Porting Guide for CloudTV Nano C++ SDK

Version 4.4

Document Revision History

Date	Version	Description
6/7/2017	4.4 GA	Updates w.r.t. Northbound API changes
28/3/2017	4.3 GA	Minor updates
10/19/2016	4.2 GA	Clarified that HTTPS is not supported by HttpLoader Minor editorial corrections

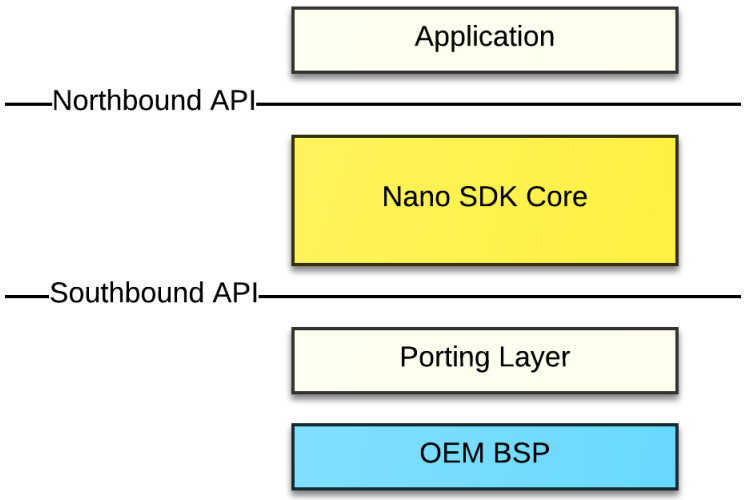
Contents

1. Introduction

1.1. Summary

This document describes the CloudTV Nano SDK and should support the process of integrating CloudTV Nano SDK into a new Set Top Box (STB) platform. This document is primarily intended for engineers and architects who are interested in understanding CloudTV Nano SDK.

CloudTV Nano SDK is the preferred client SDK for the ActiveVideo® CloudTV platform. Its main purpose is to setup and control CloudTV sessions. Its software architecture is divided in 3 layers with 2 integration points with the target platform:



CloudTV Nano SDK consists of two interfaces:

- The Northbound API, which forms the integration point between your application (e.g. an EPG guide or standalone application) and CloudTV Nano SDK. See also the API description included in the SDK distribution package.
- The Southbound API, which is the integration point between CloudTV Nano SDK and the device's middleware and operating system or Board Support Package (BSP).

ActiveVideo® delivers the CloudTV Nano SDK as both compiled software component or as source code, depending on the Non-Disclosure Agreement (NDA). When using the source code to integrate CloudTV Nano into a new platform, ActiveVideo advises against modifying the core of CloudTV Nano SDK. In the scenario where any change to the core is required, it is recommended that you discuss this with ActiveVideo Customer Support team.

A reference implementation of the Southbound API is available for a number of different platforms. Support for POSIX® is one of these. The integrator may choose to use it "as is" or re-implement it for the target platform.

1.2. Copyright

express written permission of ActiveVideo®. Information in this document is subject to change without prior notice.

Certain names of program products and company names used in this document might be registered trademarks or trademarks owned by other entities.

1.3. Other Restricted Rights

This document contains proprietary and confidential information that is the property of ActiveVideo, Inc. It is protected by copyright and distributed under licenses restricting its use, copying, and distribution.

No part of this document may be reproduced in any form without prior written authorization from ActiveVideo, Inc. This guide is provided "*as is*" without warranty of any kind, either expressed or implied, including, without limitation, merchantability, fitness for a particular purpose or non infringement. It may include technical inaccuracies or typographical errors. Changes are periodically made to the information it contains. ActiveVideo Networks may make improvements and/or changes in the programs and/or interfaces described in this publication at any time.

U.S. Patents listed at http://www.activevideo.com/patents

2. Content of the SDK

2.1. SDK content without source code included

After unpacking the archive to a work directory of your choice, you will find that the SDK package is organized as follows:

- **Makefile**: root makefile to build the example clients
- **bin**: pre-built example reference client according to the delivered platform
- **doc**: documentation with the porting guide, API reference and 3rd party license notes
- **lib**: precompiled libraries of CloudTV Nano SDK
- **server-simulator**: A CloudTV platform simulator for validation tests
- **src**: source code to build the reference client
    - **build_env**: all required makefiles and a custom STL implementation to build the source code
    - **clients**: source code of example and pc_reference client applications
    - **core**: header files of core library of CloudTV Nano SDK
    - **http_client**: header files of a simple HTTP client
    - **porting_layer**: header files of the Porting Layer of the CloudTV Nano SDK
    - **stream**: header files of the stream library of the CloudTV Nano SDK

2.2. SDK content with source code included

After unpacking the archive to a work directory of your choice, you will find that the SDK package is organized as follows:

- **Makefile**: root makefile to build the SDK and example clients
- **bin**: pre-built example clients according to the delivered platform
- **doc**: documentation with the porting guide, API reference and 3rd party license notes
- **server-simulator**: A CloudTV platform simulator for validation tests
- **src**: source code to build the reference client
    - **build_env**: all required makefiles and a custom STL implementation to build the source code
    - **clients**: source code of example and pc_reference client applications
    - **core**: source code of the core library of CloudTV Nano SDK
    - **http_client**: source code of a simple HTTP client
    - **porting_layer**: source code of the Porting Layer of the CloudTV Nano SDK
    - **sdk_library**: contains a Makefile that generates a dynamic library of CloudTV Nano SDK
    - **stream**: source code of the stream library of the CloudTV Nano SDK
    - **submodules**:
        - **rplayer**: source code of the 'remote player' deep buffer management and underrun mitigation library
    - **utils**: source code of a utilities library of the CloudTV Nano SDK

It is recommended you preserve this directory structure when integrating Nano to easily handle future Nano SDK drops.

3. Requirements and prerequisites

The requirements for CloudTV Nano SDK are:

- **Build environment requirements:**
  - C++ cross-compiler and tool-chain for the target platform.
- **Target device requirements:**
  - Audio and video decoding capabilities (hardware support is recommended).
  - Audio and video codecs: AC3 or AAC, H.264 or MPEG-2 (AC3 and H.264 are mandatory whenever YouTube pass-through is required).
  - CPU at least 400MHz.
  - API to receive user input events, such as remote control, virtual keyboard from a companion device, USB keyboard, etc…
  - API for retrieving a unique identifier of the device (e.g. STB MAC address or a serial number).
  - API for storing and retrieving persistent data (at least 128 bytes).
  - Memory requirements: 500 KB of heap usage (RAM).
- **For client side overlay support (recommended):**
  - API to draw the overlays (e.g. DirectFB)
  - Memory requirements:
    - 1800 KB of RAM memory to store the images.
    - Graphics memory (best experience): HD double buffered overlay graphics @720p, 24 bit color + 8 bit alpha 7372800 bytes
    - Graphics memory (reduced experience): SD single buffered overlay graphics @576i, 15 bit color + 1 bit alpha 811008 bytes
  - PNG decoder (Depending on your deployment. Other formats are also possible.)
- **For DRM support:**
  - Access to the required DRM system to generate the keys needed to decrypt the stream.
  - API and preferably hardware support to decrypt the stream in real time.
- **For deep buffering support:**
  - A device-specific amount of memory-mapped media RAM (but at least 2MB) to store the buffered stream.
- **For deep buffering support combined with DRM:**
  - The capability to deliver a decrypted transport stream back to the Nano SDK.

For optimal user experience, the platform shall provide low latency components across the chain. Therefore, there are certain non-functional requirements that shall be met:

- **Minimum latency for any user input**, from the moment the event was generated (e.g. user pressed a button in the RCU) to the reception by CloudTV Nano SDK. An acceptable value would be anything lower than 40 milliseconds.
- **Low latency video playback** which requires to reduce to the minimum any possible buffering the video decoder. **The maximum buffering should be 2 frames of video**. The video decoder can be fed with either:
  1. The entire MPEG2-TS; or
  2. Individual audio and video elementary streams.

4. Integration checklist

The following subsections summarize the steps to successfully port CloudTV Nano SDK into a new target platform:

4.1. Adapt the porting layer for your target platform

This is only required if your target platform is non-POSIX. CloudTV Nano SDK comes standard with support for POSIX® compliant platforms.

The Southbound API is the interface that CloudTV Nano SDK uses to interact with the device and the Operating System. In particular, the access to the underlying OS can be grouped in the following areas:

- Threads management and multi threading support (mutex, semaphore and condition variables)
- Socket communication
- Time and timer functions
- Device properties
- Logging
- Key mapping

4.2. Identify a location for persistent storage on the STB

CloudTV Nano SDK needs persistent storage for the client cookie, which is received during session setup with the platform. You can specify the path to a location in the file system of your STB by means of a call to the '`set_base_store_path()`' method of the '`ClientContext`' object instance.

4.3. Write your client application, based on the CloudTV Nano SDK

Implement the calls for all mandatory callbacks and call the methods to set the mandatory client parameters. See the example client source code and the step-by-step guide in section 5.

4.4. Compile the SDK and link it with your client application

Use your platform's cross-compiler to compile the CloudTV Nano SDK (if you did not receive precompiled binaries) and your client application. You may need to modify or add a Makefile in the '`build_env`' directory to meet your platform needs.

4.5. Testing

Install and set up the 'server-simulator'. Configure your new client application to connect to the PC/VM that hosts the server-simulator. Run the tests as described in the certification guide. (Contact your ActiveVideo representative for a copy of the certification guide.)

4.6. Example client

If you desire to make and run the example and test applications located in **src/clients/example** in this release, we have had best results building and running the **example_client** on Ubuntu.

Feel free to run the example client to verify proper server-simulator operation.

4.6.1. Prerequisites

4.6.1.1. Ubuntu Linux

You will need to install the following items for the example_client and server-simulator on Ubuntu:

- `sudo apt-get update`
- `sudo apt-get install openssl openssl-dev libssl-dev make g++ doxygen texlive-* python-pip`

4.6.1.2. CentOS Linux

You will need to install the following items for the example_client and server-simulator on CentOS:

- `sudo yum -y makecache`
- `sudo yum -y install openssl-devel make gcc-g++ doxygen-1.8.5 texlive-* python-pip`

4.6.1.3. Windows

You will need to install the following items for the example_client on Windows:

- MinGW (Minimalist GNU for Windows) with gcc version 4.8 or higher
- MSYS version 1.0
- Update your **PATH** environment variable to ensure it is pointing to the version of **GCC** from the MinGW installation and the **make** tool from MSYS.

4.6.1.4. Mac OS

You will at least need the **Xcode** command line developer tools to be installed. Try typing '`gcc`' or '`make`' in a terminal. If Xcode has not yet been installed, you will automatically be presented with a dialog window. Follow the steps in this dialog to complete the installation.

4.6.2. Build the Nano SDK and example client

The **example_client** will not build without the above packages for the respective platform.

4.6.2.1. POSIX platforms

- Unzip the distribution package (zip file) with the Nano SDK source code
- Open a terminal with bash shell
- Go to the 'root' directory of the unzipped package
- Type '`make`' on Linux systems or '`make TARGET=mac`' on Mac OS systems.

4.6.2.2. Windows platforms

- Unzip the distribution package (zip file) with the Nano SDK source code
- Start the MSYS bash shell (e.g. 'C:\MinGW\msys\1.0\bin\bash.exe')
- Go to the 'root' directory of the unzipped package
- Type '`make TARGET=win`'

5. Application development

The following subsections describe the steps to follow, in order to create your application based on the CloudTV Nano SDK.

CloudTV Nano SDK comes with support for establishing a session with the CloudTV Platform using the RFB-TV protocol and downloading the User Interface (UI) Transport Stream (TS), generated by the CloudTV Platform. How to do the integration, is described in this chapter.

5.1. Overview

In general, the client application will start the session with the CloudTV Platform. All protocol details and handling, i.e. RFB-TV over TCP/IP, are performed by CloudTV Nano SDK and hidden from the client application. This is done by encapsulating the logic in C++ objects and callback interfaces. The callback interfaces must be implemented by the client application, so that it receives a trigger whenever client interaction is required. For example, when it has to paint an overlay image on the graphics display or to tune (back) to a linear TV channel (when the CloudTV session has ended). All callback interfaces must return as fast as possible as they would otherwise affect proper operation of the SDK and internal handling of the underlying protocol.

All calls to `IControl` and `IInput` interface methods are non-blocking. That is, they only post an 'event' to the SDK's internal event queue without the need to obtain an internal lock (mutex). The events are handled by an event handler thread. As a result, calls to these interface do not have an immediate effect. For example, a call to the `suspend()` method on the `IControl` interface will not immediately lead to a state transition of the 'suspended' state. Other methods of the Northbound API may need to lock an internal lock until the requested operation has completed. This is true for all `register_xxxx()` and `unregister_xxxx()` methods. This should not pose a problem since these functions need to be called during the initialization phase of the client code and not while the session is active. The `get_state()`, `get_input()` and `get_control()` are executed without an internal lock.

Another aspect that is covered by CloudTV Nano SDK, is downloading and buffering the transport stream before it is passed to the `IMediaPlayer` interface for playback. In IP environments, the TS is typically received via the HTTP protocol (chunked data) and contains the audio and video elementary streams. Since the TS can be the CloudTV Platform generated UI stream as well as other content, CloudTV Nano SDK needs full control over it. The UI stream requires minimal buffering - for optimal user experience - while other content may allow for more buffering (up to several seconds) in order to mitigate the effects of temporary network issues.

5.2. Integration with the SDK - Northbound API

The Northbound API is the interface to setup and control sessions against the CloudTV Platform. The main classes that are required to establish a session with the CloudTV server are:

- Session: main entry point to CloudTV Nano SDK to establish and control a session with the CloudTV server.
- ISessionCallbacks: your application optionally provides an object that implements this interface to receive callbacks from CloudTV Nano SDK.
- IOverlayCallbacks: your application optionally provides an object that implements this interface to receive requests to display overlays and for clearing the graphics overlay plane (see Appendix A).
- IMediaPlayer: for playback of the transport stream.
- ICdmSession and ICdmSessionFactory: interface to handle DRM licenses.
- IStreamDecrypt: interface that Nano SDK uses to decrypt content.

In order to illustrate the main application flow, ActiveVideo provides an example of an implementation of a client application in the SDK. The following subsections describe the steps in more detail.

5.2.1. Setup and initialization

Prior to establishing an interactive session with the CloudTV Platform, the client application has to initialize the client context, provide a key translation map, initialize the session object, register the callbacks and, optionally, register additional protocols or parameters.

```
using namespace ctvc;

class MyApplication : public Session::ISessionCallbacks
{
    // implement the session callback
    virtual void state_update(State state, ClientErrorCode reason);

    // implement the graphics overlay callbacks
    class OverlayCallbacks : public IOverlayCallbacks
    {
        virtual void overlay_blit_image(const PictureData &picture_data,
                                        const uint8_t *image, uint32_t
length);
        virtual void overlay_clear();
        virtual void overlay_flip();
    };

//  ...

private:
    ClientContext     &m_context;
    OverlayCallbacks   m_overlay_callbacks;
    Session            m_session;

//  ...
};
```

Example of the constructor of your application class and initialization of the `Session` object:

```
MyApplication::MyApplication() :
    m_context(ClientContext::instance()),
    m_session(m_context, this, &m_overlay_callbacks)
//  ...
{
//  ...
}
```

The following snippet of code illustrates how to initialize the mandatory parameters. This is the absolute minimum set of parameters that are required to set up a session. It is possible to set more parameters, but this depends on your deployment.

```
m_context.set_manufucturer("Acme");        // The manufacturer name of the
STB
    m_context.set_device_type("stb-1234");     // The STB model name and/or
number
    m_context.set_unique_id(get_unique_id());  // Unique serial number or
MAC address
    init_keymap(m_context.get_keymap());    // The remote control key
translation map
```

Example on creating the key mappings:

```
void init_keymap(X11KeyMap &keymap)
{
    /// \todo Add all native remote control key codes for your STB.
    keymap.add_mapping(Keyboard::ENTER_KEY, X11_OK);
    keymap.add_mapping(Keyboard::DEL_KEY, X11_BACK);
    keymap.add_mapping(Keyboard::UP_KEY, X11_UP);
    keymap.add_mapping(Keyboard::DOWN_KEY, X11_DOWN);
    keymap.add_mapping(Keyboard::RIGHT_KEY, X11_RIGHT);
    keymap.add_mapping(Keyboard::LEFT_KEY, X11_LEFT);
    // Etc...
}
```

5.2.2. Starting a session

After having configured all the required parameters to start a session, this can be initiated as follows:

```
std::map<std::string, std::string> optional_parameters;
    optional_parameters["lan"] = "eth100";
    optional_parameters["lang"] = "en-US";

    m_session.get_control().initiate("rfbtv://127.0.0.1",
"ctvprogram:youtube",
                                     1280, 720, optional_parameters);
```

The first parameter is the URI to the CloudTV Central Session Manager (CSM). The second parameter is the application that will be launched for your client. Note that this parameter can be left empty, because it can also be configured on the platform.

5.2.3. User input handling

During the lifetime of the session, your application is responsible for forwarding any required key presses. These key presses could come from a physical device, e.g. the Remote Control Unit RCU) or be virtually generated, e.g. a Companion Device. The application has to pass these events via the `send_keycode` method of the IInput interface. The following example illustrates this in its most simple form. If your platform cannot distinguish key presses from key releases, then the `send_keycode` method must be called twice to simulate this.

```
while (m_session.get_state() == Session::STATE_CONNECTING ||
           m_session.get_state() == Session::STATE_CONNECTED) {
        int key;
        if (wait_for_key(key)) {
            // Simulate key down/up event:
            m_session.get_input().send_keycode(key,
Session::KEY_ACTION_KEYDOWN);
            m_session.get_input().send_keycode(key,
Session::KEY_ACTION_KEYUP);
        }
    }
```

5.2.4. Session termination

Your client application implementation is responsible for handling any unrecoverable errors raised by CloudTV Nano SDK. The policy could include:

- automatically restoring the session (if required), or
- sending a trap to a monitoring system, or
- display an error on the user's screen.

Whichever suits your needs and deployment best.

```
void Application::state_update(Session::State state, ClientErrorCode
reason)
{
    if (state != Session::STATE_ERROR && state !=
Session::STATE_DISCONNECTED) {
        return;
    }
    if (reason != CLIENT_ERROR_CODE_OK_AND_DO_NOT_RETUNE) {
        printf("TODO: Retune back to original program\n");
    }
    if (state == Session::STATE_ERROR) {
        printf("TODO: show message on-screen to end-user, code:%d\n",
reason);
        printf("PRESS OK TO CONTINUE\n");  // TODO: Add your keyboard
handling
    }
}
```

In the event that the session is closed, the client application is responsible for freeing its own resources.

5.2.5. Stopping a session

The client application can stop an on-going session at any point in time. This will end the session with the platform and close the connection. You may want to do this when, for example, the STB needs to go to stand-by mode:

```
m_session.get_control().terminate();
```

5.2.6. Suspending and resuming a session

The application can also suspend an on-going session and resume it later on. This may be useful when a Video On Demand (VOD) movie has been selected from a catalog:

```
m_session.get_control().suspend();
// ... play VOD asset
m_session.get_control().resume();
```

5.2.7. Custom Media Player

Below is a simplified example on how to derive from the `IMediaPlayer` interface to implement your own media player instead of using the `SimpleMediaPlayer` class that is provided by the CloudTV Nano SDK. Your implementation of the IMediaPlayer will receive the URL where the Transport Stream has to be fetched from. It has the option to route the data through CloudTV Nano SDK for further processing. This is required to implement the deep-buffering feature. However, there are use-cases where the data cannot be retrieved, but they are sent directly to the A/V decoder. This could be the case when the content is streamed over QAM and it's directly fed into the A/V decoder.

Regardless of the scenario, the STB's middleware has to inform CloudTV Nano SDK about the status of its media player to react accordingly and also inform CloudTV platform.

```
class MyMediaPlayer : public IMediaPlayer
{
public:
    MyMediaPlayer();
    ~MyMediaPlayer();
    virtual ResultCode open_stream(const std::string &uri, const
std::map<std::string, std::string> &stream_params, IStream &stream_out,
IStream *&stream_in/*out*/);
    virtual ResultCode check_stream();
    virtual void close_stream();
    virtual void get_player_info(PlayerInfo &info);
    virtual void register_callback(ICallback *callback);

private:
    ICallback *m_callback;
    bool m_is_stream_present;
};

// Implementation snippets of the MyMediaPlayer class
void MyMediaPlayer::register_callback(ICallback *callback)
{
    m_callback = callback;
}

ResultCode MyMediaPlayer::open_stream(const std::string &uri, const
```

```
std::map<std::string, std::string> &stream_params, IStream &stream_out,
IStream *&stream_in/*out*/)
{
    if (m_callback) {
        m_callback->player_event(PLAYER_STARTING);
    }

    // Open stream and start player, optionally route the stream through
the SDK
    // using stream_out and stream_in ...
    // While streaming, post player_event() messages both in case of errors
and
    // when any error condition is resolved.
    return ResultCode::SUCCESS;
}

ResultCode MyMediaPlayer::check_stream()
{
    if (m_is_stream_present) {
        if (m_callback) {
            m_callback->player_event(PLAYER_STARTED);
            return ResultCode::SUCCESS;
        }
    }

    // Process stream errors ...

    return CHECK_STREAM_IN_PROGRESS;
}

void MyMediaPlayer::close_stream()
{
    // Close stream and stop player...

    if (m_callback) {
        m_callback->player_event(PLAYER_STOPPED);
```

```
    }
}
// ...
```

5.2.8. Media Player Factory

In order to facilitate the creation of your media player object instances, you create a 'factory' class:

```
class MyMediaPlayerFactory : public IMediaPlayerFactory
{
    IMediaPlayer *create()
    {
        return new MyMediaPlayer;
    }
    void destroy(IMediaPlayer *p)
    {
        delete p;
    }
};
```

Next, your factory must be registered with the session object. This way a new instance of `MyMediaPlayer` will be created each time a stream is opened for the 'http' protocol during a session.

```
static MyMediaPlayerFactory s_my_media_player_factory;

 // Register the 'http' protocol to use this my own customized stream
loader
    m_session.register_media_player("http", s_my_media_player_factory);
```

5.2.9. Custom Stream Player using default Media Player

Alternatively to implementing the `IMediaPlayer` interface, it's also possible to use the `IStreamPlayer` interface in combination with `SimpleMediaPlayerFactory` class to get the stream data and feed it into your STB's (hardware) demuxer and audio/video decoder. Ideally, your implementation and the STB's underlying middleware should not add any latency.

The `SimpleMediaPlayerFactory` uses the SDK-provided stream loaders for HTTP and UDP to fetch the transport stream and pass it to the provided `IStreamPlayer` object:

```
void Application::StreamPlayer::stream_data(const uint8_t *data, uint32_t
length)
{
    (void)data;
    (void)length;
    printf("TODO: StreamPlayer::stream_data()\n");
}
```

```
// Registering the IMediaPlayer to be used

static SimpleMediaPlayerFactory<HttpLoader>
http_media_player_factory(m_stream_player);
session.register_media_player("http", http_media_player_factory);
session.register_media_player("https", http_media_player_factory); // TODO:
HTTPS is not yet supported by HttpLoader
```

5.2.10. Handling of encrypted streams

There are deployments where the video stream is encrypted using a specific DRM system. CloudTV Nano SDK neither implements any DRM system nor performs any decryption. However, CloudTV Nano SDK provides with the means to the middleware to decrypt the video content.

All the specifics on the supported DRM systems supported by the CloudTV platform are documented in the RFB-TV specification. All license information is passed to the middleware by creating an `ICdmSession` object via the `ICdmSessionFactory::create()` interface. The middleware shall register an object which implements the `ICdmSessionFactory`:

```
static MyCdmSessionFactory factory;
m_session.register_drm_system(factory);
```

Upon reception of the RFB-TV message *CdmSetupRequest* from CloudTV platform, Nano SDK will create an `ICdmSession` object if the DRM type returned by `ICdmSessionFactory::get_drm_type()` matches the type in the RFB-TV message. The DRM specific information to handle the session will be passed through the method `ICdmSession::setup()`.

Once the video stream is established, those packets that are signaled by the CloudTV platform as encrypted will be passed to the `IStreamDecrypt` object that was registered with `Session::register_stream_decrypt_engine()`:

```
m_session.register_stream_decrypt_engine(decrypt_engine);

//  ...

// For each encrypted chunk, Nano SDK code will invoke
m_stream_decrypt_engine->stream_data(data, length);
```

Nano SDK core expects to receive the data in the clear through the object that was set to the `IStreamDecrypt` object using the method IStreamDecrypt::set_stream_return_path():

```
class DecryptEngine : public IStreamDecrypt
{
public:
    // ...
    virtual void set_stream_return_path(IStream *stream_out)
    {
        m_stream_out = stream_out;
    }
    virtual bool stream_data(const uint8_t *data, uint32_t length)
    {
        // Decrypt data synchronously and pass it back
        decrypt(data, length, clear_data, clear_length);

        // Pass the clear data to Nano SDK core
        if (m_stream_out) {
            m_stream_out->stream_data(clear_data, clear_length);
        }

        return true;
    }

private:
    IStream *m_stream_out;
};
```

Upon closure of the DRM session (e.g. when the client receives the RFB-TV message *CdmTerminateIndication*), Nano SDK will call `ICdmSession::terminate()` of the appropriate object and will delete it.

5.3. Advanced topics

If, for whatever reason, you would not wish to use the default CloudTV Nano SDK HTTP stream download mechanism then you can implement your own and use it with the `MediaPlayer` object instance of your application.

5.3.1. Custom Stream Loader

Below is a simplified example on how to derive from the `IStreamLoader` interface to implement your own stream loader instead of using the loaders that are used by the `SimpleMediaPlayer` class provided by the CloudTV Nano SDK. Note that you must keep buffering to a minimum (no buffering at all would be ideal) and immediately pass data to the stream sink (the `IStream` interface) so that the CloudTV Nano SDK can deal with it.

```
class MyStreamLoader : public IStreamLoader
{
public:
    MyStreamLoader() {}

    ResultCode open_stream(const std::string &uri, IStream &stream_sink)
    {
  // Open the stream according to uri and start feeding stream_sink with
data
  // ...
    }

    void close_stream()
    {
  // Stop feeding stream_sink and close the session
  // ...
    }
};
```

5.3.2. Content Loader

Overlays can be transmitted 'in-band' over RFB-TV or 'out-of-band' via HTTP. The latter is referred to as URL encoding, because the CloudTV server sends the URL of the overlay instead of the image itself. CloudTV Nano SDK allows to register a "Content Loader" to fetch URL-encoded images which could be more efficient in certain configurations. The implementation shall derive from the interface `struct IContentLoader` an d use `Session::register_content_loader()` to register it before starting the session.

```
class MyContentResult : public IContentLoader::IContentResult
{
public:
    // Constructor, destructor and ancillary methods should be here

    virtual ResultCode wait_for_result()
    {
        // This call has to block until the result of the requested
operation is known
        ResultCode ret = ResultCode::SUCCESS;

        // ... checking for the result of the operation and setting it to
"ret" ...

        return ret;
    }
};

class MyContentLoader : public IContentLoader
{
public:
    // Constructor, destructor and ancillary methods should be here

    virtual IContentResult *load_content(const std::string &url,
std::vector<uint8_t> &buffer)
    {
        // The implementation can use different patterns, e.g. synchronous
calls, pool of threads, spawning threads for each request, ...
        MyContentResult *content_result = new MyContentResult();

  // ... request is performed here ...

        return static_cast<IContentLoader::IContentResult
*>(content_result);
    }

    virtual void release_content_result(IContentResult *content_result)
    {
        // Once the result is known, CloudTV Nano SDK will call this method
to release it
        delete static_cast<MyContentLoader *>(content_result);
    }

private:
    // Private methods and members
};
```

Once the implementation is available it has to be registered in the Session class

```
// Ancillary function that creates a new Session object
Session *create_session(ISessionCallbacks *session_callbacks,
IOverlayCallbacks *overlay_callbacks)
{
    MyContentLoader *content_loader = new MyContentLoader();
    Session *session = new Session(ClientContext::instance(),
session_callbacks, overlay_callbacks);

    session->register_content_loader(content_loader);

    // ... Other session parameters

    return session;
}
```

CloudTV Nano SDK provides with a default implementation of `IContentLoader` that is available to any user of the SDK, `DefaultContentLoader`. Before registering it into the Session object, the method `DefaultContentLoader::start()` shall be called with the size of the pool. If size "0" is passed, all HTTP requests will be synchronous. Any allocated resource will be release as soon as `DefaultContentLoader::stop()` is called.

```
#include <core/DefaultContentLoader.h>

// Ancillary function that creates a new Session object using CloudTV Nano
SDK DefaultContentLoader class
Session *create_session(ISessionCallbacks *session_callbacks,
IOverlayCallbacks *overlay_callbacks)
{
 static DefaultContentLoader cloudtv_content_loader;

    Session *session = new Session(ClientContext::instance(),
session_callbacks, overlay_callbacks);

 // This is a mandatory call before registering it into the session object
    cloudtv_content_loader.start(5); // We want to have a pool of 5 threads
    session->register_content_loader(&cloudtv_content_loader);

    // ... Other session parameters

    return session;
}
```

5.4. Protocol Extensions

It is impossible to foresee which CloudTV Platform applications would require an extension of the underlying RFB-TV protocol, i.e. new message types. Because of this, the CloudTV Nano SDK and the CloudTV Platform have a mechanism to 'add' new protocols. In a way, these can be seen as a sub-protocol that is 'tunneled' by means of the RFB-TV '`PassthroughData`' protocol message. One of the uses is to have a direct communication channel between the CloudTV application and the CloudTV Nano client application on the STB. Please note that this channel is not meant for transferring bulk data, as it could affect the round-trip times for handling key presses.

Example of a subclass that derives from `ProtocolExtensionBase`:

```
class ExampleProtocolExtension : public ProtocolExtensionBase
{
public:
    ExampleProtocolExtension();

    // Implement mandatory callback of the IProtocolExtension base class
    virtual void received(const uint8_t *data, uint32_t length);
};

// Create an instance of the protocol extension
class MyApplication
{
//  ...
private:
    ExampleProtocolExtension m_protocol_extension;
```

Registering the protocol extension with the session object:

```
m_session.register_protocol_extension(m_protocol_extension);
```

Below is a simplified example of an implementation:

```
ExampleProtocolExtension::ExampleProtocolExtension() :
    ProtocolExtensionBase("example")
{
}

void ExampleProtocolExtension::received(const uint8_t *data, uint32_t
length)
{
    // Handle the data received by your protocol extension
    //  ...
}
```

6. Porting Layer - Southbound API

6.1. Introduction

CloudTV Nano SDK relies upon a porting layer to abstracts the platform and the operating system where it runs on. ActiveVideo® provides with a reference implementation for POSIX based OS and Windows where certain Middleware specific functions (e.g. storing data in permanent memory) are implemented for regular PCs. The integrator of CloudTV Nano SDK has to validate these assumptions and replace the functionality when needed.

The approach of the porting layer (also known as Southbound API) follows a different paradigm compared to the Northbound API. The

implementation of the porting layer is based in composition, i.e. it resides in a second set of classes that have the same signature and are called from the main interface. Each method of the porting layer calls to its counterpart of the implementation object.

6.2. First example

Here is an example of how to create an implementation for the method Thread::start():

Thread.h

```
class Thread
{
public:
 Thread();

    ResultCode start(IRunnable &runnable)
    {
        return m_impl.start(runnable);
    }

 // ... there are more functions in Thread.h

    struct IThread
    {
        /// \{
        IThread() {}
        virtual ~IThread() {}
        virtual ResultCode start(IRunnable &runnable) = 0;
        virtual void stop() = 0;
        virtual ResultCode wait_until_stopped() = 0;
        virtual bool is_running() = 0;
        virtual ResultCode stop_and_wait_until_stopped() = 0;
        /// \}
    };

private:
    Thread(const Thread &);
    Thread &operator=(const Thread &);

    // Hidden implementation
 IThread &m_impl;
};
```

The above is an example of the Thread class where its implementation is in the Thread::IThread struct. Whenever CloudTV Nano SDK calls the method "start", the Thread class immediately calls to the same method from Thread::IThread.

The following code illustrates the implementation of the method:

Thread.cpp

```
class ThreadImpl : public Thread::IThread
{
public:
    // ... there are more public and private members in ThreadImpl

 ResultCode start(Thread::IRunnable &runnable);
}

// Constructor of the original Thread interface
Thread::Thread() :
    m_impl(*new ThreadImpl)
{
}

// Implementation of the method that will be used when Thread::start() is
called
ResultCode ThreadImpl::start(Thread::IRunnable &runnable)
{
 // ...
}
```

6.3. Generic vs OS dependent code

The porting layer has two main areas:

- Generic code: the classes that belong to this functional group are not related to the OS that hosts CloudTV Nano SDK, but to the Middleware where it runs along with. A classic example is DataStore where a regular filesystem cannot be assumed; the integrator has to decide where the permanent data files are store (e.g. Flash partition, NOR memory or encrypted partition). In order to have a working example, ActiveVideo provides with an implementation based in POSIX calls. The code can be found in **porting_layer/src/generic**.
- OS dependent code: the classes that belong to this functional group are OS related calls that have to be abstracted when the underlying operating system changes. ActiveVideo® provides with a reference implementation for POSIX compliant OS and Windows. The code can be found in **porting_layer/src/posix** and **porting_layer/src/windows**.

6.4. Creating a new porting layer

Before starting with a new porting layer, it is advisable to get familiar with the reference code in **porting_layer/src**. When the integrator is ready to create or modify the code, the first step is to identify whether the change is in the generic code or due to non-supported OS. In the case that new generic code is required, the files should be placed in **porting_layer/src/generic**. However, if the change is driven by a new OS, a new folder should be created in **porting_layer/src** next to **posix** and **windows**.

6.4.1. Adding support for a new OS (platform)

For example, if the integrator wanted to add support for vxworks, the advisable steps would be:

1. Create the directory: **porting_layer/src/vxworks**
2. Place the files in the above directory
3. Modify the Makefile to enable this new platform for **new_target** as following:

```
target_to_system_api_map := \
  test:native tools:native mac:mac linux:posix win:windows \
  vip1002:posix vip1113:posix vip1853:posix albis.8073:posix
albis.8083:posix \
  new_target:vxworks

SYSTEM_API := $(subst $(TARGET):,,$(filter
$(TARGET):%,$(target_to_system_api_map)))

# Overwrite native API with the detected system
ifeq ($(SYSTEM_API),native)
  SYSTEM_API :=
  ifeq ($(OS),Windows_NT) # MingW variable
    SYSTEM_API := windows
  else
    UNAME_S := $(shell uname -s)
    ifeq ($(UNAME_S),Linux)
      SYSTEM_API := posix
    endif
    ifeq ($(UNAME_S),Darwin)
      SYSTEM_API := mac
    endif
  endif
endif

ifeq ($(SYSTEM_API),)
  $(error No system API exists for given target: $(TARGET))
endif

SRC_DIR := src/generic src/$(SYSTEM_API)

ifeq ($(SYSTEM_API),windows)
  SOURCE_FILES := $(foreach dir,$(SRC_DIR),$(wildcard $(dir)/*.c
$(dir)/*.cpp $(dir)/*.cc))
  SOURCE_FILES += src/posix/Mutex.cpp
  SOURCE_FILES += src/posix/Thread.cpp
  INCLUDES += -Isrc/posix/
endif

ifeq ($(SYSTEM_API),mac)
  SOURCE_FILES := $(foreach dir,$(SRC_DIR),$(wildcard $(dir)/*.c
$(dir)/*.cpp $(dir)/*.cc))
  SOURCE_FILES += src/posix/Keyboard.cpp
  SOURCE_FILES += src/posix/Mutex.cpp
  SOURCE_FILES += src/posix/Socket.cpp
  SOURCE_FILES += src/posix/Thread.cpp
  SOURCE_FILES += src/posix/ThreadSleep.cpp
  INCLUDES += -Isrc/posix/
endif

LIB_NAME := porting_layer
include ../build_env/Makefile.include
```

Appendix A — Overlay Updates

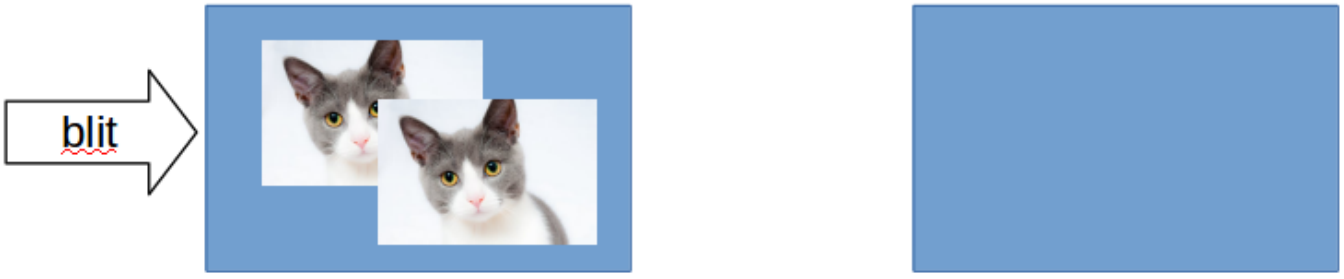
For a better understanding of how overlay updates should work, here's a simplified example to illustrate this mechanism.

- **blit** refers to the overlay_blit_image() method,
- **flip** refers to the overlay_flip() method and
- **clear** refers to the overlay_clear() method of the 'IOverlayCallbacks' interface.

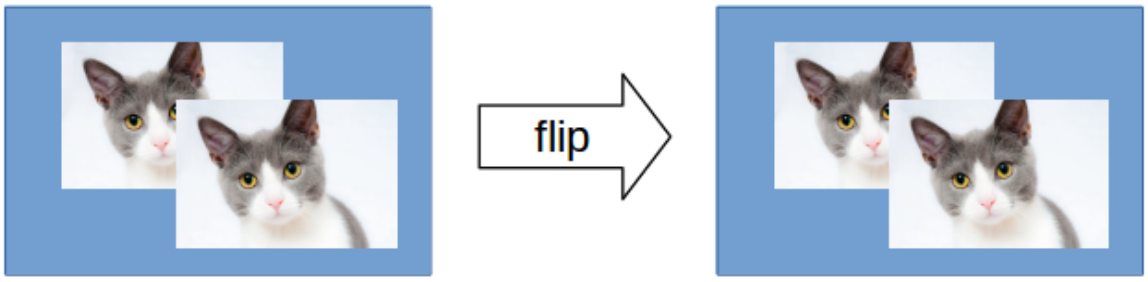
Initially the overlay plane is cleared and transparent. That is, any video playing must be visible. Overlays are typically, as the name suggests, drawn *over* video. In the examples below, blue stands for transparent.

This is the state that would also be in effect after calling 'clear' and 'flip:

When a **blit** is processed, the image is only painted to the invisible 'back buffer'. Transparency in the picture is maintained (the so called 'alpha channel'):

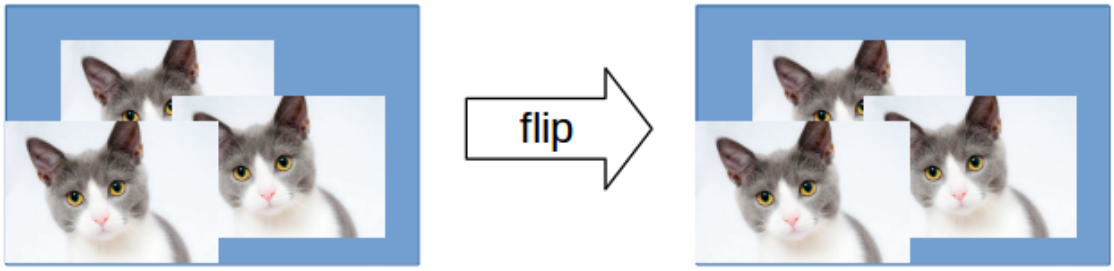


Only on a **flip**, the back buffer is copied to the visible overlay plane. This mechanism is also know as 'double buffering'. Most set-op boxes have a mechanism to prevent flickering of this operation. That means that in practice the copy is usually done during the vertical scan.



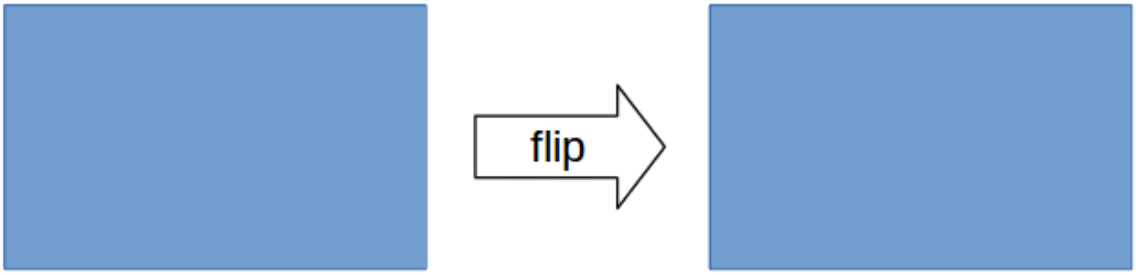
Any new blits are always painted over whatever happens to be in the back buffer. The visible buffer remains untouched.

And, again, after a **flip** the contents of the back buffer and the visible buffer are identical. Remember that this is only an illustration of the mechanism. The middleware of your STB may use a different mechanism.



A **clear** will erase the back buffer to be empty and fully transparent:

But it will only be effectuated (i.e. made visible) after a flip:


flip