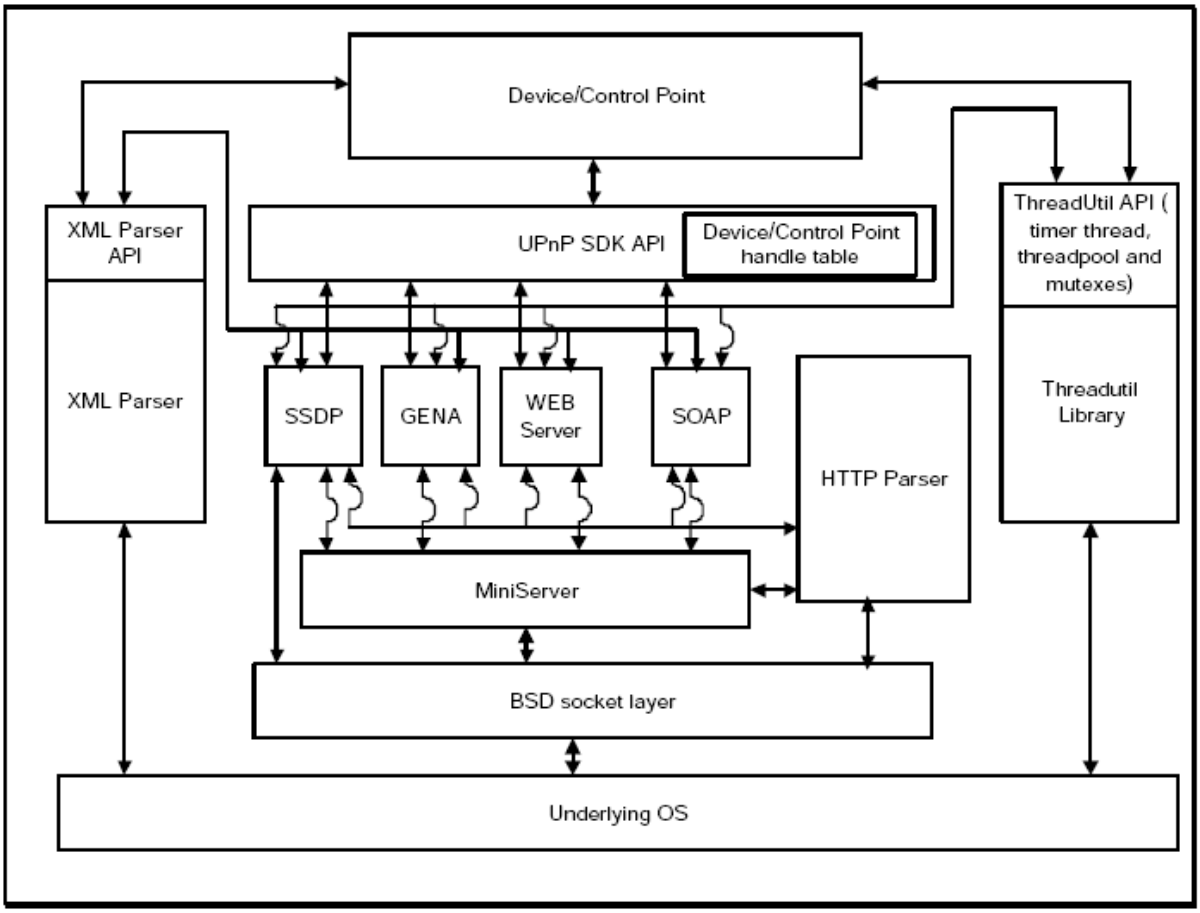
图一、控制点和设备的交互序列

1. 控制点使用UpnpSearchAsync() API发出一个搜索请求。UPnP设备开发包（SDK）产生一个多播M-SEARCH SSDP消息并发送到网络上。
2. 如果设备匹配了控制点的搜索，SDK产生一个单播UDP的NOTIFY响应，该响应包含指向设备描述文档的URL。SDK使用UpnpRegisterRootDevice()或UpnpRegisterRootDevice2()注册的设备描述文档包含的信息自动响应。
3. 如果控制点需要设备的更多信息，他将调用UpnpDownloadXmlDoc()，该调用使用设备响应给出的URL作为参数。UpnpDownloadXmlDoc()使用标准的HTTP GET请求下载XML描述文档并且返回一个代表设备描述的DOM文档。这一步可能重复执行以检索设备的服务描述文档。

4. 设备上包含的Web服务器对请求作出响应并返回XML描述文档。
5. 为了自动接收设备变化提醒，控制点订阅他感兴趣的服务。控制点通过UpnpSubscribe()或UpnpSubscribeAsync()订阅。控制点从他想订阅的服务的设备描述文档中提取出订阅URL，并且调用其中一个订阅函数。对每一个订阅调用，SDK通过包含一个URL的HTTP发送一个SUBSCRIBE消息给发送事件者。
6. 设备对订阅请求进行应答并返回一个唯一的订阅标识符。
7. 控制点通过改变设备中包含的某个状态变量来指示设备执行某种动作。发送控制请求的URL包含在设备描述文件中。控制点调用UpnpSendAction()或UpnpSendActionAsync()来改变状态。SDK通过HTTP M-POST命令产生一个SOAP动作。
8. 设备更改其内部变量的状态并产生一个SOAP应答消息。
9. 设备能通知客户端其状态的改变，不管其改变是由于如第8步中的显示动作还是设备自身的隐式改变。设备调用UpnpNotify()或UpnpNotifyExt()来发送更新。SDK通过HTTP的单播NOTIFY消息自动通知所有已订阅的控制点。

1.2、SDK架构

下图显示了SDK的架构：



图二、SDK架构高层示意图

接下来的章节将对该图的每个部分进行描述，想了解协议有关的任何信息，请参见文档《Universal Plug and Play Device Architecture》（《通用即插即用设备架构》）。

1.2.1、设备/控制点程序

客户提供客户端或服务器程序运行在 SDK 的顶层。客户端或服务器程序实现一个特定服务的功能性。例如，一个网关服务，服务器软件实现“Internet 使能”(Enable Internet)功能，控制点软件能够使用 UPnP 控制它。SDK 包含了一个示例服务和控制点程序作为其一部分。

1.2.2、SDK API

SDK API 从控制点和服务程序中抽象出核心 UPnP 协议的细节并给应用程序访问功能提供一个统一的接口。这使得开发者可以从他们关注 SSDP，GENA 和 SOAP 协议细节的编码中解脱出来。

API 层同时维护在 SDK 中注册的控制点和设备句柄的表格。对于每一次 API 函数的调用，SDK 将通过检查句柄表格来验证该句柄。目前，在一个进程中一次只能有一个控制点和一个设备句柄。换句话说，一个应用程序一次只能注册为一个设备和一个控制点。该程序的任何进一步的注册请求都将失败。

关于 API 的更多信息，参见《Intel® SDK for UPnP™ Devices v1.2 API Reference》。

1.2.3、SSDP

SSDP 模块实现了简单服务发现协议（Simple Service Discovery Protocol），提供 UPnP 的发现部分（phase)。这个模块允许控制点发送多播信息搜索网络上的服务和设备并接受应答。当网络上有新服务通告时，它也提供通知提醒。

该模块同时允许设备在网络上多播他们的服务通告。

1.2.4、迷你 Web 服务器（Mini Web Server)

迷你 Web 服务器模块处理标准的 HTTP GET 请求。许多 UPnP 元素（elements)都被用这种基本的 HTTP 服务请求。该模块管理那些能 GET 命令获得的文档的地址并实现了使用 HTTP 协议的实际的数据流。

迷你 Web 服务器模块实现了 HTTP/1.1 的 RANGE 头。这个头允许一个远程客户端请求一个文件中的特定的一片（或多片）而不是整个文件。一个实现该功能的应用程序可以在一个播放列表中查找到特定的音轨或者跳到一个媒体文件中的某个偏移位置。

迷你 Web 服务器也支持 HTTP POST 请求，他只为支持虚拟目录而非其他，关于虚拟目录的定义，参见 1.3 节。

1.2.5、GENA

GENA 模块实现通用事件提醒架构（General Event Notification Architecture）,他实现 UPnP 的事件部分（Eventing Phase)。控制点使用该模块订阅和退订感兴趣的服务。服务应用程序从该模块中接收订阅和退订提醒并产生相应的事件。

1.2.6、SOAP

SOAP 模块实现简单对象访问协议(Simple Object Access Protocol)，他提供 UPnP 的控制部分(Control Phase)。控制点使用该模块产生相应的 XML 文档来检索和改变一个服务的状态表。服务器使用该模块解码控制请求并产生正确的应答。

1.2.7、HTTP

HTTP 语法分析模块对接收的信息的 HTTP 头进行语法分析同时构建相应的发送信息的头。他能够处理 HTTP/1.0 和 HTTP/1.1 头。他也支持 HTTP/1.1 的分块编码语法。SDK 缺省不支持使用 HTTP/1.1 的 GET 请求的分块编码。不过，当接受到分块编码信息时，他完全支持对他们的解码。

1.2.8、迷你服务器

迷你服务器层为 GENA，SOAP，SSDP 和迷你 Web 服务器的通用功能。该层接收所有的网络连接，确定请求来自何处，将 HTTP 头分离出来传给 HTTP 模块进行语法分析，将连接转移／传递（transfer)给相应的协议（模块）进行处理。HTTP 头的第一行包含了请求（信息）。迷你服务器层最少要处理以下命令：

- GET：控制点使用 GET 命令来检索描述文档和他的任何子元素包括表示文档，服务描述文档，与设备关联的图标。使用 1.3 节中描述的虚拟目录，设备应用程序能够请求 SDK 为应用程序产生回调（函数）来为特定目录处理 GET 请求。参见 1.3 节了解更多。
- POST/M-POST：一个 SOAP 命令是以一个 POST 或 M-POST 命令的形式出现。所有提交到（posted to)设备描述文档指定的控制 URL 的 POST 命令都被转移到 SOAP 模块进行进一步的处理。POST 命令同样支持虚拟目录，为应用程序产生回调。详见 1.3 节。
- SUBSCRIBE：SUBSCRIBE 命令被转移／传递给 GENA 模块进行后期处理。控制点使用 SUBSCRIBE 请求来订阅或更新一个为特定服务订阅的事件提醒。
- UNSUBSCRIBE：UNSUBSCRIBE 命令也是传递给 GENA 模块进行进一步处理。控制点使用 UNSUBSCRIBE 命令来提醒服务器他们不再希望接收事件提醒。
- NOTIFY：NOTIFY 命令被传递给其他模块进行进一步处理。在采用 TCP 连接时，他们被传递给 GENA 模块（处理）；当使用 UDP 连接时（原文用 UDP connections),他们被传递给 SSDP 模块（处理）。NOTIFY 命令可以是从服务器发给控制点的事件提醒，他包含一个事件的描述；他或者可以是一个设备或服务出现在网络上或者从网络上消失时的提醒。

1.2.9、ThreadUtil 库（TreadUtil Library)

Intel UPnP SDK 广泛使用线程来尽量的并行处理 UPnP 事务（traffic)。ThreadUtil 库为 SDK 提供一个类似 POSIX(POSIX-like)的线程 API，线程管理例程和连接和非连接的链表管理工具。

1.3、虚拟目录

集成在 SDK 中的迷你 Web 服务器只是一个称为虚拟目录的概念。一个虚拟目录是一个 HTTP

客户端可访问的目录，他不是与迷你服务器的根目录结构的物理结构相匹配。通常，发给 Web 服务器的 URL 与文件在 Web 服务器存储的实际物理结构一致。Web 服务器将客户端传来的 URL 前缀映射成根目录并在文件系统中打开那个文件，将数据反馈回客户端。使用一个虚拟目录，一个设备应用程序能够将它发送请求并希望接收回调的地方注册为一个特定的目录。例如，假设一个设备拥有一个如下的目录结构：

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

为检索到 index.html，一个服务器可能使用一个标准的 HTTP GET 请求：

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

假如设备应用程序注册了一个虚拟目录叫“media”。一个客户端则可以使用一个请求如：

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

设备应用程序从迷你 Web 服务器获得一个回调，他要求设备应用程序为客户端返回数据。迷你服务器并不关心数据来自哪里，他可能来自互联网，或者从迷你 Web 服务器的根目录之外读取一个文件，也可能是动态产生的。

设备应用程序用来接收回调的 API 与标准的文件接口非常相似，他包含一个六个函数指针的结构：

get_info()是请求的第一个回调函数。他将一个含有客户请求 URL 信息的结构体传给应用程序。应用程序将文件信息如文件大小等传回给迷你 Web 服务器。该信息变成了 HTTP 回复头的基础。

open()取回一块数据。迷你 Web 服务器在一个 HTTP GET 请求中反复的调用该函数直到该函数不再返回任何数据。

write()写入一块数据。迷你 Web 服务器在一个对一个虚拟目录的 HTTP POST 请求中反复的调用该函数。

seek()在一个文件中改变位置。迷你 Web 服务器主要使用该函数来满足请求一个文件中特定偏移的 HTTP RANGE 请求。

close()关闭文件句柄。

一个设备应用程序通过 UpnpSetVirtualDirCallbacks()来注册这些回调函数。添加一个新的虚拟目录列表映射使用 UpnpAddVirtualDir()。注意，传给 UpnpAddVirtualDir()的目录变成了迷你 Web 服务器的前缀来决定他是否产生一个回调。UpnpRemoveVirtualDir()用来移除一个单一的虚拟目录映射，UpnpRemooveAllVirtualDirs()用来移除所有映射。关于这些函数的详细信息，参阅《Intel® SDK for UPnP™ Devices v1.2 API Reference》。

2、编写一个 UPnP 设备

使用 UPnP 设备开发的 SDK 实现一个 UPnP 设备有很多方法，然而，任何实现都必须完成一些基本步骤。确切来说，一个应用程序必须：

1、使用以下基本步骤安装和初始化一个设备：

a、使用 UpnpInit()初始化 SDK。

b、使用 UpnpRegisterWebServerRootDir()设置迷你 Web 服务器的根目录。

c、使用 UpnpRegisterRootDevice()或者 UpnpRegisterRootDevice2()来注册一个设备描述文档。

d、完成任何与特定设备有关的初始化工作。

e、使用 UpnpSendAdvertisement()在网络上发布设备通告。

2、处理异步请求。设备需要处理三种不同类型的请求：

a、订阅设备状态改变的提醒消息的请求。

b、检索设备状态变量的当前值的请求。

c、改变设备状态变量的值的请求。

3、通过使用 UpnpNotify()或 UpnpNotifyExt()发送事件消息来保持控制点的信息为最新。

4、遵循以下步骤关闭设备：

a、发出 SSDP 的“bye-bye”消息并且使用 UpnpUnRegisterRootDevice()来从 SDK 注销设备。

b、使用 UpnpFinish()关闭 SDK。

接下来的讨论中，我们使用/upnp/sample/tvdevice/中找到的实现一个 TV 设备的示例作为例子。为保持文档的清晰，部分示例代码被移除，对于 SDK API 函数的完整描述，参见 SDK 中包含的《Intel® SDK for UPnP™ Devices v1.2 API Reference》。

2.1、安装和初始化

2.1.1、初始化 SDK

开始实现设备前，编写或获得一个将要使用的设备和服务描述文档是很重要的。他们指定了设备支持的服务的类型和数量，以及每个服务支持的动作、参数和变量。详见《通用即插即用设备架构》文档或者与特定设备有关的规范。

TV 设备用到的设备和服务描述文档在 upnp/sample/tvdevice/web 目录。需要重点注意的是，这个示例设备不是一个以认证的 UPnP 设备。

应用程序必须在使用任何 API 函数前初始化 SDK：

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

应用程序在初始化时指定 IP 地址和端口号，这被用来设定服务器的接口和端口以侦听 UPnP

和 HTTP 请求。如果 IP 地址为 NULL，[1]。如果端口号为 0，将使用一个随机的端口号。你可在 SDK 初始化之后使用 UpnpGetServerIpAddress()和 UpnpGetServerPort()来检索 IP 地址和端口号。

2.1.2、设置根目录

一旦 SDK 初始化完成，设备应用程序可以指定 Web 服务器的根目录。这是一个本地目录，Web 服务器搜索以为应答 HTTP 请求提供文件。指定根目录是可选的。如果根目录未指定，则仅有对包含在虚拟目录列表中的文档的请求能够通过已注册的回调函数提供。应用程序用于确保指定目录下包含必要的文件（例如设备和服务描述文件）。本示例的 Web 目录是 /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;
  }
```

需要注意的事，一旦 UpnpInit()运行成功，非常重要的是当发生错误要关闭设备时，必须调用 UpnpFinish 以让 SDK 有机会清除（释放）他已分配的资源。

2.1.3、注册一个根设备

安装设备的下一步是在 SDK 中注册，有两个函数用来注册设备，UpnpRegisterRootDevice()和 UpnpRegisterRootDevice2()。第一个函数采用一个完整的（fully qualified)描述 URL 作为输入，第二个可采用各种不同类型的描述文档。下面的例子采用第一种方案：

```
     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;
     }
```

这个函数的第一个参数是描述文档 URL。他必须指向设备的有效描述文档。如果描述文档是包含在 SDK 中的 Web 服务器用来服务的文件，他可能位于 UpnpSetWebServerRootDir()指定的目录之外。第二个参数用 SDK 中注册回调函数。回调函数的原型是：

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

每当设备从网络上接收到一个请求，例如一个订阅请求，一个获取变量请求或者一个动作

1 当前实现版本侦听所有接口，初始化时指定的 IP 地址仅影响 SSDP 通告阶段的被通告的 IP 地址和（对其）搜索的回复。

请求，这个函数将使用合适的参数在一个独立的线程中调用。EventType 指定接收到的请求的类型，Event 参数是一个结构体，他的实际类型因 EventType 的不同而各异。Cookie 是在 UpnpRegisterRootDevice()中指定的应用程序相关的数据。对于 TV 示例注册的回调函数是：

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

UpnpRegisterRootDevice()的第三参数是用户指定的无类型指针。他可以是任何指针并可用于指向应用程序特定的数据。他也可以为空。如果指针被使用，应用程序负责分配和回收该指针。该指针在接收到一个请求时被传回给应用程序。UpnpRegisterRootDevice()最后一个参数是一个指针，用来分配应用程序存储设备句柄的空间。设备句柄是一个其他 API 函数使用的参数。

2.1.4、设备相关初始化

到目前位置，我们都没有讨论与设备和应用程序相关的初始化（操作）。在设备发布公告之前，必须做好所有设备和应用程序相关的初始化。一旦设备被通告出去，他能立即接收请求。设备既要负责维护设备状态变量的值，也要正确的操作他们以响应动作。在 TV 设备示例代码中，函数 TvDeviceStateTableInit()初始化内部数据结构，该数据结构用于存放状态变量，设备 ID，动作指针等。

2.1.5、发布设备公告（Advertising the Device）

安装 UPnP 设备的最后一步就是发出公告：

```
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;
  }
```

第一个参数是注册时返回的设备句柄。第二个参数是公告的超期时间。这设备的生命周期中，SDK 在他超时前会自动重新发布设备公告。

至此，设备以准备好接收请求了。应用程序必须将自己设在一个循环中或者等待（符合）某种条件（时）关闭（设备）。

2.2、处理请求

在设备的生命周期中，他的主要目的就是处理控制点发给他的各种请求。请求被 SDK 接收后，他们被通过设备注册时指定的回调函数转发给应用程序。根据请求的类型不同，回调函数使用适当的参数在一个独立的进程中调用。

设备处理的有三种请求：

- 订阅请求
- 获取变量请求

- 动作请求

注意，设备不需要处理任何发现请求，因为 SDK 拥有该设备的设备描述文档（在执行 UpnpRegisterRootDevice()时传递），他（SDK）能够自动判定搜索发现规则是否与该设备匹配。如果匹配，他将以设备描述文档 URL 作为响应。

请求类型在回调函数中通过 EventType 变量指示。TV 设备示例的回调函数如下：

```
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、订阅请求

当控制点产生一个订阅请求，SDK 将调用注册的回调函数并将 EventType 变量值设为 UPNP_EVENT_SUBSCRIPTION_REQUEST。设备负责通过 UpnpAcceptSubscription()或者 UpnpAcceptSubscriptionExt()来接受订阅。然后发送当前状态表给控制点。这两个函数的差别仅在于应用程序如何将状态表传给 SDK 以发送给订阅的控制点。UpnpAcceptSubscription()采用一对字符串数组来存储“变量/值”对。UpnpAcceptSubscriptionExt()采用 DOM 文档来存放当前变量的值。DOM 文档格式在《Universal Plug and Play Device Architecture》（《通用即插即用设备架构》）文档的 4.3 节中给出。TV 示例使用 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;
}
```

在此例中，传给回调函数的 Event 参数是一个指向结构体类型 Upnp_Subscription_Request 的指针。订阅请求结构用服务的 UDN 和 ServiceID 以及订阅标识符（SID）来指出请求的是哪个订阅。在上例中，一旦服务被标识出来，存储在 tv_service_table[i]中的状态变量将被设备使用 UpnpAcceptSubscription()传给控制点。UpnpAcceptSubscription()带有以下参数：

- 设备句柄（ device_handle ）
- UDN
- ServiceID
- 变量名（tv_service_table[i].VariableName）
- 变量值（tv_service_table[i].VariableStrVal）
- 变量数量（tv_service_table[i].VariableCount）
- 订阅标识符 SID

当订阅（请求）被接受，控制点就可以接收设备发送的后继事件。示例代码用 TVDevMutex 互斥变量来保护对线程处理的全局数据的访问。SDK 回调函数是多线程的，设备实现对于确保对受保护的共享数据的访问是必须的。

2.2.2、获取变量请求

当某个控制点请求一个变量的当前状态，SDK 调用回调函数并将变量 EventType 设为 UPNP_CONTROL_GET_VAR_REQUEST。

需要注意的是，UPnP 论坛已不赞成这种做法。当客户端要访问状态变量，更好的方法是使用一个特定的动作来检索该值。设备对这个功能的支持并不是严格必须的。下面的讨论描述了一个应用程序如何在 SDK 上面支持该功能。

传给回调函数的 Event 变量是一个指向结构体 Upnp_State_Var_Request 类型的指针。该请求结构体指定 UDN,ServiceID 和被请求变量的变量名。设备负责设置该结构体中的变量的当前值。

```
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 );
}
```

该结构体的 CurrentVal 成员必须设置为由 ixmlCloneDOMString()产生的字符串。这部分内存由 SDK 在该值被发送给控制点后释放。同上，TVDevMutex 互斥变量用于多线程的数据访问保护。

2.2.3、动作请求

当某控制点发送一个动作给设备，SDK 调用注册的回调函数并将 EventType 参数的值设置为 UPNP_CONTROL_ACTION_REQUEST。

传给回调函数的 Event 参数是一个指向结构体类型 Upnp_Action_Request 的指针。请求结构

体指定了 UDN,ServiceID,动作名称和动作请求文档（其包含了入口参数）。设备负责操纵该文档，从中抽取出相关的参数，执行请求的动作，创建带出口参数的响应文档（如果适用）以及发送任何事件（如果适用）。在 TV 设备示例中，函数 TvDeviceHandleActionRequest()处理动作请求。这个函数通过按顺序搜索表格中的动作名称和检索相应的函数来调度处理请求。每个 UPnP 动作被一个独立的函数处理，函数原型为：

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

该函数传入了一个指定动作参数的 XML 文档并且要求创建响应文档，同时返回适当的出错字符串。该函数被要求成功时返回 UPNP_E_SUCCESS，否则返回非零的错误代码。

Tv 设备首先查看控制点是在哪个服务上调用一个动作：

```
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;
}
```

基于那个服务，他搜索特定的动作并且为其分派动作处理者：

```
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;
    }
}
```

作为一个例子，这里是一个实现 SetChannel1 动作的函数：

```
int TvDeviceSetChannel1( 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;
    }
}
```

这个函数使用 UpnpAddToActionResponse()公用函数来建立动作响应。

2.3、发送事件（Sending Events）

不管何时一个事件的状态变量发生改变，设备都应该发送一个状态表更新事件。这可以是对一个动作的响应，一个外部事件，用户输入等等。设备应用程序负责决定何时要发送一个事件。SDK 将事件发给所有已订阅的控制点。在 TV 设备示例中，事件作为对动作的响应发出，事件进程（Eventing)由函数 TvDeviceSetServiceTableVar()处理：

```
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 );
}
```

该函数更新应用程序内部状态表并且使用 UpnpNotify()函数发送事件。

UpnpNotify()采用如下参数:

- 设备句柄(device_handle)
- 设备 UDN
- ServiceID
- 改变的变量的名字 (&tv_service_table[service].VariableName[variable])
- 改变的变量的值 (&tv_service_table[service].VariableStrVal[variable])
- 被改变的变量的数目 (1)

2.4、关闭 (Shutting Down)

当一个设备被被关闭, SDK 必须反初始化 (uninitialized)。这通过调用函数 UpnpUnRegisterRootDevice()和 UpnpFinish()来实现。TV 示例中在 TvDeviceStop()函数中实现了所有这些功能:

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

3、编写一个 UPnP 控制点

Intel UPnP SDK 不仅支持应用 UPnP 设备应用程序而且同样支持 UPnP 控制点程序。控制点应用程序的基本步骤如下：

1、使用以下基本步骤安装和初始化控制点：

a、使用 UpnpInit()初始化 SDK。

b、使用 UpnpRegisterClient()注册一个控制点（客户端）回调函数。

2、使用 UpnpSearchAsync()查找感兴趣的设备。

3、使用 UpnpDownloadXmlDoc()或者是 UpnpHttp()家族的函数下载描述文档。

4、使用 UpnpSubscribe()或 UpnpSubscribeAsync()订阅感兴趣的服务。

5、使用 UpnpSendAction()或 UpnpSendActionAsync()通过改变状态来让设备执行某些感兴趣的动作。

6、使用以下步骤来关闭控制点：

a、使用 UpnpUnRegisterClient()从 SDK 中注销控制点。

b、使用 UpnpFinish()来关闭 SDK。

以下讨论的是从 TV 示例控制点实现程序中抽取出来的例子，该实现可在目录 upnp/sample/tvdevice 下找到。为保持文档的清晰，部分示例代码被移除，对于 SDK API 函数的完整描述，参见 SDK 中包含的《Intel® SDK for UPnP™ Devices v1.2 API Reference》。

3.1、安装和初始化

3.1.1、SDK 初始化

正如一台设备一样，一个控制点应用程序也需要初始化 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;
  }
```

初始化时应用程序可指定 IP 地址和端口号。对于控制点，这（一步）可以设置控制点用于侦听事件的缺省 IP 地址和端口，如果（这里的）IP 地址是 NULL，那第一个非空，非回环地址将被启用。如果端口号为 0，将使用一个随机端口。你也可以在初始化之后使用函数 UpnpGetServerIpAddress()和 UpnpGetServerPort()来从 SDK 检索到 IP 地址和端口号。对于控制点，选定一个 IP 地址的唯一好处就是可以在多接口配置中侦听一个特定的接口。选定一个固定的端口号没有任何实际意义。

3.1.2、控制点应用相关的初始化

在用 SDK 注册控制点回调函数前，控制点应用程序应该完成任何应用相关的初始化。这非常重要因为一旦应用程序注册了回调函数，他将能够立即开始接收回调。

3.1.3、控制点注册

下一步就是在 SDK 注册客户回调函数。该回调是 SDK 使用的缺省提醒方法。有些函数，如 UpnpSendActionAsync()，允许每一次调用采用不同的回调函数。所有异步操作也可指定相同的回调函数，因为其原型是一样的。如设备一样，（控制点）回调函数原型如下：

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

所有搜索操作都将使用通过 UpnpRegisterClient()注册的缺省回调函数。

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

第一个参数是客户端希望 SDK 使用的回调函数，TV 示例用 TvCtrlPointCallbackEventHandler()来实现该目的。第二个参数是一个指向函数调用时传给回调函数的 cookie 的指针。TV 示例传递控制点句柄，这样在回调时就可以产生 SDK 调用。最后一个参数是一个指针，他用于存放实际的控制点句柄本身。该句柄对于产生任何后继的 SDK 调用是必须的。

Tv 示例对所有异步操作都使用 TvCtrlPointCallbackEventHandler()函数。因为这样，该函数很长，这里不包含他的全部。下面的章节中包含了该函数的从属于一个特定主题的部分片段，用于描述该示例如何处理 SDK 产生的回调。

一旦控制点应用程序调用了 UpnpRegisterClient(),任何在网络上的设备通告流量都将立即产生对该应用程序的回调。应用程序需要为处理这些事情做好准备。

3.2、搜索感兴趣的事物

控制点应用程序完全初始化后，他就可以开始搜索网络上自己感兴趣的设备。TV 示例是一个非常简单的控制点，他只搜索一个设备：TV 示例设备。

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

示例开始在 TvCtrlPointRefreshI()中搜索 TV 设备。UpnpSearchAsync()开始查找设备的处理。他带有以下参数：

- 控制点句柄
- 控制点等待响应的秒数

- 搜索的目标
- 一个可选的在调用时传给回调函数的 cookie（NULL）

搜索目标可指定为某个特定事物如某种特殊类型的设备或服务中的某个特定设备。《Universal Plug and Play Device Architecture》的1.2.2 节详细讨论了搜索目标，简而言之，一个搜索目标必须匹配以下条目中的一项：

- ssdp:all－在网络上搜索所有 UPnP 设备和服务
- upnp:rootdevice－仅搜索网络上的根设备
- uuid:device-UUID－搜索网络上一个与 device-UUID 匹配的特定设备
- urn:schemas-upnp-org:device:*deviceType*:v－搜索一个设备类型为 *deviceType* 并且版本号为 v 的特定设备
- urn:schemas-upnp-org:service:*serviceType*:v－搜索一个服务类型为 serviceType 并且版本号为 v 的特定服务

对于 TV 设备，搜索目标定义为：

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

搜索时间，在《Universal Plug and Play Device Architecture》中称为 MX，指定一个控制点将等待一个搜索返回匹配的响应的最大时间。设备将等待一个随机的时间以防止在搜索时发生发现风暴（discovery storms）。该随机时间介于 0 和控制点指定的 MX 值之间。当该值超时时，Intel SDK 将产生一个特殊的回调函数 UPNP_DISCOVERY_SEARCH_TIMEOUT。SDK 不为特殊的搜索进一步产生回调，虽然设备通告回调会持续的产生回调。

Tv 示例控制点在函数 TvCtrlPointEventHandler()中处理所有这些发现消息：

```
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;
            }
      }
}
```

当设备通告期满或者设备显式的发出了“bye-bye”消息，设备需要从已知设备列表中移除。“Bye-bye”消息如上面显示的那样处理，通告期满则每 30 秒钟调用 TvCtrlPointTimerLoop()处理，调用 TvCtrlPointVerifyTimeouts()来检查任何设备通告是否已期满。本示例采用搜索即将期满的设备的 UDN 来自动搜索即将期满的设备。通常，这并不是必须的因为设备会在期满之前更新他自己的通告。

3.3、检索描述(Retrieving Descriptions)

本例采用相同的方式处理通告和搜索结果：他使用 UpnpDownloadXmlDoc()函数来检索描述文档并将该设备加入他的已知设备列表中。当不需要时销毁 UpnpDownloadXmlDoc()返回的描述文档是非常重要的。本例中没有保留该文档但其他控制点应用程序需要这样做。

UpnpDownloadXmlDoc()返回一个描述文档的完整的经过语法分段的 DOM 文档，提供给控制点应用程序“消费”。如果描述文档较大时，这个函数会占用很多内存，因为他需要下载，语法分析然后大块返回。SDK 提供了一个替代的 API，他将传输文档分成块从而可允许更大的 HTTP 传输。不像 UpnpDownloadXmlDoc(),他需要多个调用：UpnpOpenHttpGet()新建一个 HTTP 传输，UpnpReadHttpGet()用于传输文件的一块，UpnpCloseHttpGet()则用于结束连接。更多信息请参考《Intel® SDK for UPnP™ Devices v1.2 API Guide》。

3.4、监视事件(Watching for Events)

每当一个设备上的某个状态变量改变时，设备会给所有注册了要接收这些事件的控制点发出事件提醒。SDK 有两个函数用于注册一个服务：

UpnpSubscribe()和 UpnpSubscribeAsync()。两个函数实现同样的操作，后一个函数在订阅请求完成时产生一个回调函数。需要重点注意的是，每次订阅，控制点都是订阅某个特殊服务的所有事件。订阅某个特殊的事件在当前的 UPnP 1.0 架构中不支持。同样，控制点需要分别订阅他感兴趣的每一项服务。UPnP 1.0 也不支持一个设备提供的所有服务的捆绑(bulk)订阅。

作为对一个搜索请求或者是设备发出通告时的响应，本示例在 TvCtrlPointAddDevice()中订阅 TV 服务：

```
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()带有如下参数:

- 控制点句柄
- 订阅的服务 URL
- 一个指向请求的超时值的指针，在返回时，如果设备不想采用控制点传来的值，他将包含该订阅的实际生存时间。
- 一个存储订阅 ID 的指针

UpnpSubscribeAsync()输入采用类似的值，但是 SID 和实际超时值是在调用回调函数时给出而不是在本函数返回时给出。

SDK 发送事件到通过 UpnpRegisterClient()注册的缺省回调函数。对于 TV 示例，该函数是 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;
}
```

UPNP_EVENT_RECEIVED 回调是一个从设备接收的实际事件。Event 参数包含一个描述实际事件的 Upnp_Event 结构体。示例分发这些事件给 TvCtrlPointHandleEvent()以处理该事件。

一旦一个控制点订阅到一个服务，SDK 将自动更新该订阅知道设备显式的进行了退订。

3.5、调用动作(Invoking Action)

控制点通过改变设备的内部状态来使得设备做某些事情。他通过发送动作（指令）给设备

来实现这个功能。SDK 有两个函数可用于改变处于一个设备内部的状态变量:

UpnpSendAction()和 UpnpSendActionAsync()。两个函数完成相同的事情，后者实现异步操作。每个函数都带有一个描述控制点希望执行的动作的 DOM 文档（作为参数）。《Universia1 P1ug and P1ay Device Architecture》的 3.2.1 节中讨论了这些消息的格式。SDK 有一些工具函数来组装这些 DOM 文档: UpnpMakeAction()和 UpnpAddToAction()。TV 示例中很好的使用了这些工具函数，见 TvCtr1PointSendAction():

```
 IXML_Document *actionNode = NULL;
if (0 == param_count) {
     actionNode = UpnpMakeAction(actionname, TvServiceType[service], 0, NULL);
} e1se {
     for (param = 0; param < param_count; param++) {
            if (UpnpAddToAction( &actionNode, actionname,
                                 TvServiceType[service],
                                 param_name[param],
                                 param_va1[param]) != UPNP_E_SUCCESS) {
            /* Hand1e error… */
            }
     }
}
rc = UpnpSendActionAsync( ctr1pt_hand1e,
                          devnode->device.TvService[service].
                          Contro1URL, TvServiceType[service],
                          NULL, actionNode,
                          TvCtr1PointCa11backEventHand1er, NULL );
if (rc != UPNP_E_SUCCESS) {
     Sample Uti1_Print( "Error in UpnpSendActionAsync -- %d", rc );
     rc = TV_ERROR;
}
```

如果动作不需要任何参数，UpnpMakeAction()对于为动作创建正确的 actionNode 已经足够。否则，示例反复调用 UpnpAddToAction()将所有必须的参数和值添加到 actionNode 文档中。最后，他调用 UpnpSendActionAsync()将动作信息发送给设备。

本示例中，TvSendCtr1PointAction()是一个发送动作的通用函数。调用该函数以打开 TV 电源的示例如下:

```
 int TvCtr1PointSendPowerOn(int devnum)
{
     return TvCtr1PointSendAction( TV_SERVICE_CONTROL,
                                   devnum,
                                   "PowerOn",
                                   NULL,
                                   NULL,
                                   0 );
}
```

当异步动作完成时，将为通过 UpnpSendActionAsync()传递或者通过 UpnpRegisterC1ient()注册回调处理者（ca11 hand1er)产生回调。本例倾向使用同一个回调处理者来处理所有事情以

使动作完成消息在 TvCtrlPointCallbackEventHandler()终结:

```
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、关闭 (Shutting Down)

当控制点应用程序关闭时，他需要使用 UpnpUnRegisterClient()从 SDK 中注销自己，并且使用 UpnpDeInit()[2]来关闭 SDK。

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

2 译者注：从上面的示例代码来看，这里应该是 UpnpFinish()而非 UpnpDeInit()。