Qualcomm Technologies International, Ltd.

A2DP Library

User Guide

80-CT401-1 Rev. AU

October 31, 2017

Confidential and Proprietary – Qualcomm Technologies International, Ltd.

NO PUBLIC DISCLOSURE PERMITTED: Please report postings of this document on public servers or websites to DocCtrlAgent@qualcomm.com.

Restricted Distribution: Not to be distributed to anyone who is not an employee of either Qualcomm Technologies International, Ltd. or its affiliated companies without the express approval of Qualcomm Configuration Management.

Not to be used, copied, reproduced, or modified in whole or in part, nor its contents revealed in any manner to others without the express written permission of Qualcomm Technologies International, Ltd.

CSR chipsets are products of Qualcomm Technologies International, Ltd. Other Qualcomm products referenced herein are products of Qualcomm Technologies International, Ltd.

Qualcomm is a trademark of Qualcomm Incorporated, registered in the United States and other countries. CSR is a trademark of Qualcomm Technologies International, Ltd., registered in the United States and other countries. Other product and brand names may be trademarks or registered trademarks of their respective owners.

This technical data may be subject to U.S. and international export, re-export, or transfer ("export") laws. Diversion contrary to U.S. and international law is strictly prohibited.

Qualcomm Technologies International, Ltd. (formerly known as Cambridge Silicon Radio Limited) is a company registered in England and Wales with a registered office at: Churchill House, Cambridge Business Park, Cowley Road, Cambridge, CB4 0WZ, United Kingdom.
Registered Number: 3665875 | VAT number: GB787433096

Revision history

Revision	Date	Description
1	NOV 2007	Original publication of this document. Alternative document number CS-00116660-UG.
2	OCT 2009	Updated to latest style guidelines.
3	SEP 2010	Amended to reflect updated A2DP library.
4	JUL 2011	Updated for ADK 1.0 Updated to latest style.
5	JAN 2012	Updated to latest CSR™ style.
6	DEC 2012	Updated for ADK 2.5
7	MAY 2015	Updated to latest CSR style
8	MAY 2015	Minor editorial correction
9	APR 2016	Updated to conform to QTI standards; No technical content was changed in this document revision
10	MAR 2017	Formatting corrections.
11	APR 2017	Final edits
12-15	APR 2017	Internal only releases
16	MAY 2017	Updated reference to *An Introduction to Audio Plugins*.
AU	OCT 2017	Added to Content Managemant system. DRN updated to use Agile numbering. No technical changes.

Contents

Figures

1 A2DP library - overview

The A2DP library provides a simple Application Programming Interface (API) that is used to configure application behavior, handle the signaling procedures required to perform stream negotiation and establishment, without the need to edit library source code.

Refer to

- `a2dp.h` API documentation for detailed information about the A2DP library function calls and message structures.
- *Audio/Video Distribution Transport Protocol Specification*
- *Generic Audio/Video Distribution Profile Specification*
- *Advanced Audio Distribution Profile Specification*

Single instance operation

The A2DP library is designed to operate from a single instance, which includes support for connections to multiple devices (multipoint).

The library only requires a single slot to hold all the data necessary to manage itself. The use of Dynamic Memory (Slots) has been kept to a minimum, but one extra slot per Signaling connection may be required, to store any fragmented AVDTP packets.

NOTE This slot use is only relevant when the thread of control is with the A2DP library and would effectively be shared with other libraries, above the Connection library.

2 Initialization of the A2DP library

The application only needs one instance of the A2DP library, which must be initialized before it can be used, further instances cannot be initialized and any attempt to do so is ignored.

Figure 2-1 Initialization

```
#include <a2dp.h>

/* Default capability header files will be included as part of the SDK */
#include "a2dp_default_mp3_sink.h"
#include "a2dp_default_sbc_sink.h"

/* Link loss timeout, in seconds. */
#define LINKLOSS_TIMEOUT       20

/* Stream-End Point IDs. Must be unique for each SEP. */
#define MP3_SEID                    1
#define SBC_SEID                    2

/* Resource ID for Kalimba. If a SEP becomes in use then all SEPs with the
same resource ID will also be in use. */
#define KALIMBA_RESOURCE_ID            1

/* Let the A2DP library select the optimal codec settings based on the
remote and local capabilities, when this device initiates a connection. /*
#define LIBRARY_SELECTS_CODEC_SETTINGS    1

/* Use the default flush timeout values for this SEP */
#define FLUSH_TIMEOUT                0

/* Set up the first SEP which uses the MP3 codec with default capabilities.
It takes the Sink role. */
static const sep_config_type mp3_sep = { MP3_SEID, KALIMBA_RESOURCE_ID,
sep_media_type_audio, a2dp_sink, LIBRARY_SELECTS_CODEC_SETTINGS,
FLUSH_TIMEOUT, sizeof(mp3_caps_sink), mp3_caps_sink };

/* Set up the second SEP which uses the SBC codec with default
capabilities. It takes the Sink role.*/
static const sep_config_type sbc_sep = { SBC_SEID, KALIMBA_RESOURCE_ID,
sep_media_type_audio, a2dp_sink, LIBRARY_SELECTS_CODEC_SETTINGS,
FLUSH_TIMEOUT, sizeof(sbc_caps_sink), sbc_caps_sink };
sep_data_type seps[2];

/* Initialise the first SEP */
seps[0].sep_config = &mp3_sep;
seps[0].in_use = FALSE;

/* Initialise the second SEP */
seps[1].sep_config = &sbc_sep;
seps[1].in_use = FALSE;

/* Initialise the A2DP library. */
A2dpInit(&theApp->task, A2DP_INIT_ROLE_SINK, 0, NULL, 2, seps,
LINKLOSS_TIMEOUT);
```

The initialization API is used to:

- Register the AVDTP PSM with the Connection library.
- Register one or more service records).

  NOTE Up to two service records can be registered by passing them in from the application. If no service records are supplied, the library uses one or more default service records depending on the roles passed in (that is, source, sink, or both source and sink).

- Format the SEP information to optimize use of memory resources and collate it into a format that the A2DP library can use.

The initialization function has the following format:

Use the `A2dpInit ()` function to initialize the profile library.

The application is expected to pass a task identifier, clientTask, to the A2DP library. This being the identifier of the task that receives the `A2DP_INIT_CFM` message when the library initialization has completed.

- This is also the task that sends indications of any incoming A2DP connections.

In the simplest use case, the library can be initialized just by setting the role parameter and passing in a valid `sep_data_type` structure. The A2DP library then uses a default service record based on the role selected. As the service record defined in the A2DP specification is generic, it is expected that most applications will set the `service_records` pointer to NULL and use the role parameter to select which records are registered.

If, however, an application wishes to register its own particular service record it can use the `service_records` pointer to pass in a maximum of two service records to the A2DP library. This registers the records but does not perform any sanity checks on their contents. It is the responsibility of the application to ensure that the records passed into the library are correct. If the application is using custom service records, it is expected that in most cases these will be stored as constant data. If the memory for the service record is dynamically allocated when a call to `A2dpInit` has been made all ownership of that memory is relinquished to the A2DP library. If the application is passing in a custom service record, it must set the role parameter to zero.

The initialization function also requires the application to provide a complete list of the SEPs that the application supports and all their parameters.

The library must be initialized with at least one SEP because support for the SBC codec is mandated by the A2DP specification. In most cases, the application uses the default parameters for a particular SEP so the codec plug-in header files contain the default service capabilities for a particular codec.

The order in which the SEPs are registered in the call to `A2dpInit` is also the default codec selection order when establishing a stream from the local device, see Connecting the A2DP channels). Ensure that the SBC codec is placed as the last SEP in the list.

The A2DP library returns an `A2DP_INIT_CFM` message to the application indicating the outcome of the initialization process. The example code section below shows how the A2DP library can be initialized by the application:

Figure 2-2 A2DP library initialization

```
#include <a2dp.h>

               / * The default capability header files will be included as
part of
               * the SDK.
               * /
               #include "a2dp_default_mp3_sink.h"
               #include "a2dp_default_sbc_sink.h"

               / * Link loss timeout, in seconds. * /
               #define LINKLOSS_TIMEOUT       20

               / * Stream-End Point IDs. Must be unique for each SEP. * /
               #define MP3_SEID                    1
               #define SBC_SEID                    2

               / * Resource ID for Qualcomm Kalimba. If a SEP becomes in
use then all SEPs
               * with the same resource ID will also be in use.
               * /
               #define KALIMBA_RESOURCE_ID            1

               / * Let the A2DP library select the optimal codec settings
based on
               * the remote and local capabilities, when this device
initiates a
               * connection.
               * /


               #define LIBRARY_SELECTS_CODEC_SETTINGS    1

               / * Use the default flush timeout values for this SEP * /
               #define FLUSH_TIMEOUT                0


               / * Set up the first SEP which uses the MP3 codec with
default
               *capabilities. It takes the Sink role.
               * /
               static const sep_config_type mp3_sep = {
               MP3_SEID,
               KALIMBA_RESOURCE_ID,
               sep_media_type_audio,
               a2dp_sink,
               LIBRARY_SELECTS_CODEC_SETTINGS,
               FLUSH_TIMEOUT,
               sizeof(mp3_caps_sink),
               mp3_caps_sink
               };

               / * Set up the second SEP which uses the SBC codec with
default
               * capabilities. It takes the Sink role.
               * /
               static const sep_config_type sbc_sep = {
               SBC_SEID,
               KALIMBA_RESOURCE_ID,
               sep_media_type_audio,
               a2dp_sink,
               LIBRARY_SELECTS_CODEC_SETTINGS,
               FLUSH_TIMEOUT,
               sizeof(sbc_caps_sink),
               sbc_caps_sink
               };

               sep_data_type seps[2];

               / * Initialise the first SEP * /
               seps[0].sep_config = &mp3_sep;
               seps[0].in_use = FALSE;


               / * Initialise the second SEP * /
               seps[1].sep_config = &sbc_sep;
               seps[1].in_use = FALSE;
```

```
/ * Initialise the A2DP library. * /
A2dpInit(
&theApp->task,
A2DP_INIT_ROLE_SINK,
NULL,
2,
seps,
LINKLOSS_TIMEOUT
);
```

3 Connecting the A2DP channels

The a2dp library facilitates the creation of the A2DP signaling and media channels as defined in the A2DP specification. The API has been designed to allow application writers maximum control over the creation of the signaling channels while also abstracting much of the lower level signaling details.

Outgoing A2DP connections

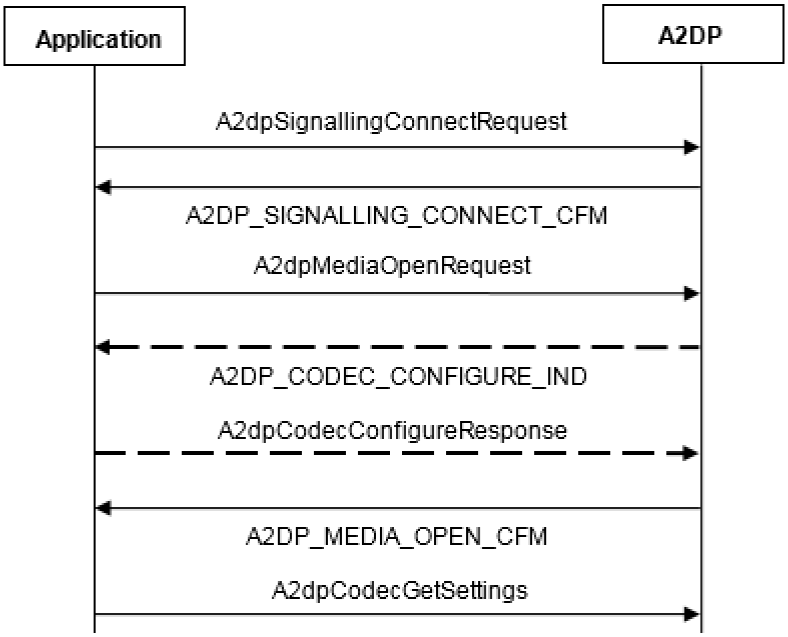

Figure 3-1 Creating signaling and media channels separately

NOTE Messages/functions with dashed arrows are only sent if the SEP is configured by the application in the SEP structure passed to `A2dpInit()` so that the A2DP library does not select the codec settings.

The application can initiate the establishment of just the signaling channel by using the `A2dpSignallingConnectRequest()` function.

This results in a single signaling channel established to the remote device whose Bluetooth address is specified in the function call. The client task, specified in the call to `A2dpInit()`, is the task that is registered to receive all further messages from the A2DP library relating to this signaling connection and any media connections that may subsequently be opened. In most cases this will be the application task that initiates the connections.

When the signaling channel has been opened successfully or if for some reason the connect attempt fails, an `A2DP_SIGNALLING_CONNECT_CFM` message is sent to the client application task. If the connect attempt is successful this message contains a `device_id` and the `bd_addr` for the signaling channel that has been created. The `device_id` is used in all further interaction with the A2DP library to identify the remote device. The `device_id` is unique to the remote device for as long as the signaling channel remains established.

NOTE The `device_id` returned in this message should be stored by the application as it is needed in all further calls to the library. If the connection fails this ID is `IINVALID_DEVICE_ID`, which is defined in the `a2dp.h` file.

The sink for the signaling channel can be obtained using the `Sink A2dpSignallingGetSink()` function.

It is expected that this sink will be used by the application to set the link policy settings for the connection rather than for sending any data. All A2DP specific signaling must be performed through the A2DP library.

To create a media channel and put the A2DP state machine into the open state the application needs to call .

The `device_id` returned in the `A2DP_SIGNALLING_CONNECT_CFM` message needs to be passed into this function. This identifies the signaling channel that will be used to open the media connection. The list of Stream End-Point Identifiers(SEIDs) registered at initialization defines the order in which stream end-point connections should be attempted.

The A2DP specification mandates that SBC must be supported and used if another codec type cannot be negotiated.

The order in which the SEIDs were registered in the call to `A2dpInit` allows the application to determine the default order in which different codecs are selected when initiating a Media connection. For example, an application that supports AAC, MP3 and SBC may always want to attempt AAC first, then MP3 and, if both fail, resort to SBC. Another application may choose to try MP3 first. Altering the order of the SEID registration, using `A2dpInit`, allows a priority order to be configured by the application as a default.

It is also possible to provide a different codec selection priority by specifying a `seid_list` in the call to `A2dpMediaOpenRequest`. The `seid_list` is supplied as an array of local SEIDs ranked in priority order, with the first SEID in the array having the highest priority. This enables an application to entirely re-order the default codec selection list and/or request that only a subset of the registered SEPs be used.

To use the default codec selection priority, the call to `A2dpMediaOpenRequest` can be provided with a `size_seid_list` of `0` and a `seid_list` pointer set to `NULL`.

The `A2DP_MEDIA_OPEN_CFM` message communicates the outcome of the open request to the application. If the attempt succeeds the media sink and SEID selected are included in the message. The `A2DP_MEDIA_OPEN_CFM` message also contains a `stream_id` which is unique to this channel. The `stream_id` is used with various other A2DP library API functions to reference this particular stream.

The A2DP library also allows the application to select the configured codec service capabilities. During initialization the application can select whether the A2DP library should select the configured codec settings or whether the application would like to do this. This is done by setting the `library_selects_settings` flag in the SEP configuration structure.

In most cases applications will let the A2DP library select the codec settings. However, the `A2dpCodecConfigureResponse()` function is provided to allow custom codecs to be negotiated when this is not the case.

If the application has requested it select the codec settings it is sent an `A2DP_CODEC_CONFIGURE_IND` message by the A2DP library, see Figure 3-1.

The application must respond to the `A2DP_CODEC_CONFIGURE_IND` message by calling the `A2dpCodecConfigureResponse()` function.

NOTE The service capabilities format is described in the Service Capabilities section of the *Audio/Video Distribution Transport Protocol Specification*.

The response should indicate whether the capabilities have been accepted or not and the actual codec service capabilities that the application has chosen. The service capabilities must be returned in the same format they were received, but with the chosen parameters selected. If the A2DP library is selecting the codec service capabilities then the `A2DP_CODEC_CONFIGURE_IND` message is not sent.

NOTE An `A2DP_CODEC_CONFIGURE_IND` message is sent to the application whenever an `AVDTP_SET_CONFIGURATION_CMD` is received from the remote device(assuming the SEP has been configured to request the application select the codec settings).

Incoming A2DP connections

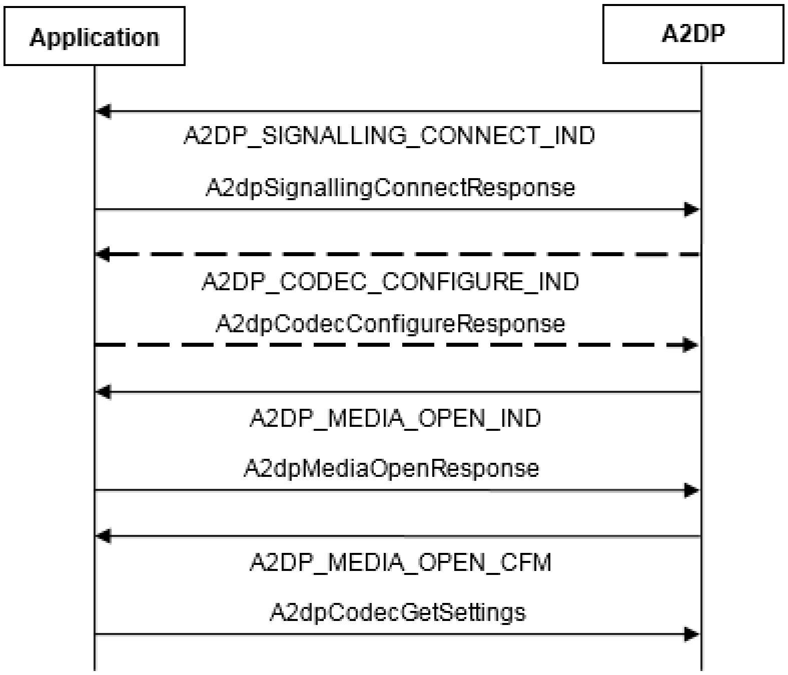

Figure 3-2 Incoming signal and media

NOTE Messages/functions with dashed arrows are only sent if the SEP is configured by the application so that the A2DP library does not select the codec settings.

The task registered in the call to `A2dpInit()` receives an `A2DP_SIGNALLING_CONNECT_IND` message. At this point the application can either accept or reject the incoming connection. In either case the application calls the `A2dpSignallingConnectResponse()` function:

The `device_id` parameter is passed to the application in the `A2DP_SIGNALLING_CONNECT_IND` message and should be copied directly into the response function.

An `A2DP_SIGNALLING_CONNECT_CFM` message is sent to the application to inform it of the outcome of the connection attempt.

NOTE If the application rejected the incoming connection an `A2DP_SIGNALLING_CHANNEL_CONNECT_CFM` message is returned with an `a2dp_operation_fail` status code.

When the signaling channel has been established the remote end may initiate a media channel. The task registered in the `A2dpInit` call receives an `A2DP_MEDIA_OPEN_IND` message. The application can choose to accept or reject the media connection by calling the `A2dpMediaOpenResponse()` function.

The application then receives an `A2DP_MEDIA_OPEN_CFM` message to indicate the outcome of the incoming media channel establishment. If the media channel has been opened successfully and configured into the open state, the message contains the media sink and SEID selected. The `A2DP_MEDIA_OPEN_CFM` message also contains a `stream_id` which is unique to this channel. The `stream_id` is used with various other A2DP library API functions to reference this particular stream.

NOTE When a signaling channel is established between two devices either side may initiate the creation of a media channel.

4 A2DP AV synch delay support

The A2DP library supports AV Synch Delay reporting, as defined by AVDTP v1.3. This requires an initial delay report to be issued on stream configuration and allows updates to the AV Sync Delay while streaming is active.

4.1 A2DP Endpoints operating as a sink

The A2DP library issues a request for the initial AV Sync Delay during the Stream Establishment process. When streaming has commenced, an application can update the AV Sync Delay as necessary.

Initial AV sync delay

During the Stream Establishment process, an application receives an `A2DP_MEDIA_AV_SYNC_DELAY_IND` message to indicate that an initial AV Sync Delay value is required.

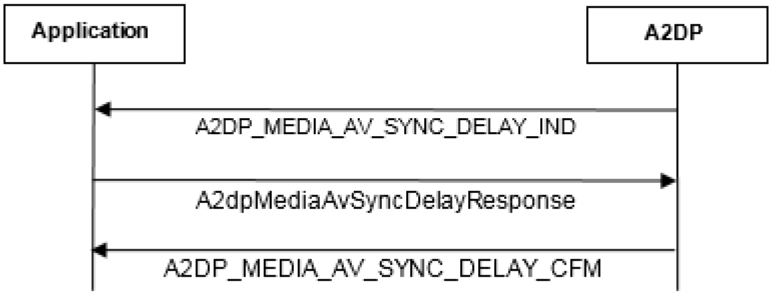
Application
A2DP
A2DP_MEDIA_AV_SYNC_DELAY_IND
A2dpMediaAvSyncDelayResponse
A2DP_MEDIA_AV_SYNC_DELAY_CFM

Figure 4-1 Initial sync delay

The application supplies this by calling the `bool A2dpMediaAvSyncDelayResponse()` function.

This causes an `AVDTP_DELAYREPORT` command to be issued to the source device.

An application receives an `A2DP_MEDIA_AV_SYNC_DELAY_CFM` message indicating the outcome of the operation.

Updating AV sync delay

When necessary, an application calls the `bool A2dpMediaAvSyncDelayRequest()` function to update the AV Sync Delay for the currently streaming endpoint, which causes an `AVDTP_DELAYREPORT` command to be issued to the source device.

The application receives an `A2DP_MEDIA_AV_SYNC_DELAY_CFM` message to indicate the outcome of the operation.

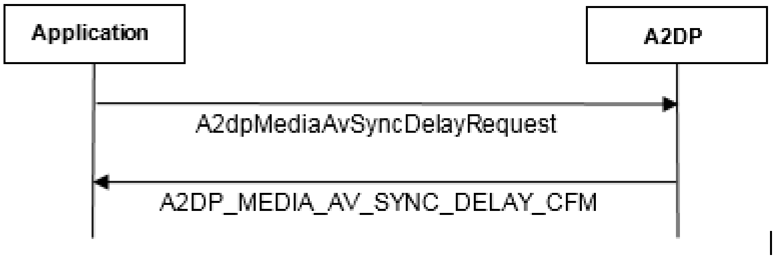

Figure 4-2 Updating AV sync delay

4.2 Endpoints operating as a source

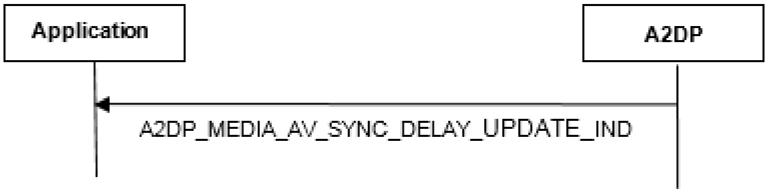

Figure 4-3 Endpoints operating as a source

The A2DP library notifies an application when it receives an `AVDTP_DELAYREPORT` command, from a remote device, by issuing an `A2DP_MEDIA_AV_SYNC_DELAY_UPDATE_IND` message. This message contains the new AV Sync Delay of the remote stream endpoint.

This message is only issued for endpoints operating as a source.

5 Starting an A2DP stream

If the A2DP library state machine is in the open state, for the source to start streaming audio data to the sink, the `AVDTP_START` command must be used to move it into the streaming state.

See the *Audio/Video Distribution Transport Protocol Specification*.

Initiating start

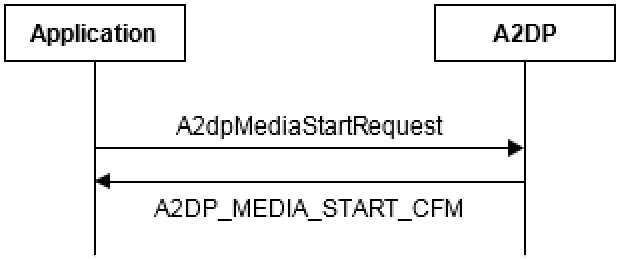

Figure 5-1 Initiating a start

An `A2DP_MEDIA_START_CFM` message is returned to the application informing it of the outcome of the request.

The application initiates a start by sending an `AVDTP_START` command over the signaling channel by calling the `bool A2dpMediaStartRequest()` function.

An `A2DP_MEDIA_START_CFM` message is returned to the application informing it of the outcome of the request.

Accepting start

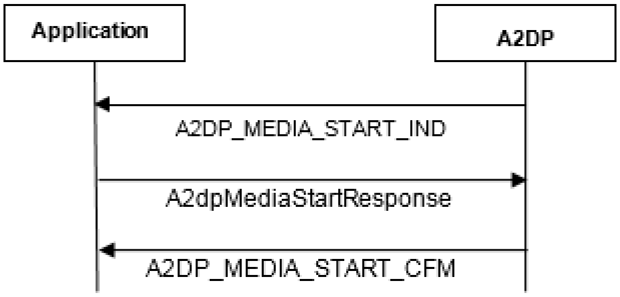

Figure 5-2 Accepting a start request

An application receives notification of a remote device issuing an `AVDTP_START` command from a `A2DP_MEDIA_START_IND` message. The application can choose to either accept/reject this request by calling the `bool A2dpMediaStartResponse()` function.

6 Suspending an A2DP Stream

The application can initiate an AVDTP_SUSPEND command over the signaling channel, by calling the `A2dpMediaSuspendRequest()` function.

An `A2DP_MEDIA_SUSPEND_CFM` message is returned to the application to inform it of the outcome of the request.

Initiating suspend

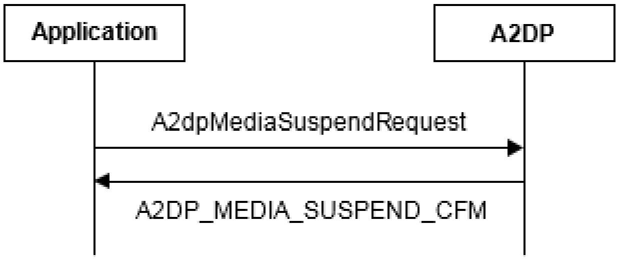

Figure 6-1 Initiating a suspend

Accepting a suspend

If the remote end sends the `AVDTP_SUSPEND` command it is automatically accepted by the A2DP library (assuming it is in the correct state) and an `A2DP_MEDIA_SUSPEND_IND` message is sent to the application to inform it that the state machine is now in the open state.

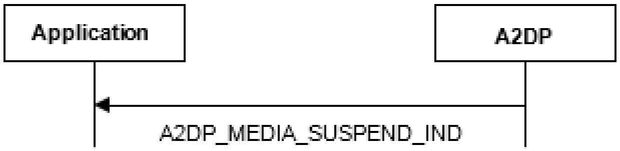

Figure 6-2 Accepting a suspend

7 Reconfiguring an A2DP stream

Use the AVDTP_RECONFIGURE command to reconfigure the operating parameters of a codec once a stream has been established.

NOTE A stream can only be reconfigured if it is in its Open (Suspended) state.

See the *Audio/Video Distribution Transport Protocol Specification*.

Initiating reconfigure

The application initiates the sending of an AVDTP_RECONFIGURE command over the signaling channel by calling the bool A2dpMediaReconfigureRequest() function.

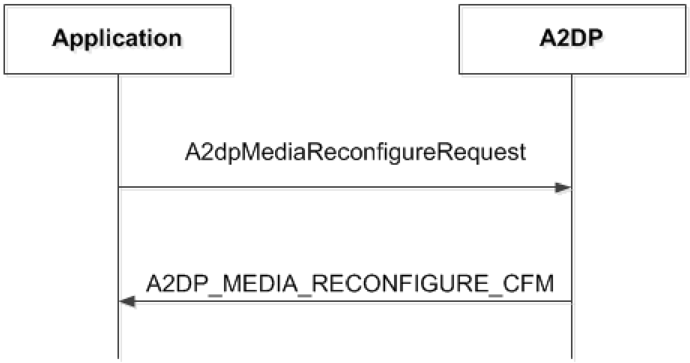

Figure 7-1 Intiating reconfigure

An A2DP_MEDIA_RECONFIGURE_CFM message is returned to the application informing it of the outcome of the request.

Accepting reconfigure

Assuming it is in the correct state, the AVDTP_RECONFIGURE command is automatically accepted by the A2DP library and an A2DP_MEDIA_RECONFIGURE_IND message is sent to the application to inform it which stream has been reconfigured.

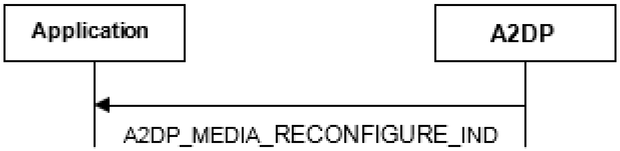

Figure 7-2 Accepting reconfigure

8 Closing an A2DP channel

The A2DP library API provides the application with the ability to close just the media channel or both the media and signaling channels.

Outgoing close command

If the stream is in the open or streaming state, an AVDTP_CLOSE command needs to be sent on the signaling channel. The application closes the stream by calling the A2dpMediaCloseRequest() function.

The A2DP library then sends an AVDTP_CLOSE to the remote end and releases the media channel. An A2DP_MEDIA_CLOSE_CFM message is sent to the application to inform it of the outcome of the request.

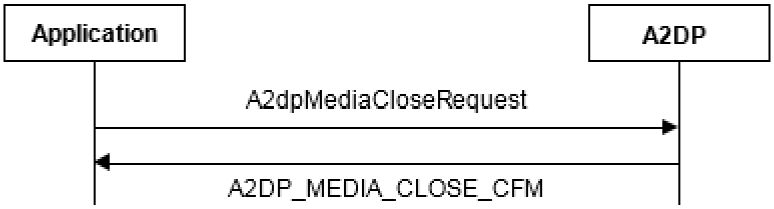

Figure 8-1 Outgoing close command

Incoming close command

If the remote end initiates the closing of the stream the A2DP library handles the response to the AVDTP_CLOSE command it receives and sends an A2DP_MEDIA_CLOSE_IND message to the application to indicate that the stream has been closed and the media channel released.

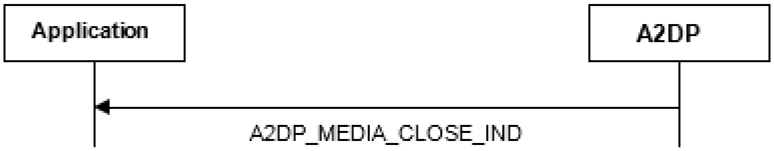

Figure 8-2 Incoming close command

Disconnecting all channels

Calling the API function to disconnect a Signaling channel when a Media channel is also established will close all channels.

However, rather than issuing an AVDTP_CLOSE to the remote device, an AVDTP_ABORT is generated which forces any media channels to close.

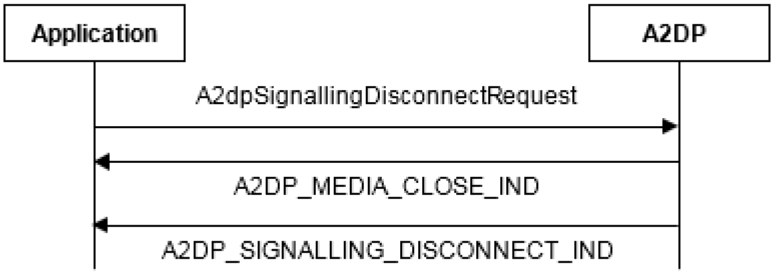
Application
A2DP
A2dpSignallingDisconnectRequest
A2DP_MEDIA_CLOSE_IND
A2DP_SIGNALLING_DISCONNECT_IND

Figure 8-3 Disconnecting all channels

The A2DP library provides a single function, `A2dpSignallingDisconnectRequest()`, used to release all A2DP channels currently active in a particular A2DP library instance.

On receiving this command the A2DP library:

1. Aborts any open media stream (if one exists) by sending `AVDTP_ABORT` to the remote end.
2. Closes the media and signaling channels, and:
   a. An `A2DP_MEDIA_CLOSE_IND` message is sent to the application if the A2DP library was in the streaming state, to indicate that the media channel has been disconnected.
   b. An `A2DP_SIGNALLING_DISCONNECT_IND` message is sent to the application when the signaling channel is closed.

9 AD2P library link loss management

Each remote device has its own timer. When enabled, the A2DP library functionality automatically attempts to reconnect a signaling channel to the remote device in the event of a link loss for the duration of the remote device's timer, as specified in a call to `A2dpInit`.

Each time a signaling channel to a remote device is established, link loss management for that device is disabled by default. Link loss management can be enabled/disabled as and when required by calling the `A2dpDeviceManageLinkloss()` function.

If a link loss occurs while link loss management is enabled the application receives an `A2DP_SIGNALLING_LINKLOSS_IND` message, to identify the device to which the link loss has occurred.

If the A2DP library manages to re-establish a signaling channel, then the application receives an A2DP_SIGNALLING_CONNECT_CFM message with the same `device_id` as in the `A2DP_SIGNALLING_LINKLOSS_IND` message.

If the A2DP library fails to re-establish a signaling channel within the specified time period, then the application receives an `A2DP_SIGNALLING_DISCONNECT_IND` message with the same `device_id` as in the `A2DP_SIGNALLING_LINKLOSS_IND` message.

Disconnecting a device during link loss, by calling `A2dpSignallingDisconnectRequest()`, terminates any ongoing link loss management for that device.

Disabling link loss management for a device during link loss, by calling `A2dpDeviceManageLinkloss()`, terminates any ongoing link loss management for that device.

10 A2DP and audio library interaction

To facilitate processing and routing of audio data whenever the A2DP stream starts or stops, the application uses the Audio library to select the correct codec plugin from a number of commonly used A2DP codecs, for example SBC and MP3.

The plugin ensures that the correct DSP decoder is used and that, when decoded, the audio data is routed correctly to the speakers. See *An Introduction to Audio Plugins*.

When the application receives an indication that streaming has started, that is an `A2DP_MEDIA_START_IND` or `A2DP_MEDIA_START_CFM` message, it calls `AudioConnect()`.

When the application receives the indication that streaming has stopped, it calls AudioDisconnect and receives one of the following messages:

- `A2DP_MEDIA_SUSPEND_IND`
- `A2DP_MEDIA_SUSPEND_CFM`
- `A2DP_MEDIA_CLOSE_IND`
- `A2DP_MEDIA_CLOSE_CFM`

Document references

Document	Reference
Audio/Video Distribution Transport Protocol Specification	Specification of the Bluetooth System, Profiles, v1.3, Audio/Video Distribution Transport Protocol Specification
Generic Audio/Video Distribution Profile Specification	Specification of the Bluetooth System, Profiles, v1.3, Generic Audio/Video Distribution Profile
Advanced Audio Distribution Profile Specification	Specification of the Bluetooth System, Profiles, v1.3, Advanced Audio Distribution Profile
Introduction to the Audio Library	80-CT561-1/CS-00402556-AN

Terms and definitions

Term	Definition
A2DP	Advanced Audio Distribution Profile
AAC	Advanced Audio Coding
ADK	Audio or Application Development Kit
API	Application Programming Interface
AV	Audio/Video
AVDTP	Audio/Video Distribution Transport Protocol
Bluetooth	Set of technologies providing audio and data transfer over short-range radio connections
Codec	COder DECoder
DSP	Digital Signal Processor
IC	Integrated Circuit
id	Identifier
MP3	MPEG-2 Layer 3
PSM	Protocol Service Multiplexor
SBC	Sub-band Coding
SEID	Stream Endpoint Identifier
SEP	Stream Endpoint
SYNC	Synchronize