CloudTV Nano SDK (C++ Northbound API)

4.4

Generated by Doxygen 1.8.5

Mon Nov 6 2017 10:23:13

Contents

Chapter 1

Introduction

Figure 1.1: width=150px

1.1 Summary

The CloudTV™ Nano SDK can set up and control CloudTV sessions, providing the same capabilities across many different device types. The CloudTV Nano SDK supports cable set-top boxes as well as IP set-top boxes.

The CloudTV™ Nano SDK can be provided as a compiled software library, offering client APIs that a controlling Application (Guide/Middleware) can use to set up a CloudTV session. A device abstraction layer implements all device specific functionality. Porting to a specific device is done by implementing the device abstraction or Device Porting Layer for that device.

The Nano Client family of SDK's support the BCP and RFB-TV protocols. BCP is intended for cable systems with limited return paths (ALOHA). The RFB-TV protocol is intended for faster return paths. There are more features in the RFB--TV protocol, so the APIs are different between the two protocols. Therefore the APIs are split into two separate SDK versions. This SDK — the CloudTV Nano SDK — is written in C++ and only supports the RFB-TV protocol.

1.2 Copyright

1.3 Other Restricted Rights

1.4 Contact

1.4.1 ActiveVideo — Main Headquarters

333 W. San Carlos St.

Suite 400

San Jose, CA 95110

United States

Toll-free: 1-800-926-8398

Main: 1-408-931-9200

Fax: 1-408-931-9100

e-mail: info@activevideo.com

1.4.2 ActiveVideo — European Headquarters

Joop van den Endeplein 1

Mediacentrum 3745

1217 WJ Hilversum

The Netherlands

Main: +31 (0)35 6774131

Chapter 2

Todo List

Class IOverlayCallbacks

Implement the methods of IOverlayCallbacks in your own derived class.

Class IStreamPlayer

Implement the methods of IStreamPlayer in your own derived class.

Class Session::ISessionCallbacks

Implement the methods of Session::ISessionCallbacks in your own derived class.

Chapter 3

Hierarchical Index

3.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 4

Data Structure Index

4.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 5

Data Structure Documentation

5.1 ICdmSession::ICallback Struct Reference

Callback interface to indicate asynchronous events from the CdmSession object.

Public Types

- enum TerminateReason {
  TERMINATE_USER_STOP, **TERMINATE_END_OF_STREAM**,
  TERMINATE_LICENSE_EXPIRED, **TERMINATE_UNSPECIFIED** }

  Values for reason parameter in terminate_indication().

Public Member Functions

- virtual void terminate_indication (TerminateReason reason)=0

  Indicate termination of a CdmSession.
- virtual void setup_result (ICdmSession::SetupResult result, const std::map< std::string, std::string > &response)=0

  Report the result of the setup() call.
- virtual void terminate_result (const std::map< std::string, std::string > &stop_data)=0

  Report the result of the terminate() call.

5.1.1 Detailed Description

See Also

ICdmSession

5.1.2 Member Function Documentation

5.1.2.1 virtual void setup_result (**ICdmSession::SetupResult** *result*, const std::map< std::string, std::string > & *response*) `[pure virtual]`

Call this to pass the result of the *setup()* call back to the SDK when the setup is complete.

Parameters

in	*result*	SETUP_OK if successful or one of the other ICdmSession::SetupResult values otherwise.
in	*response*	The DRM-specific response data, packed as key-value pairs, as result of to the call to *setup()*.

5.1.2.2 virtual void terminate_indication (TerminateReason *reason*) [pure virtual]

This can be called by the CdmSession object at any time between a successful setup() and terminate(). The use of this callback is optional. It is intended to be used in case a running session suddenly gets into a (fatal) error state and the server needs to be signaled. The SDK will call terminate() in response and destroy the session afterward.

Parameters

in	*reason*	The reason why the termination is requested.

Note

The SDK may already have deleted the calling CdmSession object when terminate_indication() returns. The calling code must be aware of that and take appropriate precautions.

5.1.2.3 virtual void terminate_result (const std::map< std::string, std::string > & *stop_data*) [pure virtual]

Call this to pass the result of the *terminate()* call back to the SDK when the terminate is complete.

Parameters

in	*stop_data*	The DRM-specific stop data, packed as key-value pairs, as a result of to the call to *terminate()*.

5.2 IMediaPlayer::ICallback Struct Reference

Callback interface for player status updates.

Public Member Functions

- virtual void player_event (PlayerEvent event)=0

  Send a player event back to the stream originator.

5.2.1 Member Function Documentation

5.2.1.1 virtual void player_event (PlayerEvent *event*) [pure virtual]

Parameters

in	*event*	The event to pass.

5.3 ICdmSession Struct Reference

CDM session interface.

Data Structures

- struct ICallback

  Callback interface to indicate asynchronous events from the CdmSession object.

Public Types

- enum SetupResult {
  SETUP_OK, **SETUP_DRM_SYSTEM_ERROR**,
  SETUP_NO_LICENSE_SERVER, **SETUP_LICENSE_NOT_FOUND**,
  SETUP_UNSPECIFIED_ERROR }

  Result values of setup().

Public Member Functions

- virtual IStreamDecrypt ∗ get_stream_decrypt_engine ()=0

  Get a related stream decryption engine.
- virtual void setup (const std::string &session_type, const std::map< std::string, std::string > &init_data, ICallback &callback)=0

  Setup a new CdmSession.
- virtual void terminate (ICallback &callback)=0

  Terminate a CdmSession.

5.3.1 Detailed Description

A client only needs to implement this interface when CDM/DRM support is required. Object instances will be created by means of a call to ICdmSessionFactory::create().

Note

> It is up to the implementation to handle a call to *setup()* and *terminate()* asynchronous (i.e. non-blocking). However, it is strongly recommended to do so because otherwise a non-responsive or slow CDM/DRM server, or poor network conditions, will also block the handling of other RFB-TV protocol messages (like key presses).

5.3.2 Member Function Documentation

5.3.2.1 virtual IStreamDecrypt∗ get_stream_decrypt_engine ( ) `[pure virtual]`

Returns

Pointer to an instance of a decryption engine. Returning a NULL pointer indicates no decryption engine is available. This method allows passing a decryption engine that can be used to decrypt encrypted streams that are related to this CdmSession instance. The set-up and control of the stream decryption engine, as well as the decryption algorithm used is to be defined by the user. Returning a valid pointer makes sure that any encrypted stream is routed through the registered object for decryption. If a valid pointer is returned, it should remain valid until terminate() is called or until the CdmSession object is destroyed.

See Also

IStreamDecrypt

5.3.2.2 virtual void setup (const std::string & *session_type,* const std::map< std::string, std::string > & *init_data,* ICallback & *callback*) `[pure virtual]`

This is called exactly once for each CdmSession object, typically right after construction.

Note

It is highly recommended to process this call asynchronously (i.e. non-blocking) and post the resulting stop data by calling *setup_result()* once the session setup is complete.

Parameters

`in`	*session_type*	The session type. Currently, can only be "temporary".
`in`	*init_data*	The DRM-specific session initialization data, packed as key-value pairs.
`in`	*callback*	A callback object that the CdmSession must use to tell the result of the setup (or to indicate premature session termination).

5.3.2.3 virtual void terminate (ICallback & *callback*) `[pure virtual]`

This is called exactly once for each CdmSession object, typically before destruction. It must be possible, however, to delete a CdmSession object without terminate() having been called first.

Note

It is highly recommended to process this call asynchronously (i.e. non-blocking) and post the resulting stop data by calling *terminate_result()* once the session termination is complete.

Parameters

`in`	*callback*	A callback object that the CdmSession must use to tell the result of the terminate.

5.4 ICdmSessionFactory Struct Reference

CDM session object factory.

Public Member Functions

- virtual void get_drm_system_id (uint8_t(&id)[16])=0

  Return the DRM system ID of this CdmSessionFactory.
- virtual ICdmSession ∗ create ()=0

  Create a new instance of a CdmSession object.
- virtual void destroy (ICdmSession ∗cdm_session)=0

  Destroy a previously created instance of a CdmSession object.

5.4.1 Detailed Description

The ICdmSessionFactory is registered and bound to a specific DRM type. This allows the owner to create CdmSession instances for the proper DRM system when required. A client must register factories by calling Session::register_drm_-system().

See Also

ICdmSession

5.4.2 Member Function Documentation

5.4.2.1 virtual ICdmSession∗ create ( ) [pure virtual]

Returns

Pointer to a CdmSession object that implements ICdmSession.

Note

Deletion of the returned object will be done by calling destroy().

5.4.2.2 virtual void destroy (ICdmSession ∗ *cdm_session*) [pure virtual]

Free all related resources of the object that is pointed to by *cdm_session*, including any threads that may have been created to support asynchronous handling.

Parameters

in	*cdm_session*	Pointer to CdmSession object.

5.4.2.3 virtual void get_drm_system_id (uint8_t(&) *id[16]*) [pure virtual]

Parameters

out	*id*	The DRM system ID.

5.5 IControl Struct Reference

Control interface for session control events.

Public Member Functions

- virtual void initiate (const std::string &host, const std::string &url, uint32_t screen_width, uint32_t screen_height, const std::map< std::string, std::string > &optional_parameters)=0

  Initiate the client session with the remote server host and start the application. The session will be in STATE_CONNECTING until the session is fully set up.
- virtual void terminate ()=0

  Stop the session and disconnect from the server.
- virtual void suspend ()=0

  Suspend the session and disconnect from the server.
- virtual void resume ()=0

  Connect to the server and resume the suspended session.
- virtual void update_session_optional_parameters (const std::map< std::string, std::string > &key_value_pairs)=0

  Update a number of session setup parameter key-value pairs at once. May be called when a session is active. The existing parameters are updated and an update message is sent to the server for those parameters that have their value changed.

5.5.1 Member Function Documentation

5.5.1.1 virtual void initiate (const std::string & *host,* const std::string & *url,* uint32_t *screen_width,* uint32_t *screen_height,* const std::map< std::string, std::string > & *optional_parameters*) `[pure virtual]`

Parameters

`in`	*host*	The remote host URL, e.g. "rfbtv://127.0.0.1:8095".
`in`	*url*	The application URL, e.g. "ctvprogram:youtube".
`in`	*screen_width*	The width of the client screen in pixels.
`in`	*screen_height*	The height of the client screen in pixels.
`in`	*optional_-parameters*	A map of key-value pairs that will be added to the session setup message.

Note

This is a non-blocking call. The session is handled in its own thread. Check the actual status with the get_state() method. When one or more mandatory properties are not set or an invalid host URL is given then the session will not be set up, even though this method initially returns success (true).

Possible parameter names and their values:

- "lang" The natural language to use by the UI application. The format is an IETF language tag, e.g. "en". If not specified, the application will use a default language.
- "lan" Type of network connection that the client is using. Valid values are: "wlan", "eth", "eth10", "eth100", "eth1000" and "LSC".
- "fw" The firmware version running on the device, e.g. "1.3.2.300".
- "configured_display" Preferred display, typically used to indicate to the server that an SD screen is connected. Example value: "pal4x3".

Please refer to the documentation of the underlying protocol for details.

5.5.1.2 virtual void resume ( ) [pure virtual]

Reconnects to the remote server and attempts to resume the session with the session identification that was saved when resume() was called.

See Also

suspend()

5.5.1.3 virtual void suspend ( ) [pure virtual]

Notifies the remote server that the client wishes to suspend the session.

See Also

resume()

5.5.1.4 virtual void terminate ( ) [pure virtual]

See Also

initiate()

5.5.1.5 virtual void update_session_optional_parameters (const std::map< std::string, std::string > & *key_value_pairs*) [pure virtual]

Note

If multiple parameters are changed while a session is active, it is preferred to call this method once rather than issuing multiple calls.

Parameters

in	*key_value_pairs*	A number of key-value pairs to be set or updated. See *initiate()* for further info about individual keys and values for optional parameters.

See Also

initiate()

5.6 IDefaultProtocolHandler Struct Reference

Default handler of non-managed RFB-TV Protocol extensions.

Public Member Functions

- virtual void received (const std::string &protocol_id, const uint8_t ∗data, uint32_t length)=0

  Handle an incoming request for all non-handled extensions.

5.6.1 Member Function Documentation

5.6.1.1 virtual void received (const std::string & *protocol_id*, const uint8_t $*$ *data*, uint32_t *length*) `[pure virtual]`

Parameters

in	*protocol_id*	Protocol identified of the received message
in	*data*	Pointer to buffer with received data.
in	*length*	Length of *data* in bytes.

5.7 IHandoffHandler Struct Reference

RFB-TV Session handoff handling interface.

Public Types

- enum HandoffResult {
HANDOFF_SUCCESS, **HANDOFF_UNSUPPORTED_URI**,
HANDOFF_FAILED_TO_DESCRAMBLE_STREAM, **HANDOFF_FAILED_TO_DECODE_STREAM**,
HANDOFF_NO_TRANSPORT_STREAM_WITH_INDICATED_ID, **HANDOFF_NO_NETWORK_WITH_INDIC-ATED_ID**,
HANDOFF_NO_PROGRAM_WITH_INDICATED_ID, **HANDOFF_PHYSICAL_LAYER_ERROR**,
HANDOFF_REQUIRED_MEDIA_PLAYER_ABSENT, **HANDOFF_ERRONEOUS_REQUEST**,
HANDOFF_ASSET_NOT_FOUND, **HANDOFF_TRANSPORT_LAYER_ERROR**,
HANDOFF_PLAYER_ERROR, **HANDOFF_APP_NOT_FOUND**,
HANDOFF_UNSPECIFIED_ERROR }

  Result values of handoff_request().

Public Member Functions

- virtual HandoffResult handoff_request (const std::string &scheme, const std::string &uri, bool resume_session_-when_done)=0

  Handle a hand off request to an internal app, like video on demand.

5.7.1 Member Function Documentation

5.7.1.1 virtual **HandoffResult** handoff_request (const std::string & *scheme,* const std::string & *uri,* bool *resume_session_when_done*) `[pure virtual]`

Parameters

in	*scheme*	The scheme that this handler was registered with. Passing the scheme back to the handler enables registering the same handler for multiple schemes if this is desirable. Otherwise, this parameter can be ignored.
in	*uri*	Uniform Resource Indicator that the handoff is supplied with, application specific.
in	*resume_session-_when_done*	Resume session after playback has finished. (The session will be suspended in that case.)

Returns

HANDOFF_SUCCESS if successful, another IHandoffHandler::HandoffResult value otherwise.

Note

Not supported by all protocol versions and depending on platform application.

5.8 IInput Struct Reference

Input interface for key and pointer events.

Public Types

- enum Action {
  ACTION_NONE, ACTION_DOWN,
  ACTION_UP, ACTION_KEYINPUT,
  ACTION_DOWN_AND_UP }

  Values for action parameter in send_keycode() or send_pointer_event().
- enum Button {
  NO_BUTTON, LEFT_BUTTON,
  RIGHT_BUTTON, MIDDLE_BUTTON,
  WHEEL_UP, WHEEL_DOWN }

  Values for button parameter in send_pointer_event().

Public Member Functions

- virtual void send_keycode (int key, Action action, bool &client_must_handle_key_code)=0

  Send key code to the server.
- virtual void send_pointer_event (uint32_t x, uint32_t y, Button button, Action action)=0

  Send pointer event to the server.

5.8.1 Member Enumeration Documentation

5.8.1.1 enum Action

Enumerator

ACTION_NONE No buttons or keys were pressed.

ACTION_DOWN Button or key was pressed.

ACTION_UP Button or key was released.

ACTION_KEYINPUT Character has been generated. This is only applicable from RFB-TV 2.0 onwards.

ACTION_DOWN_AND_UP Button or key was pressed and released.

5.8.1.2 enum Button

Enumerator

NO_BUTTON No button has changed state.

LEFT_BUTTON Left button has changed state.

RIGHT_BUTTON Right button has changed state.

MIDDLE_BUTTON Middle button has changed state.

WHEEL_UP Wheel button has changed state upward.

WHEEL_DOWN Wheel button has changed state downward.

5.8.2 Member Function Documentation

5.8.2.1 virtual void send_keycode (int *key*, Action *action*, bool & *client_must_handle_key_code*) [pure virtual]

Parameters

in	*key*	The native remote control value.
in	*action*	The Action value.
out	*client_must_-handle_key_code*	true if the client needs to handle the key, false otherwise.

If the key code map was initialized, then the *key* will be translated.

Note

If your platform is unable to distinguish the difference between a pressed and a released key, then call this method with ACTION_DOWN_AND_UP.
The value returned in *client_must_handle_key_code* depends on the state of the key filter. The key filter is updated by server commands from the platform. This happens asynchronously, so there is always a 'window' where a new application is entered on the platform where the key filter may not yet be updated while the user presses a key. As a result, this key is sent to the platform application instead of being handled locally or vice-versa.

See Also

X11KeyMap

5.8.2.2 virtual void send_pointer_event (uint32_t *x*, uint32_t *y*, Button *button*, Action *action*) [pure virtual]

Parameters

in	*x*	The X-coordinate
in	*y*	The Y-coordinate
in	*button*	Button or wheel that changed state.
in	*action*	Action that indicates the type of state change.

Note

If only a pointer move event needs to be sent, *button* should be NO_BUTTON and *action* should be ACTION_NO-NE

5.9 IMediaChunkAllocator Struct Reference

Abstract interface class for allocating chunks of media memory.

Public Member Functions

- virtual uint32_t get_chunk_size () const =0

  Get the chunk size for this allocator.
- virtual uint8_t ∗ alloc_chunk ()=0

  Allocate a single chunk of memory.
- virtual void free_chunk (uint8_t ∗p)=0

  Free a previously allocated chunk of media memory.

5.9.1 Detailed Description

IMediaChunkAllocator is an interface to an allocator of chunks of memory to be used by the media store. The memory is typically allocated more toward the early life of the object and tends to be less after. The memory is typically only freed near the very end of the object's lifetime. All allocated chunks have the same size, but this size can be determined by the implementation.

5.9.2 Member Function Documentation

5.9.2.1 virtual uint8_t∗ alloc_chunk ( ) `[pure virtual]`

Returns

Pointer to the allocated chunk or null if no memory is left.

Allocates single chunk of media memory.

5.9.2.2 virtual void free_chunk (uint8_t ∗ *p*) `[pure virtual]`

Parameters

`in`	*p*	Pointer to the previously allocated chunk.

Returns

void

Frees a previously allocated chunk of media memory. A freed chunk is no longer accessed by the system.

5.9.2.3 virtual uint32_t get_chunk_size ( ) const `[pure virtual]`

Returns

The chunk size.

Gets the fixed chunk size for this allocator. The chunk size should never change during the lifetime of the object. It should be a natural chunk size that optimizes performance with respect to memory access such as copies while keeping the memory overhead limited. Memory overhead occurs when storing small media segments using big chunks.

5.10 IMediaPlayer Struct Reference

Player for content streams i.e. various sources of streaming video.

Data Structures

- struct ICallback

  Callback interface for player status updates.
- struct PlayerInfo

  Struct that contains player information (state and statistics) to be passed back to the stream originator.

Public Types

- enum PlayerEvent {
PLAYER_STARTING, PLAYER_STARTED,
PLAYER_STOPPED, PLAYER_BUFFER_UNDERRUN,
PLAYER_BUFFER_OVERRUN, PLAYER_RECOVERABLE_ERROR,
PLAYER_UNRECOVERABLE_ERROR, PLAYER_DESCRAMBLE_ERROR,
PLAYER_DECODE_ERROR, PLAYER_TRANSPORT_STREAM_ID_ERROR,
PLAYER_NETWORK_ID_ERROR, PLAYER_PROGRAM_ID_ERROR,
PLAYER_PHYSICAL_ERROR }

  Player event definition.

Public Member Functions

- virtual ResultCode open_stream (const std::string &uri, const std::map< std::string, std::string > &stream_-params, IStream &stream_out, IStream ∗&stream_in)=0

  Called when streaming content should be opened (setup). If the media player will pass the stream through the SDK, the stream_in parameter should be filled in with the return path (i.e. the interface of the stream player). The stream_out parameter contains the SDK object the loaded stream can be sent to. If no routing is supported, stream_in must either be left unchanged or set to 0.
- virtual void close_stream ()=0

  Called when the library wishes to stop the content.
- virtual void get_player_info (PlayerInfo &info)=0

  Obtain player state and statistics.
- virtual void register_callback (ICallback ∗callback)=0

  Register a callback interface.

Static Public Attributes

- static const ResultCode CABLE_TUNING_ERROR

  There was a tuning error when trying to tune to a channel.
- static const ResultCode CONNECTION_FAILED

  Connection to a remote host could not be established.

5.10.1 Detailed Description

Subclass the IMediaPlayer to implement a media player that can resolve a media URI. This is then bound to a particular type of content with Session::register_media_player(). A MediaPlayer is given URIs to content streams, and so must be able both to retrieve the indicated resource and consume its content.

5.10.2 Member Enumeration Documentation

5.10.2.1 enum PlayerEvent

Enumerator

PLAYER_STARTING The player just started. This event should be sent as soon as the start() method of the player is called.

PLAYER_STARTED The player started. This event must be sent as soon as the first decoded frame is displayed (or as near as possible). Sent in response to a call to start().

PLAYER_STOPPED The player stopped. This event must be sent as soon as the last decoded frame was displayed (or as near as possible). Sent in response to a call to stop(), if started. May also be sent upon the call to register_callback().

PLAYER_BUFFER_UNDERRUN The player experienced a buffer underrun. Non-fatal. Once the underrun condition has stopped, the player should resume normal, minimal-latency decoding and the PLAYER_STARTED event MUST be sent.

PLAYER_BUFFER_OVERRUN The player experienced a buffer overrun. Fatal. The player can stop playing. It should expect a successive call to stop().

PLAYER_RECOVERABLE_ERROR The player experienced an error that is recoverable. Non-fatal. After recovery, the player should continue normal, minimal-latency decoding and the PLAYER_STARTED event MUST be sent.

PLAYER_UNRECOVERABLE_ERROR The player experienced an error that is unrecoverable. Fatal. The player can stop playing. It should expect a successive call to stop().

PLAYER_DESCRAMBLE_ERROR There was an error descrambling the stream. Fatal. The player can stop playing. It should expect a successive call to stop().

PLAYER_TRANSPORT_STREAM_ID_ERROR !< The client failed to decode the stream. Fatal.

PLAYER_NETWORK_ID_ERROR !< No transport stream with the indicated Transport Stream ID was found. Fatal.

PLAYER_PROGRAM_ID_ERROR !< No network with the indicated Network ID was found. Fatal.

PLAYER_PHYSICAL_ERROR !< No program with the indicated Program ID was found. Fatal. !< Unrecoverable error at the physical layer. Fatal.

5.10.3 Member Function Documentation

5.10.3.1 virtual void get_player_info (**PlayerInfo** & *info*) `[pure virtual]`

Parameters

`in`	*info*	Player state and statistics information to be returned. Fields that can be filled should be set by the player implementation; fields for which information cannot be obtained should be left alone. This way, the caller knows which fields have been set and which not.

Note

This call always succeeds.

5.10.3.2 virtual ResultCode open_stream (const std::string & *uri,* const std::map< std::string, std::string > & *stream_params,* **IStream** & *stream_out,* **IStream** ∗& *stream_in*) `[pure virtual]`

Parameters

in	*uri*	URI to open.
in	*stream_params*	Stream parameters that the player can use to check if playback is possible or not. In general, the stream parameters should be regarded as a hint, the stream itself is always leading. However, some applications require certain stream parameters to be processed for proper operation. This is application-specific. Valid parameters are documented in the RFB-TV specification (section "Optional stream parameters"), e.g. "video_width", "audio_codec" or "ca_data". RFB-TV 1.3.2 streaming parameters are mapped to the keys/values defined in RFB-TV 2.0.9.
in	*stream_out*	The IStream object the loaded stream must be sent to.
out	*stream_in*	The IStream object that will receive the processed stream.

Returns

ResultCode

5.10.3.3 virtual void register_callback (**ICallback** ∗ ***callback***) [pure virtual]

Parameters

in	*callback*	This object is to be used to signal playback events to. Passing a null pointer will unregister the callback.

5.11 IMediaPlayerFactory Struct Reference

Class to create a specific media player.

Public Member Functions

- virtual IMediaPlayer ∗ create ()=0

  Create a new instance of a media player object.
- virtual void destroy (IMediaPlayer ∗p)=0

  Destroy a previously created instance of a media player object.

5.11.1 Member Function Documentation

5.11.1.1 virtual **IMediaPlayer**∗ create ( ) [pure virtual]

Returns

Pointer to media player.

Note

Deletion of the returned object will be done by calling destroy().

5.11.1.2 virtual void destroy (**IMediaPlayer** ∗ ***p***) [pure virtual]

Parameters

in	*p*	Pointer to media player.

5.12 IOverlayCallbacks Struct Reference

Callback interface for overlay images in graphics overlay plane.

Public Member Functions

- virtual void overlay_blit_image (const PictureParameters &picture_params)=0

  Blit an image to the shadow graphics overlay plane.
- virtual void overlay_clear ()=0

  Clear the shadow buffer.
- virtual void overlay_flip ()=0

  Copy the shadow graphics overlay plane to the visible graphics overlay plane.

5.12.1 Detailed Description

Todo Implement the methods of IOverlayCallbacks in your own derived class.

Note

All overlay handling is done in a separate thread inside the Nano SDK. So your code must be prepared to receive calls to the methods in the IOverlayCallbacks interface to arrive in the context of a thread that is *different* from the main thread (i.e. the thread that the operating system uses to call your client's 'main()' function). When called, the overlay_blit_image(), overlay_clear() and overlay_flip() methods **must** block until all (graphics) processing has completed. This is necessary for the 'throttling' mechanism to work: It ensures that your Set-top Box does not get flooded with overlay images for the framebuffer updates.

Examples:

Application.h.

5.12.2 Member Function Documentation

5.12.2.1 virtual void overlay_blit_image (const **PictureParameters** & *picture_params*) [pure virtual]

The shadow graphics overlay plane is not visible until *overlay_flip()* is called.

Parameters

in	*picture_params*	PictureParameters with picture data, x and y coordinates, width and height and alpha channel information.

5.12.2.2 virtual void overlay_clear ( ) [pure virtual]

The buffer itself can remain intact, only the content has to be wiped (e.g, set to black and full transparency).

5.13 IProtocolExtension Struct Reference

RFB-TV Protocol extension interface.

Data Structures

- struct IReply

    Reply interface for the protocol extension to send messages to.

Public Member Functions

- virtual std::string get_protocol_id () const =0

    Return the protocol identifier of this extension.
- virtual void received (const uint8_t ∗data, uint32_t length)=0

    Handle an incoming request for this extension.
- virtual void register_reply_path (IReply ∗reply_path)=0

    Register a reply path for this protocol extension.

5.13.1 Member Function Documentation

5.13.1.1 virtual std::string get_protocol_id ( ) const [pure virtual]

Returns

Protocol identifier.

5.13.1.2 virtual void received (const uint8_t ∗ *data*, uint32_t *length*) [pure virtual]

Parameters

in	*data*	Pointer to buffer with received data.
in	*length*	Length of *data* in bytes

5.13.1.3 virtual void register_reply_path (IReply ∗ *reply_path*) [pure virtual]

Parameters

in	*reply_path*	Pointer to object instance of type IReply.

5.14 IProtocolExtension::IReply Struct Reference

Reply interface for the protocol extension to send messages to.

Public Member Functions

- virtual void send (const IProtocolExtension &origin, const uint8_t ∗data, uint32_t length)=0

    Send an outgoing message for this extension.

5.14.1 Member Function Documentation

5.14.1.1 virtual void send (const IProtocolExtension & *origin,* const uint8_t ∗ *data,* uint32_t *length*) `[pure virtual]`

Parameters

`in`	*origin*	Reference to object that implements this IProtocolExtension.
`in`	*data*	Pointer to data to be sent.
`in`	*length*	Number of bytes in *data*.

5.15 Session::ISessionCallbacks Struct Reference

Session callback interface. A client implementation has to implement these callbacks if it wants to be notified of relevant Session state changes.

Public Member Functions

- virtual void state_update (State state, ClientErrorCode error_code)=0

    This is called to notify the recipient of a state change of the session. This can be, but does not have to be, related to a call to one of the IControl or IInput methods.

5.15.1 Detailed Description

Todo Implement the methods of Session::ISessionCallbacks in your own derived class.

Examples:

Application.h.

5.15.2 Member Function Documentation

5.15.2.1 virtual void state_update (State *state,* ClientErrorCode *error_code*) `[pure virtual]`

Parameters

`in`	*state*	The new state of the session. This value would match the state returned by get_state() until the next call to state_update(). If the session is closed, the session is either in STATE_DISCONNECTED or STATE_ERROR.

in	*error_code*	Error code as documented in 'CloudTV Client Error Code Specification' version 1.4. This only has a meaning in STATE_DISCONNECTED or STATE_ERROR. An error code of CLIENT_ERROR_CODE_OK means no error, so this indicates normal session termination. The client should retune to whatever was running before the session started, e.g. to the last known TV channel. An error code of CLIENT_ERROR_CODE_OK_AND_DO_NOT_RETUNE is special: it also indicates 'no error' but the client should not retune after having closed the session; rather it should stay tuned to whatever was showing when the session was still active.

Note

In STATE_DISCONNECTED or STATE_ERROR, the remote server has indicated that the session has ended with *error_code*. Session termination error codes that have to be presented to the user, for example by means of an on-screen message dialog. The error codes are described in detail in the platform troubleshooting guide.

Examples:

Application.h.

5.16 IStream Struct Reference

Abstract interface class for passing stream data.

Public Member Functions

- virtual void stream_data (const uint8_t ∗data, uint32_t length)=0

  Callback to send the stream buffer back to the caller.
- virtual void stream_error (ResultCode result)=0

  Callback to indicate an error downloading the stream.

5.16.1 Member Function Documentation

5.16.1.1 virtual void stream_data (const uint8_t ∗ *data,* uint32_t *length*) `[pure virtual]`

Parameters

in	*data*	Stream data.
in	*length*	Length of the stream data.

Returns

void

Implemented in StreamForwarder.

5.16.1.2 virtual void stream_error (ResultCode *result*) `[pure virtual]`

Parameters

in	*result*	Result code indicating any problem. If ResultCode::SUCCESS, the stream is terminated without an error and no further calls to stream_data() are expected

Returns

void

Implemented in StreamForwarder.

5.17 IStreamDecrypt Struct Reference

This interface offers the functionality to decrypt a stream with given key identifier and initialization vector.

Public Member Functions

- virtual void set_stream_return_path (IStream ∗stream_out)=0

  Set the stream return path.
- virtual void set_key_identifier (const uint8_t(&key_id)[16])=0

  Set the key identifier to use for decryption.
- virtual void set_initialization_vector (const uint8_t(&iv)[16])=0

  Set the initialization vector to use for decryption.
- virtual bool stream_data (const uint8_t ∗data, uint32_t length)=0

  Decrypt the stream using given key identifier and initialization vector.

5.17.1 Member Function Documentation

5.17.1.1 virtual void set_initialization_vector (const uint8_t(&) *iv[16]*) [pure virtual]

Parameters

in	*iv*	The initialization vector.

8 byte initialization vectors can be emulated by setting byte 8-15 to 0. If no initialization vectors are used, this method doesn't need to be called.

5.17.1.2 virtual void set_key_identifier (const uint8_t(&) *key_id[16]*) [pure virtual]

Parameters

in	*key_id*	The key identifier.

The license and key retrieval is left to the underlying DRM system.

5.17.1.3 virtual void set_stream_return_path (IStream ∗ *stream_out*) [pure virtual]

Parameters

`in`	*stream_out*	The return path that should be used, or 0.

The decrypted stream should be returned using the interface that is set here. The interface can be removed by setting a null pointer (and should be if the object receiving the stream is destroyed). If no output interface is set, the decrypted data may be dropped.

5.17.1.4 virtual bool stream_data (const uint8_t ∗ *data,* uint32_t *length*) `[pure virtual]`

Parameters

`in`	*data*	Pointer to the data to be decrypted.
`in`	*length*	The number of bytes to decrypt.

Returns

bool Returns true on success, false on failure.

set_key_identifier() and set_initialization_vector() must/will have been called at least once if the DRM scheme requires such. Multiple calls to stream_data() will update the internal (stream-specific) state. set_key_identifier() and set_-initialization_vector() may or may not be called between successive calls to stream_data(), as is defined by the stream. If called, this will signal a new decrypt state. The function returns true if decryption succeeded, and false if not. Possible errors could be: failure to set key identifier or initialization vector, uninitialized DRM system, absent or expired license and more.

Note

This method can (will) be called with *data=0* and *length=0* at regular intervals (typically every 20 milliseconds. This is done to drive specific crypto engines on specific clients.

5.18 IStreamLoader Struct Reference

Generic source for content streams i.e. various sources of streaming video.

Public Member Functions

- virtual ResultCode open_stream (const std::string &uri, IStream &stream_sink)=0

  Called when streaming content should be opened (setup).
- virtual void close_stream ()=0

  Called when the library wishes to stop the content.

5.18.1 Detailed Description

Subclass the IStreamLoader to implement a stream loader that can resolve a media URI and hand over a stream for further processing.

5.18.2 Member Function Documentation

5.18.2.1 virtual ResultCode open_stream (const std::string & *uri,* IStream & *stream_sink*) `[pure virtual]`

Parameters

`in`	*uri*	URI to open.
`in`	*stream_sink*	Reference to stream sink to receive the data from the opened stream.

Returns

ResultCode

5.19 IStreamPlayer Struct Reference

Stream Player interface. The stream player plays a stream (typically MPEG-2 TS) that enters through IStream.

Public Member Functions

- virtual ResultCode start ()=0

  Start the player.
- virtual void stop ()=0

  Stop the player.

5.19.1 Detailed Description

Todo Implement the methods of IStreamPlayer in your own derived class.

Examples:

Application.h.

5.19.2 Member Function Documentation

5.19.2.1 virtual ResultCode start ( ) `[pure virtual]`

Returns

ResultCode.

Note

Data may enter through IStream after start() has been called. Normally, data should not be received before start() has been called. However, the player should be able to handle this case, although its behavior is not defined.

5.19.2.2 virtual void stop ( ) `[pure virtual]`

Note

This call always succeeds. All resources in use by the player can be freed. The player must support stop() being called multiple times without an intermediate call to start(). Normally, data should not be received after stop() has been called. However, the player should be able to handle this case, although its behavior is not defined.

5.20 PictureParameters Struct Reference

Picture parameters used by the overlay callback functions.

Data Fields

- uint16_t x

  The x position on the screen where this picture should be positioned. Coordinate origin is upper left.
- uint16_t y

  The y position on the screen where this picture should be positioned. Coordinate origin is upper left.
- uint16_t w

  The width of the picture in pixels.
- uint16_t h

  The height of the picture in pixels.
- uint8_t alpha
- std::vector< uint8_t > m_data
- std::string m_url

  URL where the overlay was retrieved from. This may be empty if the overlay was transmited over RFB-TV.

5.20.1 Detailed Description

The *alpha* value shall be ignored by the client if the picture that is referred to includes an alpha channel or another transparency mechanism. Pictures shall overwrite image data including its alpha channel at the overlay plane.

Note

The (0,0) coordinate corresponds to the upper left corner. Prior to rendering a rectangle, the client shall clear the rectangular area.

Examples:

Application.cpp, and Application.h.

5.20.2 Field Documentation

5.20.2.1 uint8_t alpha

The picture-transparency value shall only be used if the picture-object-data does not include an alpha channel or another transparency mechanism. The picture-transparency parameter range is from 0 to 255, where 0 denotes complete transparency. Pictures shall overwrite image data from previous screen updates including its alpha channel at the overlay plane

5.20.2.2 std::vector<uint8_t> m_data

The image data. The picture-object encoding supports self-identifying picture formats such as JPEG (first byte is 0xFF), PNG (first byte is 0x89) and BMP (first two bytes for some flavors are 'BM' in ASCII).

5.21 IMediaPlayer::PlayerInfo Struct Reference

Struct that contains player information (state and statistics) to be passed back to the stream originator.

Data Fields

- uint64_t current_pts

  PTS value of the frame that is currently displayed on the screen, if available. This should be the PTS as present in the stream, in 90kHz ticks.

5.22 Session Class Reference

CloudTV Nano SDK session management.

Data Structures

- struct ISessionCallbacks

  Session callback interface. A client implementation has to implement these callbacks if it wants to be notified of relevant Session state changes.

Public Types

- enum State {
  STATE_DISCONNECTED = 1, STATE_CONNECTING = 2,
  STATE_CONNECTED = 4, STATE_SUSPENDED = 8,
  STATE_ERROR = 16 }

  Values returned by get_state().

Public Member Functions

- Session (ClientContext &context, ISessionCallbacks ∗session_callbacks, IOverlayCallbacks ∗overlay_callbacks)

  Constructs a new session object with references to the client context and callbacks.
- IControl & get_control () const

  Get reference to control component.
- IInput & get_input () const

  Get reference to input processing component.
- State get_state () const

  Get current session state.
- bool register_media_player (const std::string &protocol_id, IMediaPlayerFactory &factory)

  Bind a protocol to a content source for the loading of streams.
- bool unregister_media_player (const std::string &protocol_id)

  Un-bind a protocol from a content source for the loading of streams.
- bool register_content_loader (IContentLoader ∗content_loader)

  Register a content loader of static resources, such as images used for overlays. If the client does not register a content loader, then client side images will be received as in-band data in the RFB-TV protocol (provided both the cloud application and the client support overlays). If the client does register a content loader, then the server can decide to provide images by means of download URIs instead of in-band data.

- bool register_protocol_extension (IProtocolExtension &protocol_extension)

  Register a protocol extension.
- bool unregister_protocol_extension (IProtocolExtension &protocol_extension)

  Unregister a protocol extension.
- void register_default_protocol_handler (IDefaultProtocolHandler ∗protocol_handler)

  Register a protocol extension to handle non-registered protocols.
- void register_media_chunk_allocator (IMediaChunkAllocator ∗media_chunk_allocator)

  Register a chunked media memory allocator.
- bool register_drm_system (ICdmSessionFactory &factory)

  Register a DRM system in the form of an ICdmSessionFactory.
- bool unregister_drm_system (ICdmSessionFactory &factory)

  Un-register a DRM system.
- bool register_handoff_handler (const std::string &handoff_scheme, IHandoffHandler &handoff_handler)

  Register a session handoff handler with the Session object.
- bool unregister_handoff_handler (const std::string &handoff_scheme)

  Unregister a session handoff handler.

5.22.1 Detailed Description

Examples:

Application.cpp.

5.22.2 Member Enumeration Documentation

5.22.2.1 enum **State**

Enumerator

STATE_DISCONNECTED Disconnected.

STATE_CONNECTING Session is being set up.

STATE_CONNECTED Session is running.

STATE_SUSPENDED Suspended.

STATE_ERROR Unrecoverable error.

5.22.3 Constructor & Destructor Documentation

5.22.3.1 **Session** (ClientContext & *context*, **ISessionCallbacks** ∗ *session_callbacks*, **IOverlayCallbacks** ∗ *overlay_callbacks*)

Precondition

The client *context* must be initialized, because it is used to query the unique client identifier (serial number or MAC address), the STB vendor name and the STB model name.

Parameters

`in`	*context*	Reference to client context.
`in`	*session_-callbacks*	Pointer to object that implements the ISessionCallbacks interface.
`in`	*overlay_callbacks*	Pointer to object that implements the IOverlayCallbacks interface.

Note

Any of the pointers being NULL signals that the corresponding functionality is not implemented by the client.

5.22.4 Member Function Documentation

5.22.4.1 IControl& get_control ( ) const

Returns

IControl object

5.22.4.2 IInput& get_input ( ) const

Returns

IInput object

5.22.4.3 State get_state ( ) const

Returns

Session::State

5.22.4.4 bool register_content_loader (IContentLoader ∗ *content_loader*)

Parameters

`in`	*content_loader*	Object that implements IContentLoader interface.

Returns

True if the content_loader could be properly registered

5.22.4.5 void register_default_protocol_handler (IDefaultProtocolHandler ∗ *protocol_handler*)

Parameters

`in`	*protocol_handler*	Pointer to instance of a default protocol handler. Passing a NULL pointer un-registers the current protocol_handler.

This method registers a default receiver in case there is no registered IProtocolExtension object for a received message.

See Also

IProtocolExtension
IDefaultProtocolHandler

5.22.4.6 bool register_drm_system (ICdmSessionFactory & *factory*)

Parameters

in	*factory*	The CdmSession factory to register.

Returns

true if successful, false otherwise.

Note

Registering again for the same DRM system replaces the previous *factory* in the registry of the Session object.

5.22.4.7 bool register_handoff_handler (const std::string & *handoff_scheme,* IHandoffHandler & *handoff_handler*)

Parameters

in	*handoff_scheme*	Scheme to register *handoff_handler*, e.g. "vod". RFB-TV 2.0 defines "vod", "hls", "dash", "mss", "app", "url", 'rfbtv" and 'rfbtvs".
in	*handoff_handler*	IHandoffHandler to handle the handoff of given *handoff_scheme*.

Returns

true if successful, false otherwise.

Note

Registering again for the same handoff_scheme replaces the previous *handoff_handler* in the registry of the Session object.

5.22.4.8 void register_media_chunk_allocator (IMediaChunkAllocator $*$ *media_chunk_allocator*)

Parameters

in	*media_chunk_-allocator*	Pointer to an instance of a media memory allocator engine.

Passing a NULL pointer or a new allocator un-registers any current allocator, freeing up any memory allocated using the previously registered allocator, if any. The chunked media allocator will be used to allocate memory for the deep media buffer.

See Also

IMediaChunkAllocator

5.22.4.9 bool register_media_player (const std::string & *protocol_id,* IMediaPlayerFactory & *factory*)

Parameters

in	*protocol_id*	Protocol identifier to bind to *media_player*, e.g. "http".
in	*factory*	IMediaPlayerFactory to handle the download of *protocol_id*.

Returns

true if successful, false otherwise.

Note

Registering again for the same protocol, replaces the previous *factory* in the registry of the Session object.

5.22.4.10 bool register_protocol_extension (**IProtocolExtension** & *protocol_extension*)

Parameters

in	*protocol_-extension*	Reference to instance of a protocol extension.

Optionally register an instance of a protocol extension class. Do this to receive messages for the registered protocol in a class that is derived from IProtocolExtension and instantiated in your client.

See Also

IProtocolExtension

Returns

true if successful, false otherwise.

Note

Registering again for the same protocol, replaces the previous *protocol_extension* in the registry of the Session object.

5.22.4.11 bool unregister_drm_system (**ICdmSessionFactory** & *factory*)

Parameters

in	*factory*	The CdmSession factory to unregister.

Returns

true if successful, false otherwise.

Note

Unregistering any ICdmSessionFactory will close all active CDM sessions, if any.

5.22.4.12 bool unregister_handoff_handler (const std::string & *handoff_scheme*)

Parameters

`in`	*handoff_scheme*	Scheme to unregister, e.g. "vod".

Returns

true if successful, false otherwise.

5.22.4.13 bool unregister_media_player (const std::string & *protocol_id*)

Parameters

`in`	*protocol_id*	Protocol identifier to unbind, e.g. "http".

Returns

true if successful, false otherwise.

5.22.4.14 bool unregister_protocol_extension (IProtocolExtension & *protocol_extension*)

Parameters

`in`	*protocol_-extension*	Reference to instance of a protocol extension.

Call this when the client is no longer interested in messages for a specific protocol extension.

See Also

IProtocolExtension

Returns

true if successful, false otherwise.

5.23 StreamForwarder Class Reference

Helper class to forward a stream to a specified destination.

Public Member Functions

- ResultCode open (const std::string &url)

  Open a socket to forward a stream to.
- void close ()

  Close the socket.
- virtual void stream_data (const uint8_t ∗data, uint32_t length)

  Callback to send the stream buffer back to the caller.
- virtual void stream_error (ResultCode result)

  Callback to indicate an error downloading the stream.

Static Public Attributes

- static const ResultCode INVALID_URL

  An invalid URL was passed to open()
- static const ResultCode CANNOT_CREATE_FILE

  Cannot create file.

5.23.1 Detailed Description

Examples:

Application.h.

5.23.2 Member Function Documentation

5.23.2.1 ResultCode open (const std::string & *url*)

Parameters

in	*url*	Url like "udp://127.0.0.1:9990" or "file:///home/test/grab.ts"

Returns

error code.

5.23.2.2 virtual void stream_data (const uint8_t ∗ *data,* uint32_t *length*) [virtual]

Parameters

in	*data*	Stream data.
in	*length*	Length of the stream data.

Returns

void

Implements IStream.

5.23.2.3 virtual void stream_error (ResultCode *result*) [virtual]

Parameters

in	*result*	Result code indicating any problem. If ResultCode::SUCCESS, the stream is terminated without an error and no further calls to stream_data() are expected

Returns

void

Implements IStream.

Chapter 6

Example Documentation

6.1 Application.cpp

```
#include "Application.h"
#include "MediaChunkAllocator.h"

#include <core/IControl.h>
#include <core/IInput.h>
#include <stream/SimpleMediaPlayer.h>
#include <stream/HttpLoader.h>
#include <porting_layer/ClientContext.h>
#include <porting_layer/Keyboard.h>

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

using namespace ctvc;

Application::Application()
{
}

Application::~Application()
{
}

void Application::run(const std::string &server, const std::string &app_url, const std::string &forward_url
      )
{
    Session session(ClientContext::instance(), this, &m_overlay_callbacks);

    MediaChunkAllocator allocator;
    session.register_media_chunk_allocator(&allocator);

    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);

    session.register_protocol_extension(m_my_protocol_extension);

    std::map<std::string, std::string> optional_parameters;
    optional_parameters["lan"] = "eth10";

    m_stream_player.set_forward_url(forward_url);

    m_state_observer.set_states_to_wait_for(Session::STATE_CONNECTING,
      Session::STATE_DISCONNECTED | Session::STATE_ERROR);
    session.get_control().initiate(server, app_url, 1280, 720, optional_parameters);
    if (!m_state_observer.wait_for_states()) {
        CTVC_LOG_ERROR("Session initiate() failed");
    }

    while (true) {
        if (session.get_state() != Session::STATE_CONNECTING && session.get_state(
      ) != Session::STATE_CONNECTED) {
            break; // session closed or error: break loop
        }
        // handle key presses (the simple way)
        int key = Keyboard::get_key();
        if (key == 'q' || key == EOF) {
            printf("client terminates session\n");
            session.get_control().terminate();
            break;
        }
        if (key) {
            bool client_must_handle_key_code;
            session.get_input().send_keycode(key, IInput::ACTION_DOWN_AND_UP,
      client_must_handle_key_code);
            if (client_must_handle_key_code) {
                printf("client must handle the key\n");
            }
        }
    }

    printf("session closed\n");

    session.unregister_media_player("http");
    session.unregister_media_player("https");

    session.register_media_chunk_allocator(0);
}

void Application::state_update(Session::State state, ClientErrorCode reason)
{
    m_state_observer.state_update(state, 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("####################################################################\n");
        printf("TODO: show message in on-screen dialog to end-user, code:%d\n", reason);
        printf("      PRESS OK TO CONTINUE\n");
        printf("####################################################################\n");
        char c;
        if (read(0, &c, sizeof(c))) {
        }
    }
}

void Application::OverlayCallbacks::overlay_blit_image(const PictureParameters &/*
      picture_params*/)
{
    printf("TODO: OverlayCallbacks::overlay_blit_image()\n");
}

void Application::OverlayCallbacks::overlay_clear()
{
    printf("TODO: OverlayCallbacks::overlay_clear()\n");
}

void Application::OverlayCallbacks::overlay_flip()
{
    printf("TODO: OverlayCallbacks::overlay_flip()\n");
}

Application::StreamPlayer::StreamPlayer()
{
}

Application::StreamPlayer::~StreamPlayer()
{
}

void Application::StreamPlayer::set_forward_url(const std::string &forward_url)
{
    m_forward_url = forward_url;
}
```

```
ResultCode Application::StreamPlayer::start()
{
    printf("TODO: StreamPlayer::start()\n");

    if (!m_forward_url.empty()) {
        return m_forwarder.open(m_forward_url.c_str());
    }

    return ResultCode::SUCCESS;
}

void Application::StreamPlayer::stop()
{
    printf("TODO: StreamPlayer::stop()\n");
    m_forwarder.close();
}

void Application::StreamPlayer::stream_data(const uint8_t *data, uint32_t length)
{
    m_forwarder.stream_data(data, length);
}

void Application::StreamPlayer::stream_error(ResultCode error_code)
{
    m_forwarder.stream_error(error_code);
}
```

6.2 Application.h

```
#pragma once

#include "MyProtocolExtension.h"

#include <core/Session.h>
#include <core/SessionStateObserver.h>
#include <core/IOverlayCallbacks.h>

#include <stream/IStreamPlayer.h>
#include <stream/StreamForwarder.h>

#include <string>

class Application : public ctvc::Session::ISessionCallbacks
{
public:
    Application();
    virtual ~Application();

    void run(const std::string &server, const std::string &app_url, const std::string &forward_url);

protected:
    // Implement ISessionCallbacks
    void state_update(ctvc::Session::State state, ctvc::ClientErrorCode
      reason);

    // Implement the graphics overlay callbacks
    class OverlayCallbacks : public ctvc::IOverlayCallbacks
    {
        // overlay support
        virtual void overlay_blit_image(const ctvc::PictureParameters &
      picture_params);
        virtual void overlay_clear();
        virtual void overlay_flip();
    };

    class StreamPlayer : public ctvc::IStreamPlayer
    {
    public:
        StreamPlayer();
        ~StreamPlayer();
        void set_forward_url(const std::string &forward_url);
        ctvc::ResultCode start();
        void stop();
        void stream_data(const uint8_t *data, uint32_t length);
        void stream_error(ctvc::ResultCode error_code);
    private:
        std::string m_forward_url;
        ctvc::StreamForwarder m_forwarder;
    };

private:
    ctvc::SessionStateObserver m_state_observer;
    OverlayCallbacks m_overlay_callbacks;
    StreamPlayer m_stream_player;

    MyProtocolExtension m_my_protocol_extension;
};
```

6.3 main.cpp

```
#include "Application.h"

#include <porting_layer/ClientContext.h>
#include <porting_layer/X11KeyMap.h>
#include <porting_layer/Keyboard.h>

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

using namespace ctvc;

const char *DEFAULT_SERVER_URL = "rfbtv://127.0.0.1:8095";
const char *DEFAULT_APP_URL = "webkit:http://youtube.com/tv";
const char *DEFAULT_FORWARD_URL = "udp://127.0.0.1:12345";
const char *DEFAULT_BASE_STORE_PATH = "/tmp";

void usage(const char *name)
{
    fprintf(stderr, "Usage: %s [options]\n", name);
    fprintf(stderr, "\nAvailable options:\n");
    fprintf(stderr, " -h                     Print this help.\n");
    fprintf(stderr, " -s <server URL>        Connect to the specified RFB-TV server.
       default: '%s'\n", DEFAULT_SERVER_URL);
    fprintf(stderr, " -a <app URL>           Start the specified app on the server.
       default: '%s'\n", DEFAULT_APP_URL);
    fprintf(stderr, " -b <base store path>    Path to datastore files (i.e. cookie file).
       default: '%s'\n", DEFAULT_BASE_STORE_PATH);
    fprintf(stderr, " -f <forward URL>        Forward the received stream to the specified address.
       default: '%s'\n", DEFAULT_FORWARD_URL);
    fprintf(stderr, " -c <client certificate> Client certificate in PEM format.
       default: none\n");
    fprintf(stderr, " -k <private client key> Private key that signed the client certificate.
       default: none\n");
    fprintf(stderr, " -t <server certificate> Public server certificate.
       default: none\n");
    fprintf(stderr, "\nExample: %s -s rfbtv://localhost -a webkit:http://activevideo.com -f
       udp://127.0.0.1:9999\n", name);
}

static const char *get_unique_id()
{
    return "01:02:03:04:05:06";
}

void init_keymap(X11KeyMap &keymap)
{
    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...
}

void setup_client_context(const std::string &base_store_path, const std::string &ca_client_path, const
      std::string &private_key_path, const std::string &ca_path)
{
    ClientContext &client_context(ClientContext::instance());
    client_context.set_manufacturer("MyCompany");
    client_context.set_device_type("STB1234");
    client_context.set_unique_id(get_unique_id());

    client_context.set_base_store_path(base_store_path.c_str());

    client_context.set_ca_client_path(ca_client_path.c_str());
    client_context.set_private_key_path(private_key_path.c_str());
    client_context.set_ca_path(ca_path.c_str());

    init_keymap(client_context.get_keymap());
}

int main(int argc, char *argv[])
{
    Application app;

    // Connect to server-simulator on localhost by default
    std::string server(DEFAULT_SERVER_URL);
    std::string app_url(DEFAULT_APP_URL);
    std::string base_store_path(DEFAULT_BASE_STORE_PATH);
    std::string forward_url(DEFAULT_FORWARD_URL);

    std::string ca_client_path;
    std::string private_key_path;
    std::string ca_path;

    int opt;
    while ((opt = getopt(argc, argv, "hs:a:b:f:c:k:t:")) != -1) {
        switch (opt) {
        case 's':
            server = optarg;
            break;
        case 'a':
            app_url = optarg;
            break;
        case 'b':
            base_store_path = optarg;
            break;
        case 'f':
            forward_url = optarg;
            break;
        case 'c':
            ca_client_path = optarg;
            break;
        case 'k':
            private_key_path = optarg;
            break;
        case 't':
            ca_path = optarg;
            break;
        case 'h':
            usage(argv[0]);
            return 0;
        default: /* '?' */
            usage(argv[0]);
            return 1;
        }
    }

    if (optind != argc) {
        // Print usage and exit if we have any non-option arguments
        usage(argv[0]);
        return 1;
    }

    setup_client_context(base_store_path, ca_client_path, private_key_path, ca_path);

    app.run(server, app_url, forward_url);

    return 0;
}
```

Index