Penthera Cache&Carry™

Client Developer Guide (Universal Windows)

This document describes integrating and using the **Penthera Cache&Carry Windows SDK**.

The SDK is the client piece of Penthera's Cache&Carry (C&C), a software system that manages download, storage, and playback of videos on mobile devices. We assume that you will integrate the SDK into your own streaming video app that handles all UI/UX, user authentication, DRM, and video playout.

This document is a how-to guide. It will teach you to:

1. compile and run a sample application using the SDK
2. Reference the SDK in your application
3. perform common functions using the SDK: enqueue, play, expire, configure, etc.

We assume you are an experienced Windows developer who knows your way around VisualStudio and related VS development tools.

The SDK communicates with a server, the **C&C Backplane,** using an internal, proprietary web services protocol. This communication occurs via regular client-server syncs. Penthera hosts a developer server instance, at `demo.penthera.com`, which you may use to build a proof-of-concept app.

Internally, the SDK is codenamed "Virtuoso." You'll notice this a lot in the headers.

We're here to help! Email `support@penthera.com` if you run into any problems.

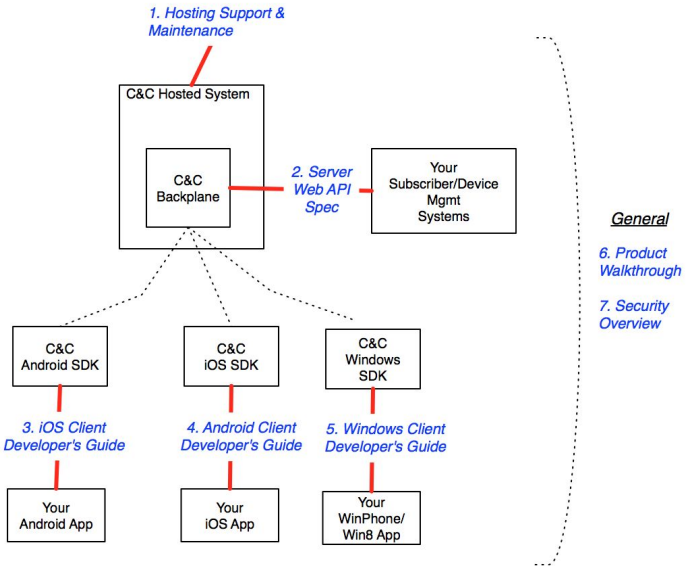
NOTE: This document contains method signatures and reference source code. We try to keep this document up-to-date, but you'll find the **authoritative** header files and reference source in the windows developer package.

Table of Contents

Other Documentation

This document is part of a family of documents covering Cache&Carry:

Cache&Carry: Documentation "Map"



Let's Get Started

The Developer Package

We'll provide you details of a public github repository where you can access the windows developer package. The contents of the deliverable are:

- **CnCSdkDemo**:Visual Studio 2017 project- contains source code for demo app that uses the SDK
- **docs** html documentation of the SDK API's
- **lib**: debug/release versions of the SDK library and its dependencies
- **ext** Third party extensions the SDK relies on.

SDK extension dependencies

The SDK Cannot currently package Visual Studio extensions. It has a dependency on the **`SQLite for Universal Windows Platform`** extension. The Extension must be installed and the client application must reference it. For convenience we have added the correct version into the ext folder of the SDK developer package.

SDK Capabilities

The SDK needs one capability declared in the application's appxmanifest, it must declare the Internet(Client) capability in order to download media. = project / solution and load in the

To build and run CnCSdkDemo:

1. Install the **`SQLite for Universal Windows Platform`** extension. It can be found in the `ext` folder of the developer package.(sqlite-uwp-3200000.vsix).
2. In Visual Studio select `File - open Project / Solution` and select the Harness Solution in the `CnCSdkDemo` folder.
3. If not already referenced, add a reference to the Sqlite library you just installed.

4. Run or Debug the SdkDemo module in an emulator or on a device.
5. You will need a public and private key to build the application. Please contact support@penthera.com to get them.

Congratulations! You've now got a video-downloading app up and running. You're ready to develop your own apps with the SDK.

Note: The Demo application only plays out mp4 files. You will need to provide a player for segmented video.

Building Your Own App

When building your own application you will need to add reference to the CnC SDK. The SDK is a multi-architecture build. Libraries for x86, x64 and ARM are provided within the deliverable. You must use the correct library for your build. To make this easier you should add the following ItemGroup to your applications project (.csproj) file.

```
<ItemGroup>
    <Reference Include="CnCWindowsSDK" Condition="'$(Configuration)|$(Platform)' ==
'Debug|x64'">
      <HintPath>..\lib\x64\Debug\CnCWindowsSDK.dll</HintPath>
    </Reference>
    <Reference Include="CnCWindowsSDK" Condition="'$(Configuration)|$(Platform)' ==
'Debug|arm'">
      <HintPath>..\lib\ARM\Debug\CnCWindowsSDK.dll</HintPath>
    </Reference>
    <Reference Include="CnCWindowsSDK" Condition="'$(Configuration)|$(Platform)' ==
'Debug|x86'">
      <HintPath>..\lib\x86\Debug\CnCWindowsSDK.dll</HintPath>
    </Reference>
    <Reference Include="CnCWindowsSDK" Condition="'$(Configuration)|$(Platform)' ==
'Release|arm'">
      <HintPath>..\lib\ARM\Release\CnCWindowsSDK.dll</HintPath>
    </Reference>
    <Reference Include="CnCWindowsSDK" Condition="'$(Configuration)|$(Platform)' ==
'Release|x64'">
      <HintPath>..\lib\x64\Release\CnCWindowsSDK.dll</HintPath>
    </Reference>
    <Reference Include="CnCWindowsSDK" Condition="'$(Configuration)|$(Platform)' ==
'Release|x86'">
      <HintPath>..\lib\x86\Release\CnCWindowsSDK.dll</HintPath>
    </Reference>
  </ItemGroup>
```

Replace the HintPath to point to the location of the SDK library.

Import the Virtuoso Public Packages

Anywhere you want to access SDK classes or methods, you must import the public namespace:

```
using Penthera.VirtuosoClient.Public;
using Penthera.VirtuosoClient.Public.Events;
```

We'll assume these lines are included in all the examples below.

Initialize the Engine

`IVirtuosoClient` is the main SDK class. It enqueues, downloads, and updates the enclosing app with state information.

`IVirtuosoClient` is a singleton: one and only one instance ever exists. It will automatically clean itself up upon receiving critical system events, like app termination or low memory conditions.

Since C# interfaces cannot implement static methods, you access the `IVirtuosoClient` instance via the

public class `VirtuosoClientFactory`. Call the `ClientInstance` method to initialize the object and get a pointer to the valid instance:

```
IVirtuosoClient* virtuosoClient = VirtuosoClientFactory.ClientInstance();
```

Startup

To prepare the SDK for use, you must call `StartupAsync`.

Penthera will provide you the URL of your Backplane instance, along with an app-specific secret/key that allows your SDK instances to authenticate to your Backplane instance.

Invoke this method as early on in the startup process as possible, so all environment settings and devices are properly synced early in your app lifecycle.

Sample Initialization Code

Here's a template for initialize/setup of the SDK. Inside of the `Login.xaml.cs` method `Login_Btn_Clicked`:

```
// The login process might take a little while.
// For simplicity, rather than using a modal progress or some
// other UI mechanism to allow the user to see what's going on, just prevent them
// from rapidly tapping the login button.
private bool _loggingIn = false;
private void Login_Btn_Click(object sender, RoutedEventArgs e)
{
    // Prevent button crash tapping
    if (_loggingIn)
        return;

    _loggingIn = true;
    IVirtuosoClient client = VirtuosoClientFactory.ClientInstance();

    // Call the StartupAsync method as early as possible in your app
    // lifecycle. Since user credentials are required, this is usually done
    // immediately after you have authenticated the user in your own app.  In this
    // example, we do not authenticate, but rather accept user inputs as valid.
    // Virtuoso will automatically create a new user and assign this device to that
    // user if the user does not already exist.
    client.StartupAsync(
                (string)DefaultViewModel["default_url"],
                (string)DefaultViewModel["default_user"],
                "<your defined external device ID>",
                Config.PRIVATE_KEY,
                Config.PUBLIC_KEY
                       ).AsAsyncAction().Completed = (isender, iargs) =>
                                {
                                    _loggingIn = false;
                                };
}
```

It's your responsibility to supply a unique user ID to the SDK in `StartupAsync`. The SDK uses this user ID in reporting, and to enforce business rules (such as "max number of download-enabled devices per user"). If you don't know the user ID by the time application startup executes, you'll need to delay calls to startup until you know the user ID, such as after an opening user login dialog.

For an example, see the SDK Demo Visual Studio Solution.

Registering for the SDK's event messages

In general, the app follows the Windows standard "property binding" data interface patterns. Some SDK actions may require additional or special handling, and events are provided to capture these instances. Make sure your app registers to receive these events:

```
event RemoteWipeCompleteEventHandler RemoteWipeComplete;
event VirtuosoClientStatusChangeEventHandler VirtuosoClientStatusChange;
```

```
event VirtuosoCcbDeviceSavedEventHandler VirtuosoCcbDeviceSaved;
event AssetSizeValidationWarningEventHandler AssetSizeValidationWarning;
```

Look for the latest set of these events, and detailed descriptions for each of them in the documentation package. The documentation will be found in the `Penthera.VirtuosoClient.Public namespace.`

Common Functions

Here we lists common ways to use the SDK. This is just a sliver of the overall SDK functionality; after you're done here, have a look at the HTML SDK documentation package to see what else is available.

In the SDK, every downloadable object is an `IAsset`. The current version of the Windows SDK only supports singleton items, e.g. a .mp4 file;and segmented HSS video formats.

Enqueue an Item

Create an `IAsset` instance, then enqueue that object:

```
var testAsset = ...
IAsset _assetToAdd = VirtuosoClientFactory.ClientInstance().NewAsset(
                         testAsset.AssetId,
                         testAsset.Title,
                         testAsset.FileUrl,
                         testAsset.Title,
                         testAsset.PublishDate,
                         testAsset.ExpiryDate,
                         testAsset.ExpiryAfterPlay,
                         testAsset.ExpiryAfterDownload,
                         testAsset.AssetType

                                                                  );

bool success = VirtuosoClientFactory.ClientInstance().EnqueueFileAsync(_assetToAdd);

...
```

Refer to the SDK documentation for a full description of the behavior and syntax of every parameter.

Background Downloading

Microsoft strictly constrains what an app can do while it's not in the foreground. The SDK provides as much functionality as possible given these constraints.

Download Limits: The SDK makes use of Microsoft's BackgroundDownloader to download while the enclosing app is not actively running. Microsoft's BackgroundDownloader policies around download are intentionally opaque and the SDK is not capable of generating system events or enforcing all SDK policies while the enclosing app is not actively in the foreground. The BackgroundDownloader will not generate events or wake the enclosing app to notify of download completion. Thus, the download will not be considered complete and Backplane events for download completion will not be sent, until the app is returned to the foreground.

Configure Download Rules

Several behavioral properties control the SDK's behavior:

In general, the SDK will download assets in the order that they were enqueued. This is not, however, guaranteed, and various conditions on the device or OS may cause the SDK to complete downloading one asset before another, earlier-enqueued asset. Since the SDK cannot guarantee the queue is downloaded strictly in order, the SDK does not support queue re-ordering.

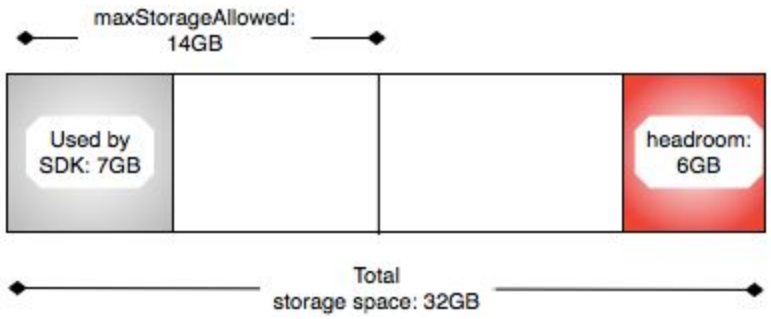
All SDK settings are accessed via the `IVirtuosoSettings` object retrieved from the `IVirtuosoClient` instance. Every property in this object follows the `INotifyPropertyChanged` interface, and is therefore bindable via XAML. Properties that provide both a get and set accessor are two-way bindable. However, the SDK will not commit the changes until a the `Save` method is called on the `IVirtuosoSettings` instance.

The following properties are available:

`ulong` **`MaxStorage`** (in MB; default=10 TB): disk space that the SDK will download and manage on the device. While downloading in the foreground, the SDK will continue to download until it reaches `MaxStorage`, even if that means stopping with a partially completed file. While downloading in the background, the SDK will not initiate a download unless, after downloading that item, the total device storage used by the SDK will still be under `MaxStorage`.

`ulong` **`Headroom`** (in MB; default=300): disk space that the SDK will leave available on the device. While downloading with the app in the foreground, the SDK will continue to download until `Headroom` space is left on the device, even if that means stopping with a partially downloaded item. When downloading in the background, the SDK will not initiate a download unless, after downloading that item, at least `headroom` space will still be free on the device, based on the item's `expectedSize` value. If you didn't provide an expected size, the SDK will perform the download in the background and may violate headroom restrictions.

ulong **MaxPermittedSegmentErrors** This setting controls how many segments in a segmented file may fail before the download is considered to be in an errored state.



Visualizing `MaxStorage` *and* `Headroom` *parameters. Here the device has 32GB disk space. The SDK will always preserve 6GB free space on disk. Currently, the SDK is using 7GB, and will never use more than 14GB total.*

Boolean **`DownloadOverCell`**: whether the SDK is permitted to download over a cellular network. This value is a **permission**, and does not guarantee that downloads **will** continue on a cellular network. It only indicates the the SDK **may** download over cellular, should internal business rules allow it. If this value is NO, downloads will never happen over a cellular connection.

Pause/Resume Downloading

Use the '`DownloadsEnabled`' property on the `IVirtuosoSettings` object to start and pause downloads. You can also use two-way XAML binding.

Pause:

```
IVirtuosoSettings settings = VirtuosoClientFactory.ClientInstance().Settings;
settings.DownloadsEnabled = false;
settings.Save();
```

Resume:

```
settings.DownloadsEnabled = true;
settings.Save();
```

Cancel Downloads

To cancel one download:
```
VirtuosoClientFactory.ClientInstance().RemoveFromQueue(asset);
```

To cancel *all* downloads:
```
VirtuosoClientFactory.ClientInstance().FlushQueueAsync();
```

Clear SDK State

If the enclosing app were to de-authenticate a user, the app may at the same time also wish to delete all downloaded assets:

```
VirtuosoClientFactory.ClientInstance().DeleteAll();
```

In addition, the administrator may choose to schedule a remote wipe of any device. See the Backplane documentation for additional details.

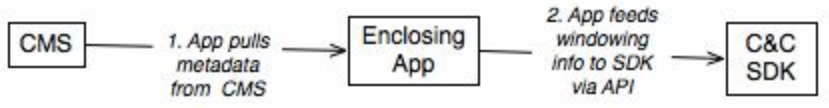
Set Availability Window for an Item

The 'Availability Window' governs when the video is actually available for playout by the user. The SDK enforces several windowing parameters on each video:

Windowing Parameter	Description
Publish Date	The SDK will download the video as soon as possible, but will not make the video available through any of its APIs until after this date.
Expiry Date	The SDK will automatically delete the video as soon as possible after this date.
Expiry After Download (EAD)	The duration a video is accessible after download has completed. As soon as possible after this time period has elapsed, the SDK will automatically delete this video.
Expiry After Play (EAP)	The duration a video is accessible after first play. As soon as possible after this time period has elapsed, the SDK will delete this video. To enforce this rule, the SDK has to know when the video is played, so be sure to register a `play-start` event when the video is played.

The Backplane stores a global default value for EAP and EAD. You may set these values through the Backplane web API. The Backplane transmits these default values to all SDK instances.

Typically, a Content Management System (CMS) stores the windowing information for an item, and communicates it through a web API to the enclosing app. The app then feeds this windowing information to the SDK:



Step 2 in the above diagram occurs when you create the `IAsset` object. You can also modify the values later, via the appropriate class properties.

```
VirtuosoClientFactory.ClientInstance().NewAsset(
        "your_unique_asset_id",
        ”Test Video”,
        ”http://path/to/your/asset.mp4”,
        null,
```

```
        DateTime.UtcNow(),                      // Publish Date: Available now
        DateTime.UtcNow().AddSeconds(604800),   // Expiry Date: Expires in 7d
        43200,                                  // Expiry After Play : 12h
        86400,                                  // Expiry After Download : 24h
        EAssetType.Single
                                            );
```

The SDK will delete an asset as soon as possible after the asset expires. Also, when a caller tries to access an expired item via the API, any downloaded files associated with that item will be auto-deleted from disk and any attempts to access the local file URLs will return nil.

Enable Device for Download

The Backplane tracks which devices are enabled for download, and enforces a global "max download devices per user" parameter.

Individual SDK instances "know" their own 'download-enabled' status, because the Backplane communicates it to them. An SDK instance whose download-enabled status is false can enqueue, but will not download enqueued items.

The IDevice interface implements the `INotifyPropertyChanged` interface, so you can use two-way XAML bindings to include the download-enabled state in your app. You can also attempt to toggle the download-enabled status for a device as follows:

```
// Set the event handler to handle success or error
VirtuosoClientFactory.ClientInstance().VirtuosoCcbDeviceSavedEventHandler +=
Instance_VirtuosoCcbDeviceSaved;

VirtuosoClientFactory.ClientInstance().ThisDevice.DownloadEnabled =
    !(VirtuosoClientFactory.ClientInstance().ThisDevice.DownloadEnabled);

...
...

void Instance_VirtuosoCcbDeviceSaved(object sender, VirtuosoDeviceSavedEventArgs e)
{
    // Handle success or error
    if (!e.Success)
    {
        MessageDialog popup = new MessageDialog("Device Save Error: " + e.ErrorString);
        popup.ShowAsync();
    }
}
```

NOTE: The Backplane also offers a web API to do this; see the server documentation.

Configuring Logging

There's two separate logging paths in the SDK:

Event Logging

The SDK will capture many different events of potential business value, e.g. when a download starts and stops, offline playout, etc. These events will be tracked automatically by the SDK and will be reported to the Backplane during sync.

Since the SDK is not aware of asset playback, you must include special calls within the enclosing app whenever an asset is played.

To log when an asset is played:

```
VirtuosoClientFactory.ClientInstance().VirtuosoLogger.LogPlayStart(asset);

To log when an asset stops playing:
VirtuosoClientFactory.ClientInstance().VirtuosoLogger.LogPlayStop(asset,elapsedTime);
```

Debug Logging
The SDK generates lots of developer-friendly debug information.

In release builds, by default, the SDK logging system is very quiet. To configure logging level:
`VirtuosoClientFactory.ClientInstance().VirtuosoLogger.LoggingLevel = VirtuosoLogginLevel.Warning;`

Play a Downloaded Item

The SDK does not provide a video player.

To play an `IAsset`, retrieve theMediaPlaybackSource for the object. If the file path is not nil, then the file exists and is within its viewing window. You can provide this local path to your player for playback.

Note: Make sure to configure proper play start and play stop events, so the SDK can later enforce "expiry after playback."

Example:

XAML:

```
<Grid Grid.Row="2" x:Name="videoRegion">
    <MediaElement x:Name="mediaPlayer"
          Width="600"
          AutoPlay="False"
          AreTransportControlsEnabled="True" />
</Grid>
```

XAML.CS:

```
//Set the playback source once the asset is retrieved from the client.
private async void NavigationHelper_LoadState(object sender, LoadStateEventArgs e)
        {
            var guid = (Guid)e.NavigationParameter;
            QATestDataSource.GetAssetAsync(guid).AsAsyncOperation<IAsset>().Completed =
(sender2, args) =>
            {
                Asset = sender2.GetResults();

                //set source to a local variable
                var source = Asset.MediaPlayBackSource;
                //check if source exist
                if (source != null)
                {
                    //set source to player
                    mediaPlayerElement.SetPlaybackSource(source);
                }
                DefaultModel["Asset"] = Asset;
            };
        }

// Handle media player state changes and log with SDK
private static DateTime timestamp;
void mediaPlayer_CurrentStateChanged(object sender, RoutedEventArgs e)
{
    TimeSpan diff;
    switch( mediaPlayer.CurrentState)
    {
        case MediaElementState.Playing:
            timestamp = DateTime.UtcNow;

            VirtuosoClientFactory.ClientInstance().
                    VirtuosoLogger.LogPlayStart((IAsset)DefaultViewModel["Test"]);
```

```
            break;

        case MediaElementState.Paused:
            diff = DateTime.UtcNow - timestamp;
            VirtuosoClientFactory.ClientInstance().
                    VirtuosoLogger.LogPlayStop((IAsset)DefaultViewModel["Test"],
                    (ulong)diff.TotalSeconds);
            break;

        case MediaElementState.Stopped:
            diff = DateTime.UtcNow - timestamp;
            VirtuosoClientFactory.ClientInstance().
                      VirtuosoLogger.LogPlayStop((IAsset)DefaultViewModel["Test"],
                      (ulong)diff.TotalSeconds);
            break;

        default:
            break;
    }
}
```

Appendix A: How Downloading Works

Here we describe what happens after you enqueue a video for download. This section is for the curious developer. You don't need to understand this in order to use the SDK.

Virtuoso follows a "Rule of Threes" in downloading:

1. Proceed through the download queue in order. Virtuoso will download one asset at a time.

2. If Virtuoso encounters an error downloading the file, it will try that file two more times before moving on. (This is the "inner" three).

3. When it reaches the end of the download queue, Virtuoso will return to the beginning of the queue and make another pass, trying to download the errored files.

4. Once it has encountered a file in three separate passes through the queue, Virtuoso will mark the file in error and will no longer try to download the file, until you reset that file using the `IAsset.ResetErrorState()` method. (This is the "outer" three).

The above is a slight simplification. If Virtuoso encounters a "fatal" error in step 2, it will not retry the file immediately. An error is fatal if there's no point retrying immediately (e.g. the HTTP server advertises an unexpected mime type). Depending on the type of error, Virtuoso may retry the download again at a later time, until the "rule of threes" is satisfied.

The IVirtuosoClient communicates download issues in two ways.

1. When the client encounters an issue that will eventually cause the file to be marked as blocked, it updates the asset's `DownloadErrorCount, DownloadRetryCount, and DownloadError` properties appropriately. These properties receive property change notices, and you can bind to them in XAML or use the change notification events within code to take appropriate actions.

2. When the client determines a potential issue exists it issues an `event`. This notice indicates that something unexpected happened, but the file download will still finish and be marked as successfully completed.

The following error conditions are processed by the IVirtuosoClient. For the most up-to-date list of errors, see the `VirtuosoDownloadErrorType` enumeration in the documentation.

Condition	Description	Retry?
Invalid mime type	The MIME type advertised by the HTTP server for the file isn't included in the MIME types whitelist you provided earlier.	No
Observed file size disagrees with expected file size	After the download completes, the observed final size didn't match the expected size.	Yes
Network error	Some network issue (HTTP 404,416, etc.) caused the download to fail.	Yes
File System Error	The OS couldn't write the file to disk. In most cases, the root cause is a full disk.	No

**** END OF DOCUMENT ****