OpenHome
ohNet Device Stack

www.openhome.org

Document revision date 07 July 2011

Table of Contents

Introduction ... 3

Prerequisites ... 3

Related documents ... 3

Why choose ohNet? ... 4

Device stack ... 5

Initializing the Device stack ... 5

Closing the Device stack ... 5

Device ... 6

Creating the Device ... 6

Setting the Device's attributes ... 6

Enabling the Device ... 7

Altering the Device ... 7

Provider ... 8

Actions ... 8

Properties ... 9

Updating related properties ... 9

Creating a Provider ... 10

Appendix ... 11

ohNetGen ... 12

Prerequisites ... 12

ohNetGen arguments ... 12

Introduction

This document is for developers using the OpenHome ohNet stack to publish a device's services over a local network.

ohNet is a full UPnP stack. The public APIs are protocol independent and offered in C++, C# and C.

Note

All code examples in this document use C++. Each C++ function used has an equivalent in the other languages. The naming scheme for the functions is consistent between the languages so you can easily find the functions in the other languages published in the API.

All snippets also assume the use of two namespaces: `OpenHome` and `OpenHome::Net`.

The Device stack exposes the parts of ohNet required to present your device's services to interested clients, such as Control Points.

Specific emphasis is placed on the use of Providers to publish services, giving access to the actions and properties that define each service. To help you write your Provider, ohNet comes supplied with the tools needed to automatically generate code from the service XML. This document includes details on how to use the generation tool and a detailed walk-through of how to write a Provider using example code.

Note

You must have detailed knowledge of how the services are implemented on your device. The code generated for the Provider gives you a framework which you can add to. You must add the code in the Provider to fully implement each service's action.

Prerequisites

To begin development of your device Provider, you must be in possession of the ohNet code.

The ohNet code includes all OpenHome and UPnP AV services. However, if you are using a different service you must also have your service's XML description file. You will use this to generate your own code.

See *ohNetGen on page 12* in the Appendix for more details of using your service XML to generate a Provider.

Related documents

The following related ohNet documentation may be of interest to you:

- *OpenHome Overview*
- *OpenHome ohNet Control Point Stack*

Why choose ohNet?

The ohNet Device stack provides developers with several benefits to quickly and efficiently generate code required to access devices.

The benefits to using the ohNet Device stack are:

- **Multiple API support** — ohNet is developed in C++ but the API is published in C++, C and C# to cover a wide range of developer experience and preference.
- **Open source** — OpenHome is committed to the continued open development of the UPnP protocol and provides the Control Point stack completely free and open under a BSD-style license.
- **Rapid development** — ohNet provides developers with tools to automatically generate code from the associated XML files, removing the possibility of error prone code and automating an otherwise manual process. Generated code presents services as classes with consistent and intuitive naming conventions. Updates to your device's state are automatically pushed out to registered subscribers. No extra work is required to keep Control Points and users up-to-date with live updates.
- **Readability** — APIs use function and variable names generated from the action and property names contained in the original service XML for maximum readability. New code generated from new service XML is also intelligently named following this model.

Device stack

The Device stack has been designed specifically for use by software that offers services on a local network. It provides the mechanism used to receive and respond to messages requesting a change in a physical device's state.

The Device stack uses a class-based architecture to present the device and its services on the network. The device is implemented in a Device class. The device's services are implemented in Provider classes.

You use Devices and Providers together with the Device stack to send updates about the device's state to interested clients, such as Control Points.

Details on how you use the Device stack and the Device and Provider classes are contained in the following sections.

Initializing the Device stack

You must initialize the Device stack before you can use any of the other ohNet code.

To initialize the Device stack, include the following three lines of code:

```
InitialisationParams* initParams = new InitialisationParams();
// initParams defaults to useful values for all fields but you can optionally
change any values here
UpnpLibrary::Initialise(initParams);
UpnpLibrary::StartDv();
```

See the ohNet API's listing of `UpnpLibrary` for more details on initializing the stack.

Closing the Device stack

The final thing you must do with the Device stack is close it when you have finished using it.

To close the Device stack you must include the following single line of code:

```
UpnpLibrary::Close();
```

You cannot make any further library calls after you call this function.

Device

A Device offers 1..n services over a network. Each service can offer 0..n actions and 0..n properties to allow a user to change the state and be informed of changes in the state of the Device.

Each service that a Device offers is represented in a Provider, and each Device can own several Providers in a 1:n relationship.

The Device that offers your services can be represented in code using the classes included in the ohNet API:

- `DvDevice`
- `DvDeviceStandard`

The `DvDevice` object is not published by any protocol and is not accessible over a network. It should not be used to publish Devices over a network.

The `DvDeviceStandard` object is used to publish Devices and their services over all the protocols supported by ohNet. You should use `DvDeviceStandard` to represent your services' device in the code you write.

Code samples used in the following sections of this document show instances of `DvDevice` being used. `DvDeviceStandard` is derived from `DvDevice`. While Providers will operate on either type of Device, almost all clients of ohNet will find `DvDeviceStandard` most appropriate for their needs.

Creating the Device

To create a new `DvDeviceStandard` you must pass the Device's UDN as an argument:

```
VolumeControl::VolumeControl()
{
    string udn = GenerateGloballyUniqueIdentifier();
    iDevice = new DvDeviceStdStandard(udn);
}
```

Important

The UDN must be globally unique to ensure the service is correctly recognized by Control Points. A Control Point cannot differentiate between Devices that use the same UDN. Only one Device that uses the duplicated UDN will be displayed.

Typically the Device's MAC address or a GUID are sufficient. Physical devices that host multiple `DvDevice`-derived instances must have the MAC address augmented so that each UDN is unique.

Setting the Device's attributes

The method you use to create the Device is also the best place to set its attributes. This keeps the Device's details separate from each Provider the Device may end up using. For example:

```
            iDevice->SetAttribute("Upnp.Domain", "openhome.org");
            iDevice->SetAttribute("Upnp.Type", "Volume");
```

```
            iDevice->SetAttribute("Upnp.Version", "1");
            iDevice->SetAttribute("Upnp.FriendlyName",
                "onNet Volume Controller");
            iDevice->SetAttribute("Upnp.Manufacturer", "None");
            iDevice->SetAttribute("Upnp.ModelName",
                "ohNet Volume Controller");
```

Important

You must set the Device's attributes before you enable it (see *Enabling the Device* below). You cannot set attributes while the Device is enabled.

Enabling the Device

You enable the Devices by calling `SetEnabled` on each one:

```
iDevice->SetEnabled();
```

The Device stack broadcasts the required alive messages to allow your Device to be discovered. Providers won't be advertised or discoverable until after `SetEnabled` has been called. Providers can be used as soon as the associated Device has been detected.

This is typically the last thing you do after you have created both the Device and its Providers, and also only after you have set the Device's attributes.

Altering the Device

You cannot add new Providers or change the attributes after a Device has been enabled. However, you can temporarily disable the Device to make required changes as required.

You can call `SetDisabled` on a Device at any time:

```
iDevice->SetDisabled(aDisableCompleted);
```

This function takes a callback as its only argument. The callback confirms that all outstanding notifications of the device's disappearance, over all protocols, are complete. You must wait for this callback to return before you can begin altering the Device.

When the callback later runs the Device is no longer available and cannot be detected. The Device and all of its services are removed from the network.

You can now add more services by creating new Providers or alter the Device's description by changing attributes.

When you have finished making your changes you can reenable the Device as before:

```
iDevice->SetEnabled();
```

Provider

Devices offer at least one service, with each service defined by a list of actions and a list of properties. Each action and property is described in XML. The XML is used to generate code which a Device uses to receive, interpret and execute messages sent from a range of different Control Points.

The generated code is called a Provider. A Provider is a class which encapsulates the service on the Device, turning the actions into functions and properties into member variables. Each Provider presents the service it represents as an API that you can program to.

Providers are generated as base classes which you must sub-class. You add your own code to provide the specific implementation of your Device's service.

Note

A high percentage of code for Providers can be auto-generated from a service XML file. If you need to generate your own Provider you can use the tools supplied with ohNet. See *ohNetGen on page 12* for more information on generating your own Provider from a service XML.

Every AV service from both UPnP and OpenHome is included in the ohNet library as pre-generated Providers available for immediate use.

Actions

A service's actions are represented in code by functions. Every action that a service offers is given its own function in the Provider.

ohNet never treats an action as mandatory. ohNet does not require you to implement a service's action if you do not want to support it. However, some protocols will assume the implementation of certain actions is mandatory.

Note

Clients using your Device's service may not respond as expected if you do not implement an action that your protocol treats as mandatory. Refer to your service's definition for the protocols it is published over to check if an action is mandatory or not.

If you choose to override the base Provider's function and advertise your support for the associated action, you must enable it in the Provider's constructor:

```
EnableAction<action_name>;
```

You must include the `EnableAction<action_name>` call for each action you want to define in your Provider.

You then include a virtual function in the body of the Provider's class definition:

```
void <action_name>(uint32_t aVersion, <argument_list>);
```

Override the function by including it in your class implementation:

```
void <class_name>::<action_name>(uint32_t aVersion, <argument_list>)
```

```
{
    //function definition here
}
```

Add your own code in each function to provide its implementation.

Properties

A service's properties are represented in code as member variables of generated Provider classes. Each property generates two access functions, as follows:

`GetProperty<property_name>(value_to_be_returned)`

`SetProperty<property_name>(<new_value>)`

A property's `GetProperty` function is used to respond to Control Point requests for an update on the property's current value.

The `SetProperty` function is used to change a property's value. This applies when the update comes via a request from a Control Point and also when the update is local to the Device itself.

Calling a property's `SetProperty` function also automatically updates any registered subscribers. You do not need to add any extra code to enable this. However, updates can be temporarily suspended in the event you want to ensure several properties are updated in sync (see *Synchronous updates below* for more details).

Updating related properties

Some services offer properties that are closely related to each other and are updated at the same time. The Device Stack offers a way for you to ensure that the Provider broadcasts the updated values of several different properties at the same time.

Calling `PropertiesLock` ensures that the Provider will not event changes to any of its properties. This means that several properties can have their state changed without the change being immediately updated on a Control Point.

Finally you can release the Provider and allow it publish the updates to all changed properties in one message. Altogether, the code for this looks like this:

```
void ClockProvider::UpdateTime(uint aHour, uint aMinute, uint aSecond)
{
    PropertiesLock();
    PropertySetHour(aHour);
    PropertySetMinute(aMinute);
    PropertySetSecond(aSecond);
    PropertiesUnlock();
}
```

The Provider always remains fully aware of updates to all properties while `PropertiesLock` is in effect. All properties that have changed when `PropertiesUnlock` is called will have their new value evented out to subscribers.

Duplicate updates

The Device Stack automatically and safely filters out events for properties whose value hasn't actually changed when a related property's value has. A good example of this is the clock sample code shown above.

The property called second will be updated very regularly with values cycling through 0 to 59, and then repeating the pattern. Its value will change each time. However, the minute property will only change once during the 60 second cycle, and the hour property once every 3600 seconds.

The properties' unchanged state will not be evented out to avoid sending duplicate values to subscribers even though the `PropertySetHour` and `PropertySetMinute` functions are called.

Rapid updates

If a Provider's properties receive very rapid updates, the stack may not publish each of those updates. The Device stack will always publish the most up-to-date value of each property in a Provider, but may skip some of the values the properties were assigned in the interim period (typically less than a second).

This avoids flooding the network with traffic from a Provider whose properties are updated very quickly.

Creating a Provider

You must create a `DvDevice` object for the Provider to belong to before you create the Provider. See *Device on page 6* for more details on Devices and how to create them.

Each Provider takes one Device as an argument. The Device you pass in is the Device that offers the service encapsulated in the Provider.

To create a Provider you must pass in the `DvDevice` object you created earlier:

```
iVolumeProvider = new VolumeProvider(*iDevice);
```

If a Device offers multiple services, you must create a new Provider for each service in the same way:

```
iPlayListProvider = new PlayListProvider(*iDevice);
iScheduledRecordingProvider = new ScheduledRecordingProvider(*iDevice);
```

Appendix

ohNetGen

Providers for all AV services published by UPnP and OpenHome are provided as part of the ohNet SDK for you to use when developing your device's network communication. If the service you require is not provided in the list of Providers available you can create your own using the ohNetGen tool and the service XML files you need to describe the device's services.

Prerequisites

ohNetGen is a .NET development tool and requires a .NET runtime to be able to run. A suitable runtime should already be available if you are using Windows. You may need to install mono if you are using Linux.

To generate a Control Point stack Proxy:

1. Open a command line terminal.
2. Change to your installed ohNet directory.
3. On Windows enter the following command:

   ```
   ohNetGen.exe --language=<...> --stack=<...> --xml=<...> --
   output=<...> --domain=<...> --type=<...> --version=<...>
   ```

   On Linux enter the following command:

   ```
   LD_LIBRARY_PATH=<path_to_ohNetGen> mono <path>/ohNetGen.exe --
   language=<...> --stack=<...> --xml=<...> --output=<...> --
   domain=<...> --type=<...> --version=<...>
   ```

   See the table below for descriptions of each of the arguments you must pass to ohNetGen.
4. The tool will parse the XML you supply it and use the T4 templates included in the OpenHome SDK to generate the required code for your Proxy.

Note

ohNetGen can be integrated into the build process for your Provider. This allows you to treat your service XML file as a source, rather than the Provider that the ohNetGen tool will generate.

ohNetGen arguments

The ohNetGen tool needs several arguments to be defined before you can successfully generate the Provider. All of the following arguments are mandatory.

Option	Description
`--language`	The language you want the code to be output in. Options are: • `cpp` — C++ • `cppcore` — OpenHome's own C++ variant, using custom buffers.

Option	Description
	• `c` — C • `cs` — C# • `js` — JavaScript Note that the `js` option is used only for generating the proxies. You must specify `--stack=cp` to be able to use the `js` language option.
`--stack`	The stack you are generating files for. Options are: • `cp` — Control Point stack. Use this option to generate a Proxy. • `dv` — Device stack. Use this option to generate a Provider. See the *OpenHome ohNet Device stack* document for more details about using Providers.
`--xml`	The path and name of the service XML file you are using to generate your Proxy.
`--output`	The path to the directory where you want your generated files to be created.
`--domain`	The domain for the service. For example: `openhome.org`.
`--type`	The service's type. For example: `Volume`.
`--version`	The version number of the service. For example: `1`.

The `domain`, `type` and `version` options are used to name the generated file. For example, if you specify the following options in an ohNetGen command:

```
ohNetGen.exe --language=cpp --stack=dv --xml=c:\UPNP\Services\Volume.xml
--output=c:\ohNet\Providers --domain=openhome.org --type=Volume --
version=1
```

you'll be provided with a device Provider, which contains code generated in the C++ language, and is called **DvOpenhomeOrgVolume1Std.cpp**.

Notes

- The Dv prefix denotes this file as a Device Stack Provider. In contrast, Control Point Proxies are prefixed with Cp.
- The output for C and C++ generated files also includes a header file. The header file is also named according to the options you provide in the ohNetGen command. For example, the header file for the sample given above would be called **DvOpenhomeOrgVolume1.h**.

  The Std affix is used only in **.cpp** files and not in the associated header files.