CyberLink for C++ Programming Guide

Document Version 1.7.1

Table of Contents

1 Introduction

UPnP™*[1] architecture is based on open networking to enable discovery and control of networked devices and services, such as media servers and players at home.

UPnP™ architecture is based on many standard protocols, such as GENA, SSDP, SOAP, HTTPU and HTTP. Therefore you have to understand and implement these protocols to create your devices of UPnP™.

CyberLink for C++ is a development package for UPnP™ developers. CyberLink controls these protocols automatically, and supports to create your devices and control points quickly.

Please see the following site and documents to know about UPnP™ in more detail.

Document	URL
UPnP™ Forum	http://www.upnp.org/
Universal Plug and Play Device Architecture	http://www.upnp.org/download/UPnPDA10_20000613.htm
Universal Plug and Play Vendor's Implementation Guide	http://www.upnp.org/download/UPnP_Vendor_Implementation_Guide_Jan2001.htm

[1] UPnP™ is a certification mark of the UPnP™ Implementers Corporation.

2 Setup

2.1 Package Contents

The CyberLink package has the header files, the source files, the project files to build the package and the some samples. The files are included the following directories.

File Type		Directory
Source files		CyberLink/src
Header Files		CyberLink/include
Sample files		CyberLink/sample
Project files	Unix (Automake)	CyberLink
	WindowsXP (VisualC++ 6.0)	CyberLink/*/win32/vc60
	MacOSX (Project Builder) (Xcode)	CyberLink/*/macx/ProjectBuilder CyberLink/*/macx/xcode
	T-Engine (GNU)	CyberLinkC/*/tengine/gnu
	uITRON	CyberLinkC/*/itron

On MacOSX platform, the package is distributed using Disk Copy utility. To build the package, you have to copy the all files from the mounted disk image folder into your local file systems.

2.2 System Requirement

CyberLink needs the following package to parse the XML and SOAP requests. Please get the parser package and install in your platform.

Package	URL
Apache Xerces C++	http://xml.apache.org/xerces-c/index.html

On Unix Platform, the samples are built using the window system if GTK+/GDK and Imlib are installed in the platform. Otherwise, the samples are built using the command line version.

2.2.1 WindowsXP

On Windows platform, you have to install latest Platform SDK and build on WindowsXP if you can. Please get the SDK and install in your platform.

Package	URL
Platform SDK	http://www.microsoft.com/msdownload/platformsdk/sdkupdate/

2.2.2 T-Engine

On T-Engine platform, you have to use the following development kit based on GNU GCC and TCP/IP protocol stack that supports the multicast protocol. The CyberLink uses the multicast protocol to search and announce UPnP devices and you have to use the protocol stack because the old package doesn’t support the multicast protocol.

Package	URL
T-Engine Development Kit	http://www.personal-media.co.jp/te/welcome.html
KASAGO for T-Engine	http://www.elwsc.co.jp/japanese/products/kasago_tengine.html

The CyberLink supports the following TCP/IP protocol stack for T-Engine too, but the protocol stack doesn't support the multicast protocol and the functions are not implemented yet.

Package	URL
PMC T-Shell Kit	http://www.personal-media.co.jp/te/welcome.html

On T-Engile, the CyberLink needs the following package as the default XML parser. Please get the parser package and install in your platform.

Package	URL
Expat	http://expat.sourceforge.net/

On MacOSX platform you have to install latest Project Builder and gcc. Please get the tools and install in your platform.

Package	URL
Developer Connection	http://developer.apple.com/tools/macosxtools.html

2.3 Building library and samples

2.4.1 Unix

On Unix platforms, you can build the library and samples using the following steps. Use use the –enable-libxml2 option of the configure script instead of the compiler option. to use libxml2.

```
cd CyberLink
./boostrap
./configure
make
```

2.4.2 Windows

The CyberLink has the platform projects for Visual C++ 6.0. Please check the platform directories, CyberLinkC/*/win32/vc60, to use the projects. On WindowsCE, the CyberLink has no the platform projects, but a contributer have been checked to compile the source codes normally.

2.4.3 MacOSX

On MacOSX, you can buuld the library and samples using same steps of Unix platform or using Xcode or Project Bulider projects. Please check the platform directories to use the projects, CyberLinkC/*/macx.

2.4.4 T-Engine

On T-Engine platforms, you have to set the following compiler options. The CyberLink supports the process based and T-Kernel based program. Use PROCESS_BASE option to compile the process based program. Please see the development manual of your T-Engine development kit.

Option	URL
TENGINE	Enable the platform option.
TENGINE_NET_KASAGO	Enable KASAGO for T-Engine option..
USE_XMLPARSER_EXPAT	Use Expat as the XML parser

The CyberLink is compiled using the functions for PMC T-Shell Kit as the TCP/IP protocol stack, but it is no good because the protocol stack doesn't support the multicast protocol and the functions are not implemented yet.

To run applications using the CyberLink, the driver of the TCP/IP protocol stack has to be loaded and the network address has to be determined. Please see the manual of the protocol stack how to set the network interface.

You have to set EXPATROOT environment to an installed top directory of Expat on your shell as the following. The source codes of Expat have to be included the "lib" directory.

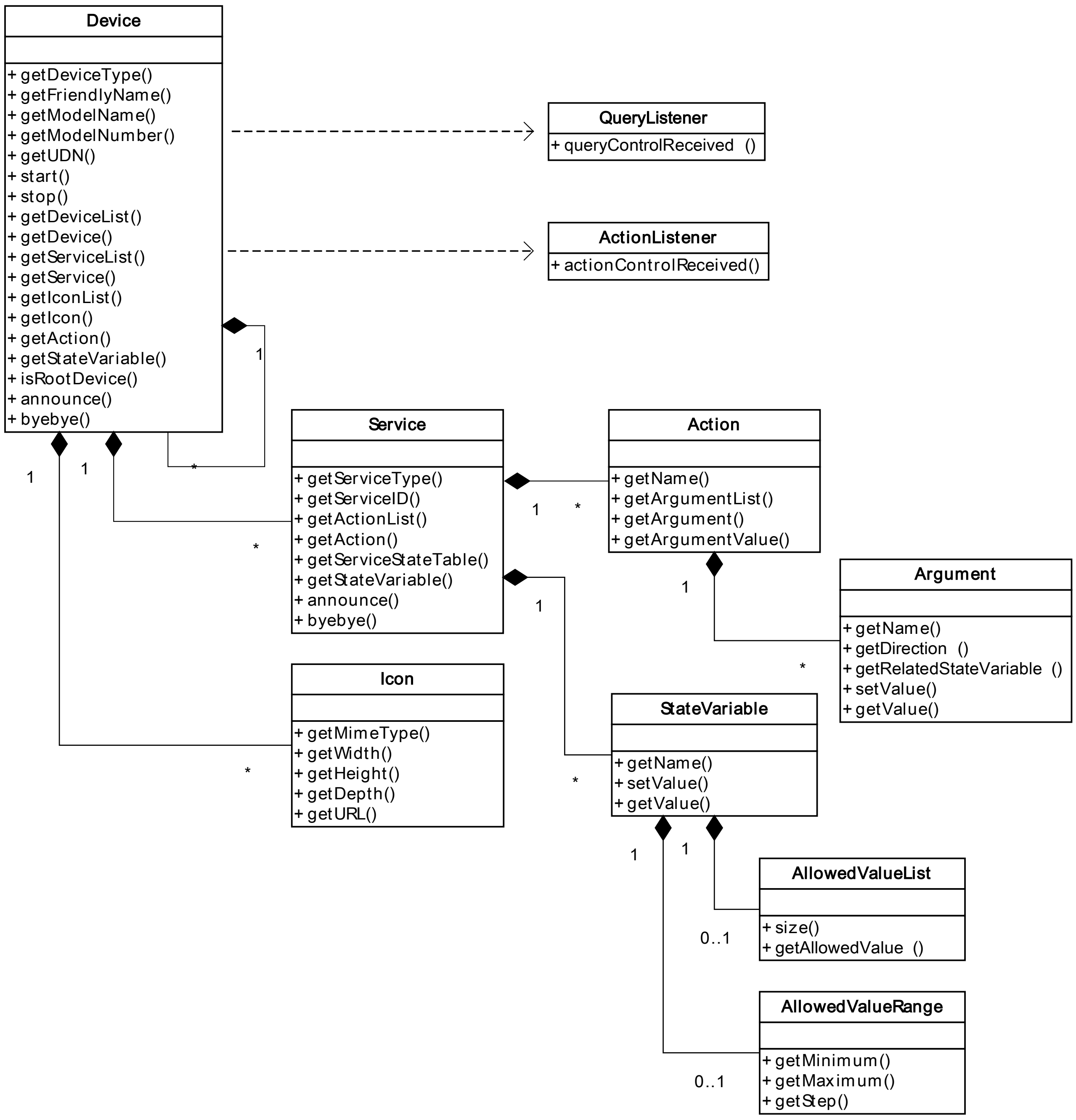
```
export EXPATROOT=/usr/local/expat-1.95.8
```

I have built the library with T-Engine/SH7727 development kit with KASAGO for T-Engine. Please check the platform directories, CyberLinkC/*/tengine/gnu , for the sample projects. To compile the samples, run configure script in the directory at first. Please see the development manual of your T-Engine development kit if you want to use on other T-Engine platforms.

3 Device

3.1 Class Overview

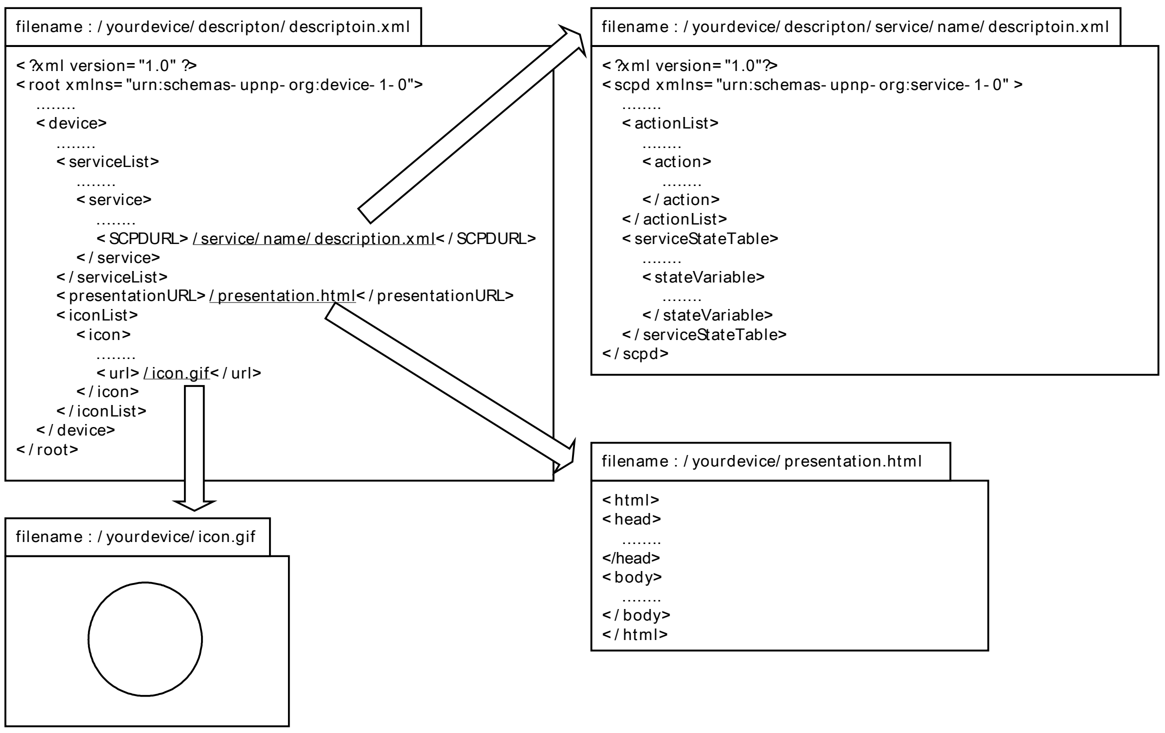
The following static structure diagram is related classes of CyberLink to create your device of UPnP™. The device has some embedded devices and services, and the services have some actions and state variables.



The above static structure diagram is modified simplify to explain.

3.2 Description

At first, you have to make some description files of your devices and the services when you want to create your UPnP™ device. The URLs in the device description should be relative locations from the directory of the device description file.



filename : /yourdevice/descripton/descriptoin.xml

```
<?xml version="1.0" ?>
<root xmlns="urn:schemas-upnp-org:device-1-0">
   ........
   <device>
      ........
      <serviceList>
         ........
         <service>
            ........
            <SCPDURL>/service/name/description.xml</SCPDURL>
         </service>
      </serviceList>
      <presentationURL>/presentation.html</presentationURL>
      <iconList>
         <icon>
            ........
            <url>/icon.gif</url>
         </icon>
      </iconList>
   </device>
</root>
```

filename : /yourdevice/descripton/service/name/descriptoin.xml

```
<?xml version="1.0"?>
<scpd xmlns="urn:schemas-upnp-org:service-1-0" >
   ........
   <actionList>
      ........
      <action>
         ........
      </action>
   </actionList>
   <serviceStateTable>
      ........
      <stateVariable>
         ........
      </stateVariable>
   </serviceStateTable>
</scpd>
```

filename : /yourdevice/presentation.html

```
<html>
<head>
   ........
</head>
<body>
   ........
</body>
</html>
```

filename : /yourdevice/icon.gif

The description of the root device should not have URLBase element because the element is added automatically when the device is created using the description.

The service descriptions are required to create a device, but the presentationURL and the iconList are recommended option. Please see UPnP™ specifications about the description format in more detail.

3.3 Initiating

To create a UPnP™ device, create a instance of Device class with the root description file. The created device is a root device, and only root device can be active using Device::start(). The device is announced to the UPnP™ network when the device is started. The following shows an example of the initiating device.

```
#include <cybergarage/upnp/CyberLink.h>
using namespace CyberLink;
......
const char *descriptionFileName = "description/description.xml";
Try {
    Device *upnpDev = new Device(descriptionFileName);
    ……
   upnpDev->start();
}
catch (InvalidDescriptionException e){
  const char *errMsg = e.getMessage();
 cout << “InvalidDescriptionException = ” << errMsg << endl;
}
```

The InvalidDescriptionException is occurred when the description is invalid. Use the getMessage() to know the exception reason in more detail.

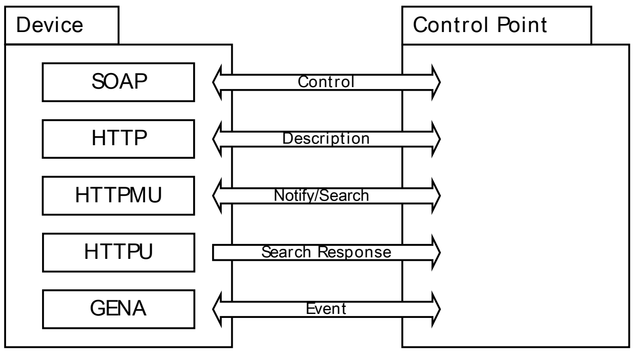
Alternatively, you can load the descriptions using Device::loadDescription() and Service::loadSCPD() instead of the description files as the following. The loading methods doesn't occur the exception.

```
     const char DEVICE_DESCRIPTION[] =
    "<?xml version=\"1.0\" ?>\n"
    "<root xmlns=\"urn:schemas-upnp-org:device-1-0\">\n"
    . . . .
     "</root>";

    const char SERVICE_DESCRIPTION[] =
     "<?xml version=\"1.0\"?>\n"
     "<scpd xmlns=\"urn:schemas-upnp-org:service-1-0\" >\n"
     . . . .
    “</scpd>”;

    Device *upnpDev = new Device();
     bool descSuccess = upnpDev->loadDescription(DEVICE_DESCRIPTION);
    Service *upnpService = getService("urn:schemas-upnp-org:service:****:1");
    bool scpdSuccess = upnpService->loadSCPD(SERVICE_DESCRIPTION[]);
```

The active root device has some server processes, and returns the responses automatically when a control points sends a request to the device. For example, the device has a HTTP server to return the description files when a control point gets the description file. The device searches an available port for the HTTP server automatically on the machine when the device is started.

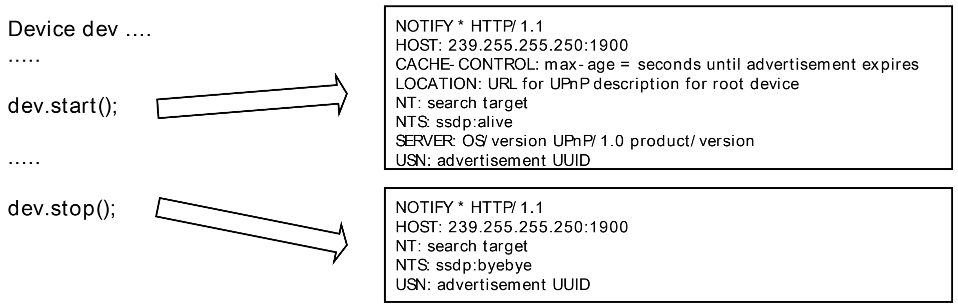


The root device is created with the following default parameters, you can change the parameters using the following methods before the root device is started.

	Parameter	Default	Method	Detail
1	HTTP port	4004	setHTTPPort()	The http server uses the port in the root device.
2	Description URI	/description.xml	setDescriptionURI()	The description URI of the root device.
3	Lease time	1800	setLeaseTime()	The lease time of the root device.

3.4 Notify

Your device is announced using Device::start() to the UPnP™ network using a notify message with ssdp::alive automatically when the device is started. When device is stopped using Device::stop(), a notify message is posted with ssdp::byebye. You can announce the notify messages using Device::announce() and Device::byebye().



```
Device dev ....
.....
dev.start();
.....
dev.stop();
```

```
NOTIFY * HTTP/1.1
HOST: 239.255.255.250:1900
CACHE-CONTROL: max-age = seconds until advertisement expires
LOCATION: URL for UPnP description for root device
NT: search target
NTS: ssdp:alive
SERVER: OS/version UPnP/1.0 product/version
USN: advertisement UUID
```

```
NOTIFY * HTTP/1.1
HOST: 239.255.255.250:1900
NT: search target
NTS: ssdp:byebye
USN: advertisement UUID
```

When a control point sends a search request with M-SEARCH to the UPnP™ network, the active device send the search response to the control point automatically. The device repeats the announcement in the lease time automatically.

3.5 Embedded Devices

The devices may have some embedded devices. Use Device::getDeviceList() to get the embedded device list. The following example outputs friendly names of all embedded devices in the device.

```
void printDevice(Device *dev)
 {
      counst char *devName = dev->getFriendlyName();
     cout << devName << endl;

    DeviceList *childDevList = dev->getDeviceList();
   int nChildDevs = childDevList->size();
   for (int  n  =0;  n  <nChildDevs;  n  ++) {
        Device *childDev = childDevList ->getDevice(n);
        printDevice(childDev);
   }
 }
 ......
 Dev *rootDev = ....;
 ......
 DeviceList *childDevList = rootDev->getDeviceList();
  int childDevs = childDevList->size();
for (int n=0; n< childDevs; n++) {
    Device *childDev = rootDevList->getDevice(n);
    printDevice(childDev);
 }
```

You can find a embedded device by the friendly name or UDN using Device::getDevice(). The following example gets a embedded device by the friendly name.

```
 Device *homeServerDev ....
Device *musicDev = homeServerDev->getDevice("music");
```

3.6 Service

Use Device::getServiceList() to access embedded services of the device. The service may has some actions and state variables. Use Service::getActionList() to get the actions, and use Service::getServiceStateTable() to the state variables by the name. The following example outputs the all actions and state variables in a device.

```
 Device *dev ....
 ServiceList *serviceList = dev->getServiceList();
int serviceCnt = serviceList->size();
 for (int n=0; n<serviceCnt; n++) {
        Service *service = serviceList->getService(n);
         ActionList *actionList = service->getActionList();
         int actionCnt = actionList->size();
        for (int i=0; i<actionCnt; i++) {
         Action *action = actionList->getAction(i);
    cout << "action [" << i << "] = "<< action->getName() << endl;
         }
        ServiceStateTable *stateTable = service-> getServiceStateTable ();

        int varCnt = stateTable->size();
        for (int i=0; i<actionCnt; i++) {
             StateVariable *stateVar = stateTable->getServiceStateVariable(i);
             cout << "stateVar [" << i << "] = " << stateVar->getName() << endl;
         }
 }
```

You can find a service in the device by the service ID using Device::getService(), and you can find an action or state variable in the service by name too. Use Device::getAction() or Service::getAction() to find the action, and use Device::getStateariable() or Service::getStateVariable() to find the state variable. The following example gets a service, a action and a state variable in a device by name.

```
Device *clockDev ....
 Service *timerSev = clockDev->getService("timer");
 Action *getTimeAct = clockDev->getAction("GetTime");
StateVariable *timeStat = clockDev->getStateVariable("time");
```

3.7 Control

To receive action control events from control points, the device needs to implement the ActionListener interface. The listener have to implement a actionControlReceived() that has the action and argument list parameter. The input arguments has the passed values from the control point, set the response values in the output arguments and return true when the request is valid. Otherwise return a false when the request is invalid. UPnPError response is returned to the control point automatically when the returned value is false or the device has no the interface. The UPnPError is INVALID_ACTION as default, but use Action::setSetStatus() to return other UPnP errors.

To receive query control events from control points, the device needs to implement the QueryListener interface.

The listener have to implement a queryControlReceived() that has the service variable parameter and return a true when the request is valid. Otherwise return a false when the request is invalid. UPnPError response is returned to the control point automatically when the returned value is false or the device has no the interface. The UPnPError is INVALID_ACTION as default, but use ServiceVariable::setSetStatus() to return other UPnP errors.

The following sample is a clock device. The device executes two action control requests and a query control request.

```
class ClockDevice : public Device, public ActionListener, public QueryListener
{
 public:
  ClockDevice() : Device(“/clock/www/description.xml”)
 {
   Action *setTimeAction = getAction(“SetTime”);
    setTimeAction->setActionListener(this);
   Action *getTimeAction = getAction(“GetTime”);
    getTimeAction->setActionListener(this);
    StateVariable *timeVar = getStateVariable(“Timer”);
  timeVar->setQueryListener(this);
 }

    bool actionControlRecieved(Action *action)
 {
         ArgumentList *argList = action->getArgumentList();
        const char *actionName = action->getName();
        if (strcmp(actionName, "SetTime") == 0 {
            Argument *inTime = argList->getArgument(“time”);
            const char *timeValue = inTime->getValue();
              If (timeValue == NULL || strlen(timeValue) <= 0)
               return false;
            ……..
             Argument *outResult = argList->getArgument(“result”);
              arg->setValue(“TRUE”);
            return true;
    }
   else if (strcmp(actionName, “GetTime”) == 0) {
              const char *currTimeStr = …..
    Argument *currTimeArg = argList->getArgument(“currTime”);
            currTimeArg->setValue(currTimeStrs);
            return true;
    }
      action->setStatus(UPnP::INVALID_ACTION, "…..");
          return false;
    }

    bool queryControlReceived(StatusVariable *stateVar)
    {
     if (strcmp(varName, "Time") == 0) {
      cont char *currTimeStr = ….;
      stateVar.setValue(currTimeStr);
      return true;
      }
     stateVar->setStatus(UPnP::INVALID_VAR, "…..");
     return false;
    }
   }
```

Use Device::setActionListner() or Service::setActionListnerer() to add a listener into all control actions in a device or a service. Use Device::setQueryListner() or Service::setQueryListner() to add a listener into all query actions in a device or a service. The following sample sets a listener into all actions in a device.

```
   class ClockDevice : public Device, public ActionListener, public QueryListener
   {
    public:
     ClockDevice() : Device("/clock/www/description.xml")
    {
          setActionListner(this);
         setQueryListener (this);
    }
      bool actionControlRecieved(Action *action) { ……. }
   bool queryControlReceived(StateVariable *stateVar) { ……. }
   }
```

3.8 Event

The control point may subscribe some events of the device. You don't need manage the subscription messages from control points because the device manages the subscription messages automatically. For example, the device adds a control point into the subscriber list when the control point sends a subscription message to the device, or the device removes the control point from the subscriber list when the control point sends a unsubscription message.

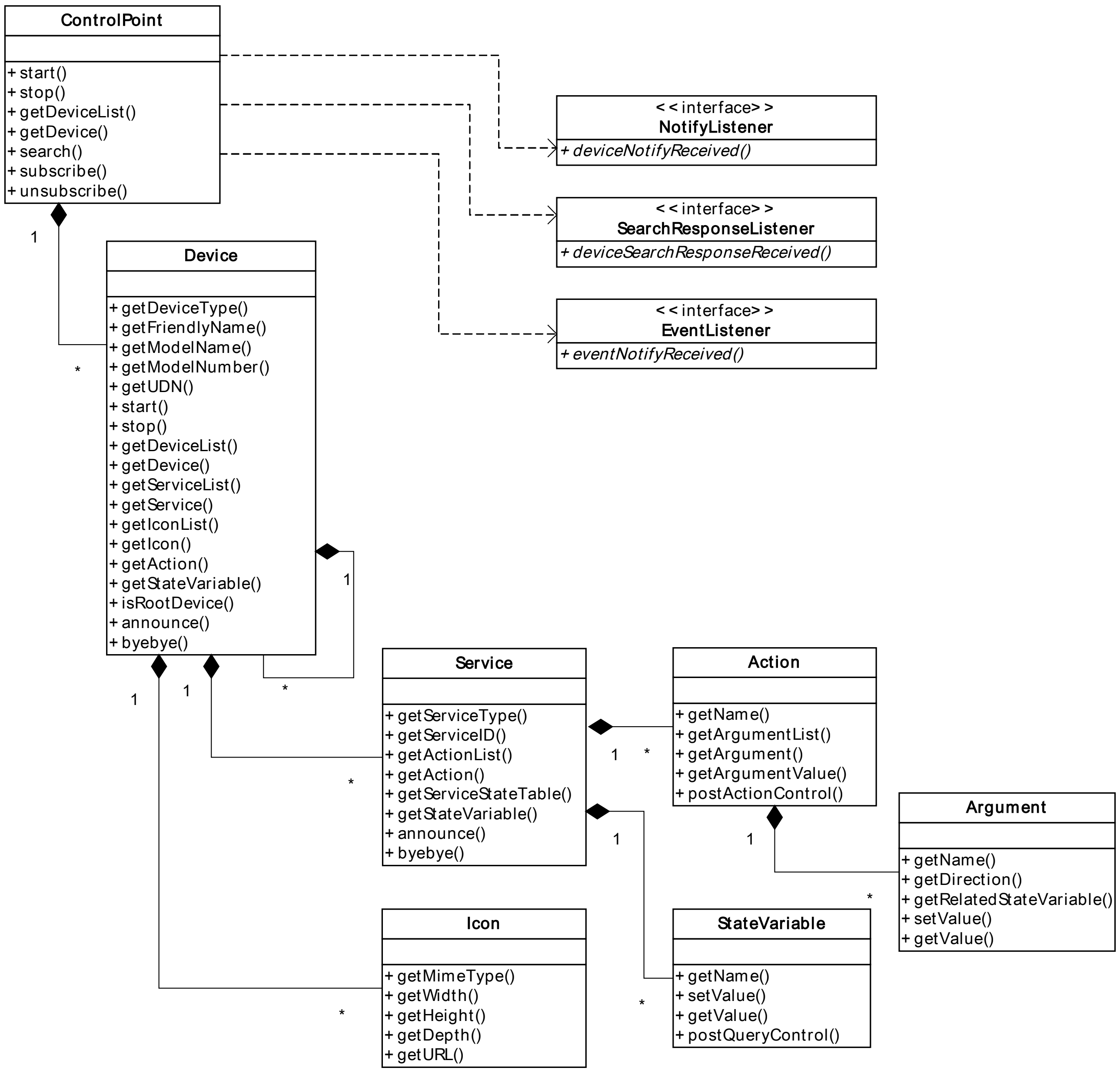
Use ServiceStateVariable::setValue() when you want to send the state to the subscribers. The event is sent to the subscribers automatically when the state variable is updated using ServiceStateVariable::setValue(). The following example updates a state variable, and the updated state is distributed to the subscribers automatically.

```
Device *clockDevice = ....
StateVariable timeVar = clockDevice->getStateVariable("Time");
 const char *timeStr = .....
 timeVar->setValue(timeStr);
```

4 Control Point

4.1 Class Overview

The following static structure diagram is related classes of CyberLink to create your control point of UPnP™. The control point has some root devices in the UPnP™ network.



4.2 Initiating

To create a UPnP™ control point, create a instance of ControlPoint class. Use ControlPoint::start() to active the contol point. The control point multicasts a discovery message searching for all devices to the UPnP™ network automatically when the control point is active.

```
#include <cybergarage/upnp/CyberLink.h>
using namespace CyberLink;
......
ControlPoint *ctrlPoint = new ControlPoint();
......
ctrlPoint->start();
```

The active control point has some server processes, and returns the responses automatically when other UPnP™ devices send the messages to the control point. For example, the control point has a SSDP server to get M-SEARCH responses, and the control point searches a available port for the SSDP server automatically on the machine when the control point is started.

The control point is created with the following default parameters. You can change the parameters using the following methods before the control point is started.

	Parameter	Default	Method	Detail
1	HTTP port	8058	setHTTPPort()	The port is used to receive subscription events.
2	SSDP port	8008	setSSDPPort()	The port is used to receive search responses.
3	Subscription URI	/eventSub	setEventSubURI()	The URI is used to receive subscription events.
4	Search Response	3	setSerchMx()	Maximum wait for device searching

4.3 Notify

The control point receives notify events from devices in the UPnP™ network, and the devices are added or removed form the control point automatically. The expired device is removed from the device list of the control point automatically too. You don’t manage the notify events, but you can receive the event to implement the NotifyListener interface. The following sample receives the notify messages.

```
class MyCtrlPoint : public ControlPoint, public NotifyListener
{
 public:
 MyCtrlPoint()
    {
      ........
       addNotifyListener(this);
```

```
            start();
        }
        void deviceNotifyReceived(SSDPPacket *ssdpPacket)
        {
          string use, nt, nts, location;
          const char *uuid = ssdpPacket.getUSN(usn);
            const char *target = ssdpPacket.getNT(nt);
            const char *subType = ssdpPacket.getNTS(nts);
            const char *where = ssdpPacket.getLocation(location);
        .......
        }
```

To know only the added and removed device, you may use the following interface, DeviceChangeListener.

```
    class MyCtrlPoint : public ControlPoint, public DeviceChangeListener
    {
     public:
    MyCtrlPoint()
        {
          ........
            addDeviceChangeListener (this);
            start();
        }
        void deviceAdded (Device *dev)
        {
          ……..
        }
        void deviceRemoved(Device *dev)
        {
          ……..
        }
 }
```

4.4 Search

You can update the device lists using ControlPoint::search(). The discovered root devices are added to the control point automatically, and you can receive the response to implement the SearchResponseListener interface. The following sample receives the notify messages.

```
    class MyCtrlPoint : public ControlPoint, public SearchResponseListener
```

```
{
 public:
MyCtrlPoint()
   {
     ........
      addSearchResponseListener(this);
       start();
     ........
    search("upnp:rootdevice");
   }
   void deviceSearchResponseReceived(SSDPPacket *ssdpPacket)
   {
     string usn, st, location;
       const char *uuid = ssdpPacket.getUSN(usn);
      const char *target = ssdpPacket.getST(st);
      const char *where = ssdpPacket.getLocation(location);
  ........
 }
```

4.5 Root Devices

Use ControlPoint::getDeviceList() that returns only root devices to get the discovered device list. The following example outputs friendly names of the root devices.

```
ControlPoint *ctrlPoint = new ControlPoint();
......
ctrlPoint->start();
......
DeviceList *rootDevList = ctrlPoint->getDeviceList();
int nRootDevs = rootDevList->size();
for (int n=0; n<nRootDevs; n++) {
   Device *dev = rootDevList->getDevice(n);
    const char *devName = dev->getFriendlyName();
    cout << "[" << n << "] = " << devName << endl;
}
```

You can find a root device by the friendly name, the device type, or the UDN using ControlPoint::getDevice(). The following example gets a root device by the friendly name.

```
ControlPoint *ctrlPoint = new ControlPoint();
```

```
......
ctrlPoint->start();
......
Device *homeServerDev = ctrlPoint->getDevice(“xxxx-home-server”);
```

4.6 Control

The control point can send action or query control messages to the discovered devices. To send the action control message, use Action::setArgumentValue() and Action::postControlAction (). You should set the action values to the all input arguments, and the output argument values is ignored if you set. The following sample posts a action control request that sets a new time, and output the response result.

```
Device *clockDev = ....
Action *setTimeAct = clockDev->getAction(“SetTime”);
char *newTime = ....
setTimeAct->setArgumentValue(“time”, newTime); // setTimeAct->getArgument(“time”)->setValue(newTime);
if (setTimeAct->postControlAction() == true) {
    ArgumentList *outArgList = setTimeAct->getOutputArgumentList();
    int nOutArgs = outArgList->size();
    for (int n=0; n<nOutArgs; n++) {
        Argument *outArg = outArgList->getArgument(n);
        const char *name = outArg->getName();
        const char *value = outArg->getValue();
        ......
    }
}
else {
  UPnPStatus *err = setTimeAct->getUPnPStatus();
  System.out.println("Error Code = " + err->getCode());
  System.out.println("Error Desc = " + err->getDescription());
}
```

To send the query control message, use StateVariable::postQueryControl(). The following sample posts a query control request, and output the return value.

```
Device *clockDev = ....
StateVariable *timeStateVar = clockDev->getStateVariable(“time”);
if (timeStateVar->postQueryControl() == true) {
  String value = timeStateVar.getValue();
    ......
```

```
 }
else {
 UPnPStatus *err = timeStateVar->getUPnPStatus();
   System.out.println("Error Code = " + err->getCode());
   System.out.println("Error Desc = " + err->getDescription());
 }
```

Use Argument::getRelatedStateVariable() to get the related StatiVariable of the argument, and use StateVariable:: getAllowedValueRange() or getAllowedValueList() to get the the allowed value range or list.

```
Device *clockDev = ....
 Action *timeAct = clockDev->getAction("SetTime");
Argument *timeArg = timeAct->getArgument("time");
 StataVariable *stateVar = timeArg->getRelatedStateVariable();
if (stateVar != NULL) {
 if (stateVar->hasAllowedValueRange() == true) {
   AllowedValueRange *valRange = stateVar->getAllowedValueRange();
    ......
 }
 if (stateVar->hasAllowedValueList() == true) {
   AllowedValueList *valList = stateVar->getAllowedValueList ();
    ......
 }
}
```

4.7 Event

The control point can subscribe events of the discovered devices, get the state changes of the services Use ControlPoint::subscribe() and implement the EventListener interface. The listener have to implement a eventNotifyReceived().

```
MyControlPoint : public ControlPoint, public EventListener
{
 public:
  MyControlPoint()
 {
  .....
  addEventListener(this);
 }
 .....
```

```
  void eventNotifyReceived(const char *uuid, long seq, const char *name, const char *value)
  {
      ....
  }
}
```

The ControlPoint::subscribe() returns true when the subscription is accepted from the service, and you can get the subscription id and timeout.

```
ControlPoint *ctrlPoint = .....
Device *clockDev = ctrlPoint->getDevice("xxxx-clock");
Service *timeService = clockDev->getService("time:1");
bool subRet = ctrlPoint->subscribe(timeService);
if (subRet == true) {
   const char *sid = timeService->getSID();
  long timeout = timeService->getTimeout();
}
```

5 Networked Media Product Requirements

The Intel Networked Media Product Requirements (NMPR) is a implementation guidelins for digital networked devices. Please see the following page about NMPR in more detail.

Home	URL
Intel Develper Network for Digital Home	http://www.intel.com/technology/dhdevnet/

5.1 NMPR Mode

The NMPR features are include Device and ControlPoint class of the CyberLink, but the features are not available as default. Please use Device::setNMPRMode() and ControlPoint::setNMPRMode() as the following.

```
ControlPoint *ctrlPoint = .....
ctrlPoint->setNMPRMode(true);
ctrlPoint->start();

Device *device = …
device->setNMPRMode(true);
device->setWirelessMode(true); // if your device is on wireless network.
device->start().;
```

5.2 Implementaion Status

UPnP Technology Requirements	M/S/O	Support	Default	NMPR	Detail
<Device Discovery and Control>	M	Y	O	O	The NMPR tag is added automatically when the NMPR mode is true.
<UPnP iNMPR'03 Description Tag>	M	Y	X	O	
<UPnP Discovery Flood Control>	M	Y	O	O	
<UPnP SSDP Default Port>	M	Y	O	O	
<UPnP Device Advertisement Frequency>	M	Y	O	O	Use Device::setWirelessMode() too.
<UPnP Discovery over Wireless>	M	Y	X	O	The subscription is update automatically when the NMPR mode is true.
<UPnP Device Sync Rules>	M	Y	X	O	Depend on your platform and device implemenation.
<UPnP Auto-IP Support>	M	-	-	-	Not support to the dynamic changes in network configuration.
<UPnP Device Reset>	M	N	X	X	The inavtivity connection close and HTTP HEAD request are not supported yet.
<UPnP HTTP Support and General Rules>	M	N	X	X	
<UPnP HTTP1.0 Rules>	M	Y	O	O	Only HTTP1.0 is supported.
<UPnP HTTP 1.1 Rules>	M	N	X	X	The HTTP pipelining is not supported yet.
<UPnP HTTP Pipelining General Rules>	M	N	X	X	
<UPnP Unknown Tag/Field Decoding>	M	Y	O	O	
<UPnP Error Codes>	M	-	-	-	Depend on your device implementation.

<UPnP Device Responsiveness>	M	-	-	-	Depend on your device implementation.
<UPnP SOAP Packet Size>	S	Y	O	O	There is no limitation for the embedded devices.
<UPnP Embedded Device Support>	M	N	X	X	
<UPnP Multi-Homing Rules>	M	Y	O	O	Depend on your device implementation. The UPnP errors are returned automatically when the services are defined and not implemented.
<UPnP Service Descriptions Rules>	M	-	-	-	
<UPnP Subscription Handling>	M	Y	O	O	
<UPnP Event Notification Handling>	M	Y	O	O	Depend on your device implementation.
<UPnP Device Icons>	M	-	-	-	Depend on your device implementation.
<UPnP PNG Icon Recommendations>	M	-	-	-	Depend on your device implementation.
<UPnP Device Icon Support for .ICO>	S	-	-	-	
<UPnP UTF-8 Support>	M	Y	O	O	
<UPnP XML Comments>	M	Y	O	O	Depend on your device implementation.
<UPnP Boolean Type>	M	-	-	-	Depend on your device implementation. To not implement the QuaryStateVariable, don't use Device::setQueryListner.
< UPnP QueryStateVariable>	M	-	-	-	
< UPnP Action Arguments Encoding>	M	Y	O	O	The BaseURL is removed only when the NMPR mode is true. The service descriptions must be relative in the device description.
<UPnP URI Rules>	M	Y	O/X	O	There is no limitation for he UDN length. To change the device descriptions and the services, create the new device and start it.
<UPnP UDN Usage>	M	Y	O	O	The control point is not renew the subscriptions automatically.
<UPnP Subscription Renewals>	O	Y	X	O	
<UPnP Device Handling of Subscription Renewals>	M	Y	O	O	
<UPnP IP Address Rules>	M	Y	O	O	

6 XML Parser

CyberLink supports the following XML parsers to read device descriptions or execute actions of UPnP.

XML Parser	URL	Compiler Option
Apache Xerces	http://xml.apache.org/xerces-c/	
Expat	http://expat.sourceforge.net/	USE_XMLPARSER_EXPAT
libxml2	http://xmlsoft.org/	USE_XMLPARSER_LIBXML2

CyberLink uses Xerces as the default XML parser. Use the compiler options to your compiler to use Expat or libxml2 instead of the Xerces.

On Unix platform, use –enable-expat, or –enable-libxml2 to the configure script instead of the compiler options as the following.

```
./configure --enable-expat
```

7 IPv6

CyberLink binds all interfaces in the platform when the devices or control points are created, and the IPv6 sockets are created automatically if the interfaces have IPv6 addresses.

CyberLink supports IPv4 and IPv6 both as default. If you want to use only IPv6 interfaces, call the following method before the devices or control points are created.

```
UPnP::SetEnable(UPnP::USE_ONLY_IPV6_ADDR)
```

Link local is the default scope for multicasting of CyberLink. Use UPnP::SetEnable() to change the scope. The following example changes the scope to the site local.

```
UPnP::SetEnable(UPnP::USE_IPV6_SITE_LOCAL_SCOPE)
```

On Unix platform, CyberLink get the local interfaces IPv4 and IPv6 using getifaddrs(). If you want to use CyberLink with IPv6, please check the function supports IPv6 on your platform. I know that the function doesn't support IPv6 yet on Redhat 9 and MacOSX 10.2.6.

8 Inside CyberLink

8.1 Overriding HTTP Service

The Device class of CyberLink implements a HTTPRequestListner interface of CyberHTTP package to handle some HTTP requests from the control points. The HTTPRequestListener interface is bellow.

```
class HTTPRequestListener
{
public:
        virtual void httpRequestRecieved(HTTPRequest *httpReq) = 0;
};
```

To overide the interface, use CyberHTTP namespace and override the httpRequestRecieved method in your device that is a sub class of the Device class. The following example is a clock device using CyberLink, and adds the override method to return the presentation page.

```
using namespace CyberHTTP;
const char PRESENTATION_URI[] = "/presentation";
......
class ClockDevice : public Device, public ActionListener, public QueryListener
{
    ......
    void httpRequestRecieved(HTTPRequest *httpReq) {
        string uri = httpReq->getURI();
        if (uri.find(PRESENTATION_URI) == string::npos)  {
            Device::httpRequestRecieved(httpReq);
            return;
        }
        string clockStr = .......;
        string contents;
        contents = "<HTML><BODY><H1>";
        contents += clockStr;
        contents += "</H1></BODY></HTML>";
        HTTPResponse httpRes;
        httpRes.setStatusCode(HTTP::OK_REQUEST);
        httpRes.setContent(contents);
        httpReq->post(&httpRes);
    }
```

```
        }
```

9 Transitioning From Version 1.2

9.1 QueryListner

CyberLink v1.3 has changed the QueryListner interface to return user error infomation and set the listener to the StateVariable instance instead of the Service instance. The difference is bellow.

Version	Interface
1.2	bool queryControlReceived(Service *service, const char *varName, std::string &result)
1.3	bool queryControlReceived(StateVariable *stateVar)

Use StateVariable::getName() to know the variable name, and use StateVariable::setValue() to return the result value. The following sample shows the difference between v1.2 and v1.3.

```
v1.2:
 bool queryControlReceived(Service *service, const char *varName, string &result) {
        result = Clock::GetCurrentTimeString();
        return true;
 }

v1.3:
 bool queryControlReceived(StateVariable *stateVar) {
        const char *varName = stateVar->getName();
        stateVar->setValue(Clock::GetCurrentTimeString());
        return true;
 }
```

To set the query listener, use StateVariable:: setQueryListener() instead of Service::setQueryListner(). However, Service::setQueryListner() that sets the specifed listener to all state variables in the service is not deprecated to ensure the compatibility. The implematation is bellow.

```
void Service::setQueryListener(QueryListener *listener)
{
        ServiceStateTable *stateTable = getServiceStateTable();
        for (int n=0; n< stateTable->size(); n++)
                StateVariable *var = stateTable->getStateVariable(n);
                var->setQueryListener(listener);
        }
}
```

10 ChangeLog

v.1.7 2005-05-29

* Changed some headers to compile normally on cygwin and mingw.
* Added support for Expat XML parser.
* Added support for HTTP chunked stream to CyberHTTP.
* Fixed StateVariable::setValue() to set blank value when a null value is specified.
* Fixed XML::EscapeXMLChars() to convert from """ to "&apos;" instead of "\".
* Changed Socket::send() to retry when the packet is not sent normally.
* Changed StateVariable::getAllowedValueList() and AllowedValueList::getAllowedValue() to use AllowedValue instead of std::string as the member.
* Fixed Device::getNotifyDeviceNT() to return the UDN when the device is not root device.
* Changed httpPostRequestRecieved() to return the bad request when the post request isn't the SOAP action.
* Added Device::loadDescription(const char *) to load the description from memory.
* Added Service::loadSCPD() to load the description from memory.
* Added a exclusive access control to Device::getDescriptionData().
* Removed Service::setDescriptionURL() and Service::getDescriptionURL().
* Changed httpGetRequestRecieved() to return the description stream using Device::getDescriptionData() and Service::getSCPDData() at first
* Changed Thread::start() and stop() to virtual method for overriding in the sub class.
* Changed Device::deviceSearchResponse() answer with USN:UDN::<device-type> when request ST is device type.
* Changed Device::getDescriptionData() and Service::getSCPDData() to add a XML declaration at first line.
* Added a new Device::setActionListener() and serQueryListner() to include the sub devices.
* Changed ActionRequest::getActionName() to return when the delimiter is not found.
* Added support for gcc 4.0.

v.1.6	2004-11-1
* Added support for HTTP v1.1 connection. * Added support for HEAD and Content-Range headers. * Fixed ControlPoint::removeExpiredDevices() to remove using the device array. * Fixed HTTPRequest::getHeader() and getHTTPVersion() no to return "HTTP/HTTP/version" * Added HTTPRequest::isHeadRequest(). * Added HTTPPacket::setContentRange() and getContentRange(). * Changed HTTPRequest::post(HTTPResponse*) not to close the connection. * Changed HTTPRequest::post(const char *, int) to add a connection header to close. * Added a isOnlyHeader parameter to HTTPSocket::post(). * Added a updateWithContentLength parameter to HTTPPacket::setContent(). * Changed to HTTPPacket::set() not to change the header of Content-Length. * Changed HTTPServer::accept() to set a default timeout, HTTP.DEFAULT_TIMEOUT, to the socket. * Added SocketImp::setTimeout(). * Added HTTPPacket::init() and read(). * Added HTTPRequest::isKeepAlive(). * Added skip() to IntputStream, SocketInputStrean and FileInputStream. * Added HTTPStatus::isSuccessful() and HTTPRequest::isSuccessfule(). * Added some patches for FreeBSD. * Removed a SOAP header from DIDLite::output() * Added support for Range header to HTTPPacket::getContentRange(). * Added a MYNAME header to SSDP messages. * Changed socket functions to ignore SIGPIPE signal * Added a fix to HTTPRequest::post() when the last position of Content-Range header is 0. * Added a Content-Range header to the response in HTTPRequest::post(). * Changed the status code for the Content-Range request in HTTPRequest::post(). * Added to check the range of Content-Range request in HTTPRequest::post(). * Added support the intinite time and fixed a bug in Subscriber::isExpired(). * Fixed a bug when Service::notify() removes the expired devices() * Fixed SSDPPacket::isRootDevice() to check the ST header. * Added Device::getStateVariable(serviceType, name). * Changed XML::output() to use short notation when the tag value is null. * Changed SSDP message to set the TTL as four. * Changed StateVariable::setValue() to update and send the event when the value is not equal to the current value.	

v.1.5	2004-08-10
* Added support for Intel NMPR. * Added Device::setNMPRMode() and isNMPRMode(). * Changed to advertise every 25%-50% of the periodic notification cycle for NMPR. * Added Device::setWirelessMode() and isWirelessMode(). * Changed Device::start() to send a bye-bye before the announce for NMPR. * Changed Device::annouce(), byebye() and deviceSearchReceived() to send the SSDP messsage four times when the NMPR and the Wireless mode are true. * Fixed Device::announce() and byebye() to send the upnp::rootdevice message despite embedded devices. * Fixed Device::getRootNode() to return the root node when the device is embedded. * Fixed Service::announce() to set the root device URL to the LOCATION field. * Added ControlPoint::renewSubscriberService(). * Changed ControlPoint::start() to create renew subscriber thread when the NMPR mode is true. * Changed Action::postControlAction() to set the status code to the UPnPStatus. * Changed StateVariable::postQuerylAction() to set the status code to the UPnPStatus. * Added Device::getParentDevice(); * Changed Service::notify() to remove the expired subscribers and not to remove the invalid response subscribers for NMPR.	
v1.4.1	2004-06-13
* Added a sample for AV MediaServer, CyberMediaGate. * Changed HTTPRequest::post(HTTPResponse *) to close the socket stream from the server. * Changed HTTPPacket::getHeader() to compare using string::equals(). * Added I18N support for SORPRequest and SOAPResponse classes. * Fixed invalide SERVER field of SSDP and HTTP response on Unix Plathome. * Changed SubscriptionRequest::setServie() to get the host address from the SSDP Location field when the URLBase is null.	

v1.4	2004-05-12
* Fixed to set HTTP v1.1 to SSDPRequest and SSDPResponse class. * Fixed setRequestHost() for Sony's UPnP stack when the URLBase has the path. * Changed to update the UDN only when the field is null. * Fixed Service::getDeviceList() to public method. * Fixed URI::getProtocol() to retuen the name without "://". * Added some methods for StateVariable to get the AlloedValueList and the AllowedValueRange. * Changed the upnpdump sample to pring all descovered devices. * Fixed Praser::parse() to read under the buffer size. * Fixed Subscription::GetTimeout() to return the valid value. * Fixed Subscription::toTimeoutHeaderString() to set the valid value instead of the invalid URL value. * Fixed to initialize members of ActionData such as the action listener and the control response. * Added start(const char *target, int mx) and start(const char *target) to ControlPoint. * Added Device::isDeviceType(). * Fixed Argument::initArgumentList() to share a argument node in three Argument when the argument lists are initialized. * Fixed Argument::getArgument(name) to return the valid pointer. * Added StateVariable::hasAllowedValueList() and hasAllowedValueRange().	
v1.3.2	2004-03-11
* Fixed a HTTP server bug to return v1.1 instead of v1.0. * Added a null line to SSDP messages for Intel UPnP tools. * Added a header of xml version to setConetent() in SOAPRequest and SOPAResponse. * Changed to use the control URL when it is absolute. * Changed to get URLBase from the SSDP packet when the URLBase is null and the control URL is not absolute. * Fixed a interval time of the advertiser on Unix Platform. * Added Xcode projects on MacOSX.	
v1.3	2004-01-13
* Improved the HTTP server using multithreading. * Updated a udn field by a generated uuid automatically. * Added Device::setActionListener() and Device::setQueryListener(). * Added Service::setActionListener(). * Added a Content-Length header to subscription responces. * Added Action::setAgumentValue(String name, int value). * Added Agument::setValue(int value). * Added Action::getAgumentIntegerValue(). * Added Agument::getIntegerValue(). * Added automatic device advertiser * Added automatic device disposer that remove expired devices from the control point. * Changed action listener and query listener to return user error messages.	
v1.0	2003-10-02
The first release.	

11 License