

---

# S60 Platform: Music Application Developer's Guide

**Version 1.1**  
November 14, 2006

S60  
p l a t f o r m

## Legal Notice

Copyright © 2005, 2006 Nokia Corporation. All rights reserved.

Nokia and Nokia Connecting People are registered trademarks of Nokia Corporation. Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. Other product and company names mentioned herein may be trademarks or trade names of their respective owners.

### Disclaimer

The information in this document is provided "as is," with no warranties whatsoever, including any warranty of merchantability, fitness for any particular purpose, or any warranty otherwise arising out of any proposal, specification, or sample. Furthermore, information provided in this document is preliminary, and may be changed substantially prior to final release. This document is provided for informational purposes only.

Nokia Corporation disclaims all liability, including liability for infringement of any proprietary rights, relating to implementation of information presented in this document. Nokia Corporation does not warrant or represent that such use will not infringe such rights.

Nokia Corporation retains the right to make changes to this specification at any time, without notice.

### License

A license is hereby granted to download and print a copy of this specification for personal use only. No other license to any other intellectual property rights is granted herein.

## Contents

<b>1.</b>	<b>Introduction .....</b>	<b>5</b>
<b>2.</b>	<b>Audio Priority and its ramifications.....</b>	<b>6</b>
<b>3.</b>	<b>Playing music from a file.....</b>	<b>7</b>
3.1	Available APIs for file playback.....	7
3.2	Selecting the appropriate API .....	8
3.3	Play utility life cycle for file playing.....	9
3.3.1	Initialization.....	10
3.3.2	Playing.....	15
3.3.3	Pause.....	18
3.3.4	Stopping.....	19
3.3.5	Destruction .....	20
3.4	Seeking .....	21
3.5	Playing a series of files .....	23
3.5.1	Playing a new file after the previous file completes playing.....	23
3.5.2	Playing a new file when another file is already playing .....	24
3.5.3	Playing a new file after pausing a previous file playing.....	26
3.5.4	Playing a new file after stopping a previous file playing.....	28
<b>4.</b>	<b>Reading metadata .....</b>	<b>30</b>
<b>5.</b>	<b>Streaming buffers from the application .....</b>	<b>34</b>
5.1	CMdaAudioOutputStream life cycle.....	34
5.1.1	Initialization.....	35
5.1.2	Playing.....	37
5.1.3	Stopping.....	39
5.1.4	Destruction .....	40
<b>6.</b>	<b>Internet music streaming service by streaming within an application .....</b>	<b>42</b>
<b>7.</b>	<b>References .....</b>	<b>44</b>
<b>8.</b>	<b>Terms and abbreviations.....</b>	<b>45</b>
<b>9.</b>	<b>Evaluate this resource.....</b>	<b>46</b>

## Change History

December 16, 2005	Version 1.0	Initial document release
November 14, 2006	Version 1.1	Removed information about Metadata Utility API (not included in the SDK)

---

## 1. Introduction

The S60 platform provides developers with many options for creating rich music applications and delivering them to consumers. S60 devices also have extensive audio capabilities that can be used to deliver exciting applications based on audio alone.

The purpose of this document is to provide guidance on using S60 multimedia APIs to create music applications for S60 3rd Edition and future devices. The document contains descriptions and examples that illustrate the use of multimedia APIs to accomplish common operations for music applications. In addition, this document covers alternative techniques for accomplishing similar operations while explaining the design tradeoffs between the different approaches.

The format of the document is based upon translating application use cases into multimedia services so that the appropriate use of APIs to meet the needs of the use case is fully illustrated. In some instances, the use case may only require utilization of a single API; in other instances, it may require use of multiple APIs. In the latter case, the cooperative use of these APIs will be discussed and the reasoning behind the coordination of the multiple APIs will be elaborated upon.

This document assumes that the reader is already familiar with the general guidelines for creating an S60 application. More information about creating an S60 application can be found in *Getting Started With C++ Development On The S60 SDK [1]*. Therefore, the standard issues surrounding use of the UI framework, build environment, and common Symbian structures will not be discussed here.

If you are building a full-blown music service client, you may also want to familiarize yourself with other S60 APIs such as browser control and download management. These are documented in specific developer's guides, for example, *S60 Platform: Browser Control API Developer's Guide* and *S60 Platform: Download Manager API Developer's Guide*. For more information about S60 APIs, visit [www.forum.nokia.com](http://www.forum.nokia.com).

---

## 2. Audio Priority and its ramifications

There are times when a device's audio hardware will receive simultaneous requests from multiple applications (clients). A common example of this is when a user receives a phone call while using the device for a multimedia application. At that moment, the device can either keep playing the media or stop it and play the phone ring instead. Audio Priorities and Audio Preference are used to resolve such situations. The Audio Policy component manages access priority to audio hardware on a device by resolving simultaneous requests from different clients. When multiple applications make simultaneous audio playback requests, Audio Policy decides which should be permitted to play, which is denied access to the audio resources, and whether it is possible to mix two or more sounds.

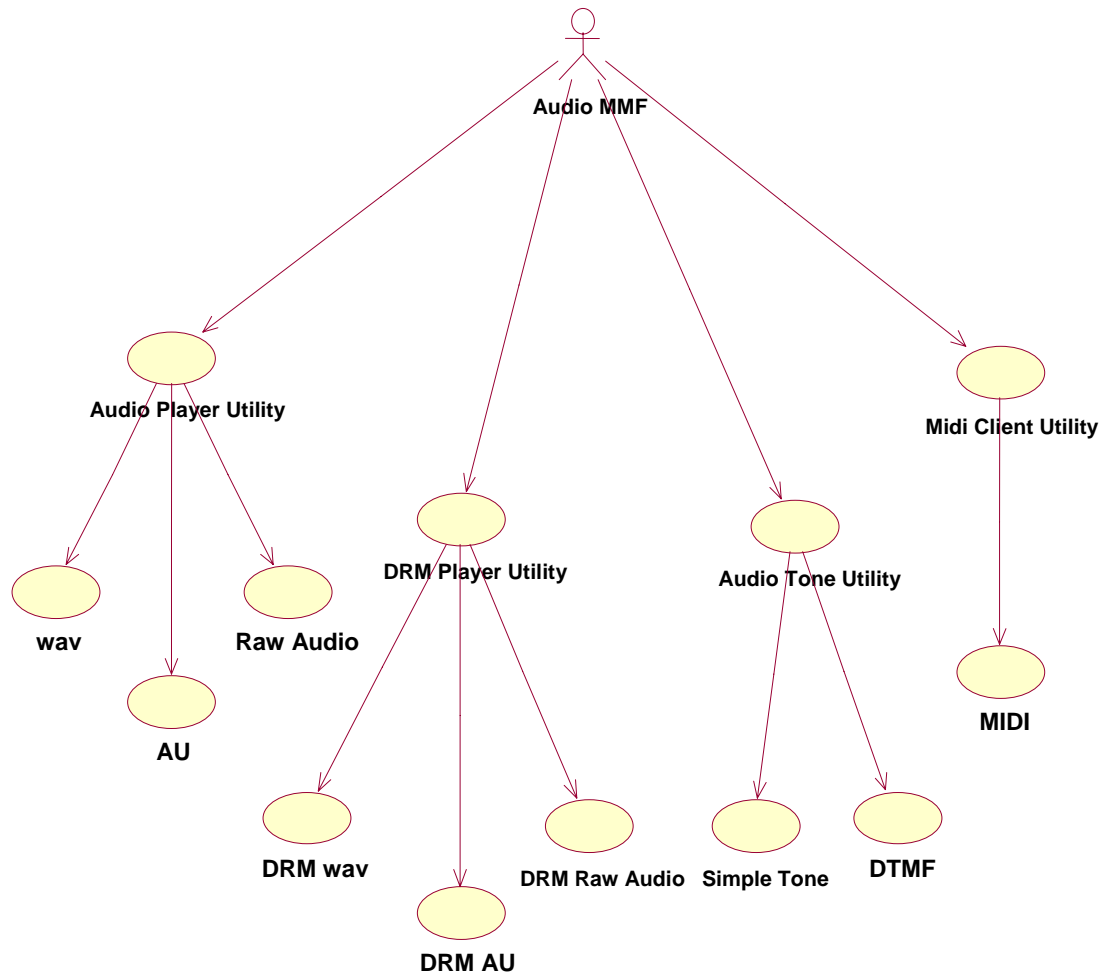
Several design aspects should be considered when designing a custom application. Different requirements for product behavior, capability of the hardware, and availability of system resources will determine the appropriate Audio Priority implementation.

Audio Priorities range from `EMdaPriorityMin` to `EMdaPriorityMax`. The former is the lowest priority and indicates that any other client can interrupt the client; the latter indicates that other clients cannot interrupt the client. A priority of `EMdaPriorityNormal` means that the client can be interrupted but only by higher-priority clients. These values are defined in the `TMdaPriority` enum [2].

Audio Priority determines which events have greater importance and therefore should be taken care of first. That, however, is not its only function—priority ramifications can help you define the behavior that will be adopted by a client using a sound device if a higher-priority client takes over that device.

### 3. Playing music from a file

The S60 platform uses the multimedia framework (MMF) API to provide multimedia services to applications. It acts as a repository for plug-in multimedia processing units, and serves as the generic interface to a device's hardware. There are several interfaces within this API for playing music from a file; the next Use Case shows this.



#### 3.1 Available APIs for file playback

The S60 MMF provides a client API consisting of several interfaces that enable manipulation of audio features. The MMF is capable of both local and streaming playback of a wide variety of audio formats, depending on the vendor product (different vendors might support different formats for streaming). Here is a brief description of each of the interfaces that can be used for playing music from a file.

- Audio Player Utility – A general-purpose interface that can play Digital Rights Management (DRM) protected content if the user application is DRM

capable<sup>1</sup>. This interface provides methods to create, play, and manipulate audio data stored in files and descriptors. The `CMdaAudioPlayerUtility` class [3] provides this functionality. The list of supported audio formats for input and output is open ended because the audio class is plug-in based. The audio file formats supported as standard by MMF are AU and WAV. Input and output audio data can be of any format supported by the installed plug-ins.

- DRM Player Utility – A general-purpose interface that allows applications without DRM capability to play DRM-protected content, but performance is slightly impacted. It also provides methods to access audio data stored in files (both encrypted and unencrypted) and descriptors (unencrypted). The `CDrmPlayerUtility` class [4] provides this functionality. The list of supported audio formats for playback is open ended because the audio class is plug-in based. The audio file formats supported as standard by MMF are AU, WAV, and raw audio data.
- Audio Tone Utility – An interface that provides methods for playing and configuring single sequenced tones as well as Dual-Tone Multifrequency (DTMF) strings. The tone player functionality is provided by the `CMdaAudioToneUtility` class [5], and this class can be used to play:
  - Single tones of a specified duration and frequency (if supported by the device);
  - Dual tones (if supported by the device);
  - DTMF strings;
  - Sequences of tones held in files or descriptors (different vendors might support different formats/types);
  - Predefined (fixed) sequences of tones held in the mobile equipment (different vendors might or might not support this. Nokia devices do not currently support it).

Midi Client Utility – Provides an interface to open, play, and obtain information in MIDI format. MIDI data can be supplied either in a file or a descriptor. The `CMIDIClientUtility` class [6] provides Midi Player functionality. This API also supports playing live midi events and provides greater control over midi playback attributes (tempo, etc.).

## 3.2 Selecting the appropriate API

As stated earlier, the Audio Player Utility is a general-purpose API, but it requires that the applications that use this API provide DRM capability in order to play DRM-protected content. Applications using the Audio Play API must have implemented the error handles for DRM content through the use of `DRMHelper` [7]. These applications must also implement data checking and decryption through the use of `CDRMLicenseChecker` [8]. Most commonly used DRM functions can be accessed through `DRMCommon` [9]. This class will provide vast functionality such as getting expiration details from a file or obtaining the respective Play, Display, Print, and Execute rights from a file.

`CDrmPlayerUtility` [4] provides the same interface as `CMdaAudioPlayerUtility` [3] to open, play, and obtain information from sampled audio data. Audio data can be supplied to `CDrmPlayerUtility` [4] in the exact same ways as in `CMdaAudioPlayerUtility` [3]. The main difference between these two APIs is that

---

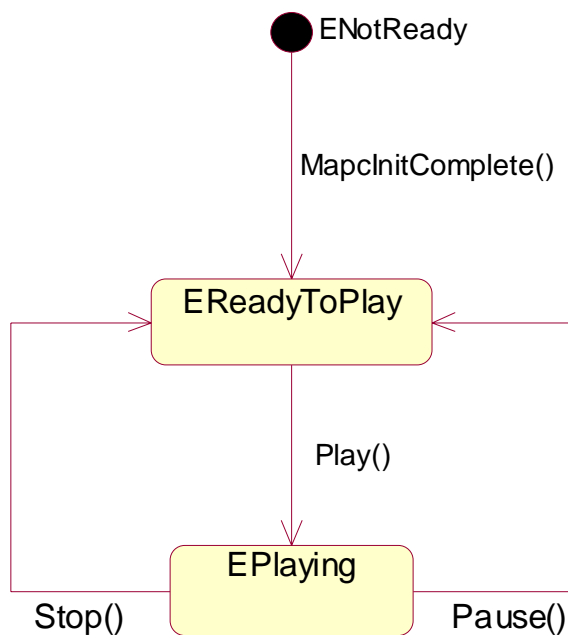
<sup>1</sup> DRM capability is a part of the Symbian capability model. If you absolutely need this, it needs to be granted by each manufacturer. However, the DRM player utility may be sufficient.

`CDrmPlayerUtility` [4] allows applications without DRM capability to play DRM-protected content. The tradeoff for using `CDrmPlayerUtility` [4] to play DRM-protected content is that application performance is slightly impacted.

`CDrmPlayerUtility` [4] provides the same interface as `CMdaAudioPlayerUtility` [3], which means that simply by changing the imported class, most of the code can remain the same, because the methods from `CMdaAudioPlayerUtility` [3] can be called in the same way from `CDrmPlayerUtility` [4].

### 3.3 Play utility life cycle for file playing

The three states defined by the developer for a file-playing application are:



These states can be declared within the audio player client application as follows:

```
enum TState
{
    ENotReady,
    EReadyToPlay,
    EPlaying
};
```

```
TState iState;
```

This API, provided by the MMF for audio playback, is independent of the format or type of audio being played. The same API is used for playback of both local and streaming contents and is supported by the following S60 classes:

- `CMdaAudioPlayerUtility` – The Audio Player class that supports audio playback operations and simple metadata retrieval operations. This is the class that implements the decoder [3].
- `MMdaAudioPlayerCallback` – The Callback class for the `CMdaAudioPlayerUtility` class that reports errors and notifies the application of the status of file-open operations or when play has completed. This is an observer class. Basically this class requires the implementation of

`MapcInitComplete()` and `MapcPlayComplete()`, which will be described in Section 3.3.1 [3].

### 3.3.1 Initialization

Initializing an audio sample is a fairly simple task, but there are some things that must be taken in consideration. Since all operations in the `CMdaAudioPlayerUtility` class are asynchronous, it is necessary to have a client class observing the audio-playing operation. This class must inherit the mixing class `MMdaAudioPlayerCallback`, which provides two functions: `MapcInitComplete()` and `MapcPlayComplete()`. `MapcInitComplete()` defines required client behavior when an attempt to open and initialize an audio sample has been completed, successfully or otherwise. `MapcPlayComplete()` defines required client behavior when an attempt to play an audio sample has been completed, successfully or otherwise [3].

It is a good practice to change the state of the object from `ENotReady` to `EReadyToPlay` when the callback function `MapcInitComplete()` [3] gets called. This way you can signal to the application that the initialization process was completed and successful and that it is now ready to play the audio file. You can also take the necessary measures if there was an error during initialization.

```
void CAudioPlayer::MapcInitComplete(TInt aError, const
TTimeIntervalMicroSeconds& /*aDuration*/)
{
    iState = aError ? ENotReady : EReadyToPlay;
}
```



**Note:** If the client application requires it, the duration of the clip can be displayed in the GUI.

It is also good practice to use `MapcPlayComplete()` [3] to signal the application that the playing procedure is done and the client is ready for a new request. Through this function you can take the necessary measures if there was an error during playback of an audio file.

```
void CAudioPlayer::MapcPlayComplete(TInt aError)
{
    iState = aError ? ENotReady : EReadyToPlay;
}
```

Once these two callback functions have been implemented from `MMdaAudioPlayerCallback` [3], the `CMdaAudioPlayerUtility` object can be initialized. To begin the initialization process it is first necessary to create an instance of the `CMdaAudioPlayerUtility` class [3]. `CMdaAudioPlayerUtility` provides several options for initializing the object. There are two scenarios at the moment of the initialization where the audio file to be played can either be specified or not.

As mentioned previously, in the first scenario the player object can be instantiated and, at the same time, the audio file can be loaded (opened) and made ready for playing, all in one step. In this case, the `CMdaAudioPlayerUtility` provides the following functions:

- The function `NewFilePlayerL()` [3] constructs and initializes a new instance of the audio player utility for playing sampled audio data from a file (`aFileName`). The next code snippet shows how to create the instance:

```
CMdaAudioPlayerUtility* iMdaAudioPlayerUtility =
CMdaAudioPlayerUtility::NewFilePlayerL(aFileName, *this);
```

- The function `NewDesPlayerL()` [3] constructs and initializes a new instance of the audio player utility for playing sampled audio data from a descriptor (`aDescriptor`). The next code snippet shows how to create the instance:

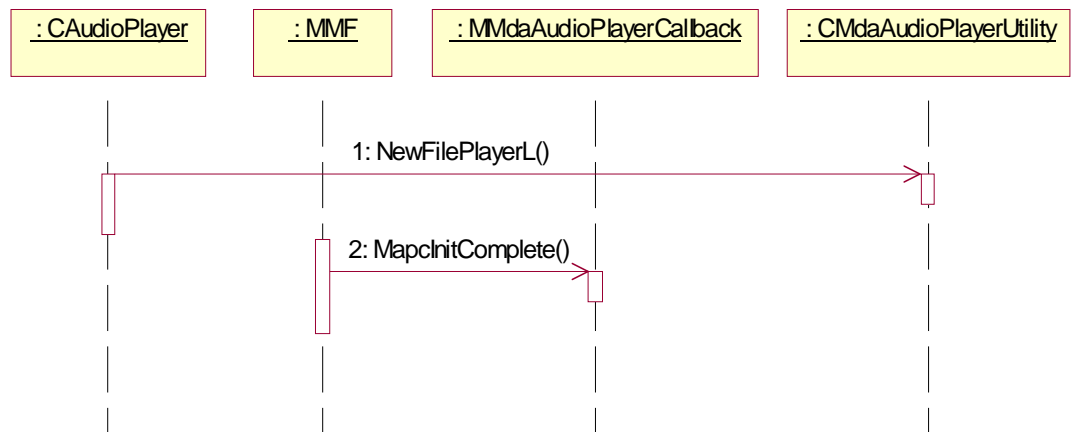
```
CMdaAudioPlayerUtility* iMdaAudioPlayerUtility =
CMdaAudioPlayerUtility::NewDesPlayerL(aDescriptor, *this);
```

- Finally, the function `NewDesPlayerReadOnlyL()` [3] constructs and initializes a new instance of the audio player utility for playing sampled audio data from a read-only descriptor (`aDescriptor`). The next code snippet shows how to create the instance:

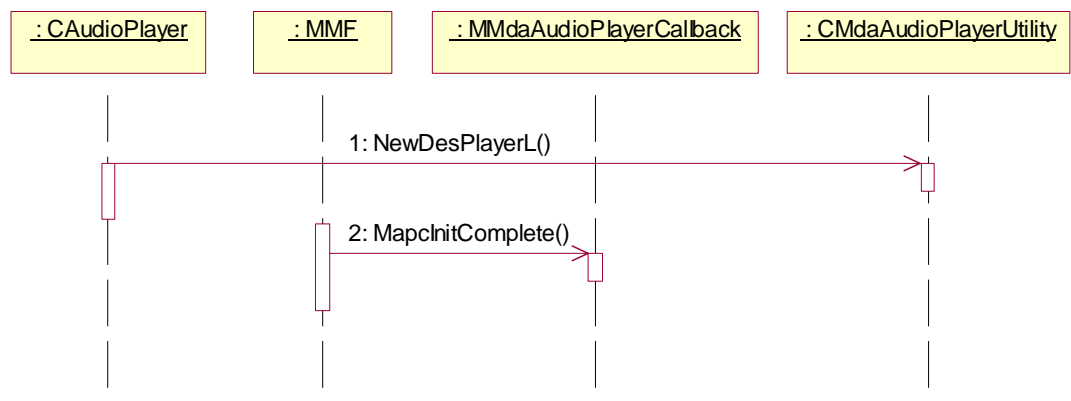
```
CMdaAudioPlayerUtility* iMdaAudioPlayerUtility =
CMdaAudioPlayerUtility::NewDesPlayerReadOnlyL(aDescriptor,
*this);
```

All these functions must receive the audio data in a supported format, and will leave if the object cannot be created.

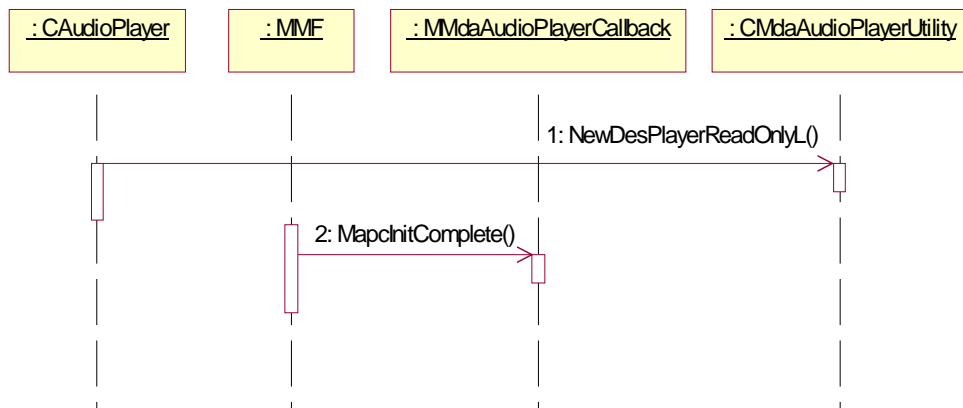
The following sequence diagram shows the initialization process using the method `NewFilePlayerL()`:



The following sequence diagram shows the initialization process using the `NewDesPlayerL()` function:



The following sequence diagram shows the initialization process using the `NewDesPlayerReadOnlyL()` function:



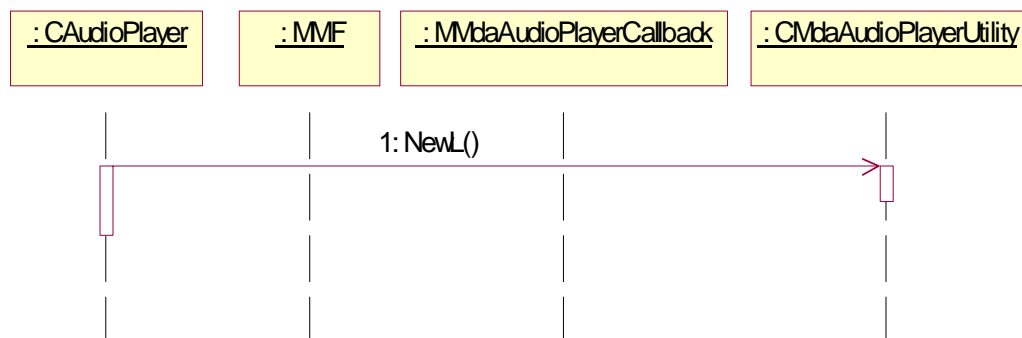
Obviously, the initialization process is pretty much the same on each of the three available functions for the first scenario. You may also notice that there is a call to the `MapcInitComplete()` after the audio player object has been initialized; this happens because the audio file has been opened within the same function that is used to initialize the object.

In the second scenario, when you just want to instantiate the player object without having to specify an audio clip, the next `CMdaAudioPlayerUtility` member function must be used:

- The `NewL()` function constructs and initializes a new instance of the audio player utility and it leaves if the audio player utility object cannot be created. The next code snippet shows how to create an instance:

```
CMdaAudioPlayerUtility* iMdaAudioPlayerUtility =
CMdaAudioPlayerUtility::NewL( *this );
```

The initialization process with the `NewL()` function is shown in the following sequence diagram. Note that there is not a callback after the initialization of the object when this function is used.



For each of the functions above, when creating the `CMdaAudioPlayerUtility` instance, it is possible to specify the priority and preference for the instance. These arguments are optional and the default values are `EPriorityNormal` and `RMdaPriorityPreferenceTimeAndQuality`.

All of the methods to initialize the instance are asynchronous. Therefore the application is not blocked while performing this operation.

Continuing with the second scenario option, whenever you need to play an audio file, once you have created the instance with the `NewL()` function, the file needs to be opened in order to have it ready for playback. Remember that this step was

not necessary in the first scenario since it is possible to open the audio file within the same initialization function. A file can be opened by passing in the file name with the full path information or a descriptor of the file using one of the following:

- The `OpenFileL()` [3] function opens an audio clip from a file (`aFileName`). The audio data must be in a supported format. The next code snippet shows an example of this function:

```
iMdaAudioPlayerUtility ->OpenFileL( aFileName );
```

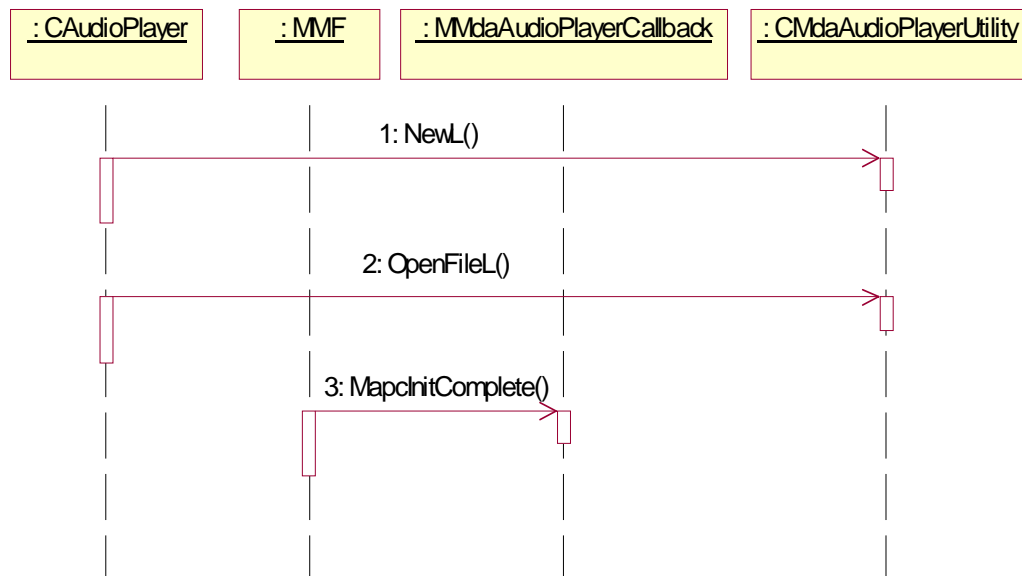
- The second option to open an audio clip is to use the `OpenDesL()` [3] function, which will open the audio clip from a descriptor (`aDescriptor`). The audio file must be in a supported format. The following code snippet shows an example of this function:

```
iMdaAudioPlayerUtility ->OpenDesL( aDescriptor );
```

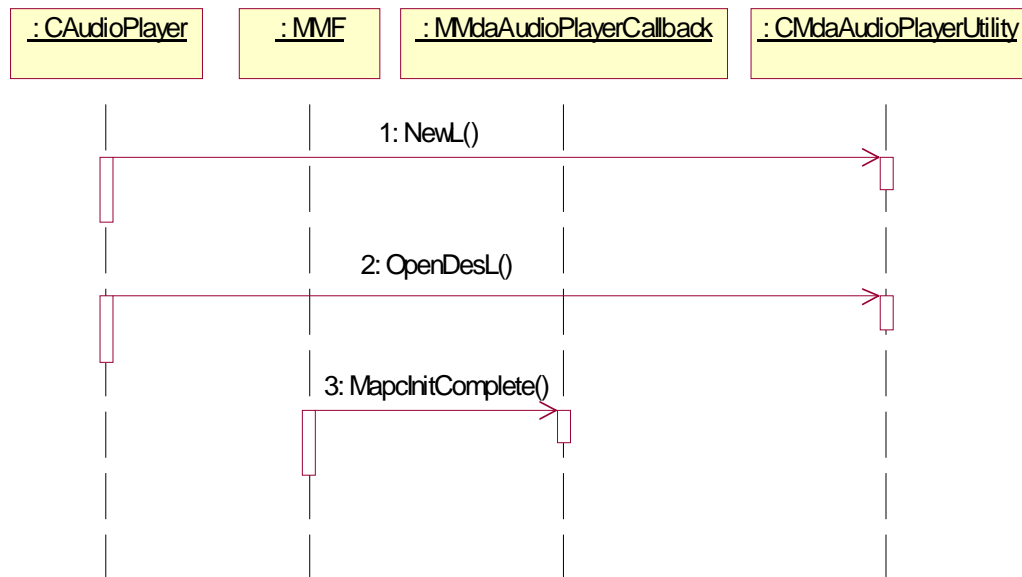
The audio file provided for these two functions must be in a supported format. These functions leave with `KerrInUse` if there is a previous open statement awaiting notification of completion. Note that an MP3 descriptor needs to include the mime type in the beginning of the data.

The next two sequence diagrams show the initialization process when using the `NewL()` function to create an instance of the player and when opening an audio file, either with the `OpenFileL()` function to open the audio file or the `OpenDesL()` function when opening the audio file through its descriptor.

When using `OpenFileL()` :



When using `OpenDesL()`:



As you can see, each time after opening a file the `MapchritComplete()` [3] method is called with the returned error information, such as `KerrNone`, `KerrNotSupported`, or `KerrNoMemory` [10], to notify the application of the status of the file-open operation. The player is not ready to play until the application gets this notification. Then, if no error has occurred—`KerrNone` during the file-open operation—the playback operation can be started. This happens with any of the initialization functions described for the first scenario (`OpenFileL`) since the audio files are open within each of these functions, but it also happens in the second scenario (`OpenDesL`) after an audio file has been opened. There are also other possible error values but these values are dependent on the EPOC platform.

In order to handle the possible error cases it is enough to ask whether the returned error information in `MapchritComplete()` [3] is different from `KerrNone` and then proceed according to the error signaled.

```

void CAudioPlayer::MapchritComplete(TInt aError, const
TTimeIntervalMicroSeconds& /*aDuration*/)
{
    iState = aError ? ENotReady : EReadyToPlay;

    switch(aError)
    {
    case KerrNone:
        ... Actions required after a successful
        initialization...
        break;
    case KerrNotFound:
        ... Actions required after a KerrNotFound error...
        break;
    case KerrNotSupported:
        ... Actions required after a KerrNotSupported error...
        break;
    case KerrNoMemory
        ... Actions required after a KerrNoMemory error...
        break;
    }
}

```

```

    case KerrDied
        ... Actions required after a KerrDied error...
        break;
    default:
        ... Default actions required...
        break;
}
}

```

As mentioned previously, the `NewFilePlayerL()` and `OpenFileL()` functions require the audio file to be played as an argument. The next code snippet shows an example of how to get the audio file for these functions. In this example, a function named `GetFile()` has been implemented within the audio client application. The parameter for this function is a descriptor containing the name of the audio file to be played, and it will return the audio file together with its path.

```

TFileName CAudioPlayer::GetFile(const TDesC& aAudioFile)
{
    TFileName fname(aAudioFile);
    CompleteWithAppPath(fname);
    return fname;
}

```

The next example shows how to use the `GetFile()` function:

```

//First we specify the name of the audio file to be played
//in a descriptor
_LIT( KAPlayerFileWAV, "audiofilename.wav" );

//Then we call the GetFile() function to get the
//audio file from its location
TFileName aFileName = GetFile(KAPlayerFileWAV);

//And now we are ready to open the audio file
TRAPD(aError, iMdaAudioPlayerUtility->OpenFileL( aFileName ));

```

Just remember that for the `OpenFileL()` function, an initialization with `NewL()` has to be performed before opening the file.

### 3.3.2 Playing

Once the initialization is complete and the audio file has been opened, you are ready to play the audio file. As shown in the code below, the playback of the file is asked by a call to `CMdaAudioPlayerUtility::Play()` [3], which will start playback of audio data from the current position.

```

void CAudioPlayer::PlayL()
{
    if(iState==EReadyToPlay)
    {
        iMdaAudioPlayerUtility ->Play();
        iState=EPlaying;
    }
}

```

In the code snippet above, you will notice that the first thing to do is to validate the `iState` variable. Remember that each time an audio file is opened, the `iState` variable is set to `EReadyToPlay` within the `MapcInitComplete()` callback, and immediately after the audio file begins to play, the `iState` variable has to be set to `EPlaying`.

After playback is stopped by Audio Policy or completed, the `MapcPlayComplete()` [3] method is called, either with a `KErrNone`, to signal that the operation was completed successfully, or with one of the possible error codes. `KErrCorrupt` lets you know that the sample data is corrupt. `KErrInUse` indicates that the sound device is in use by another, higher-priority client. This error can also be returned during playback if a current audio job is preempted by a higher-priority audio job. `KErrNoMemory` will let you know that there is insufficient memory to play this audio sample. Other values are possible, indicating a problem opening the audio sample. These values are device dependent [10]. Just as with the initialization, an identical code can help you handle errors during playback.

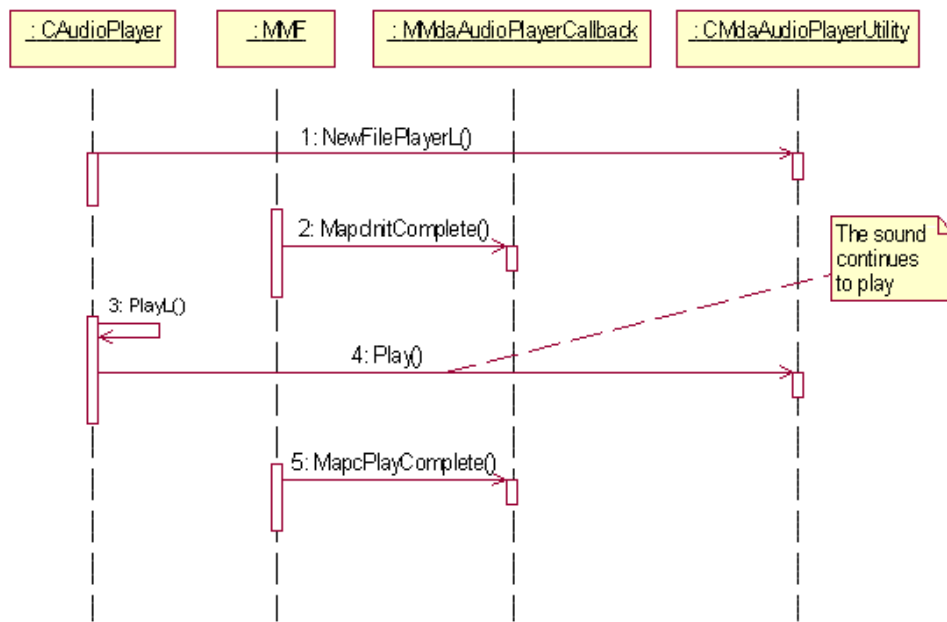
```
void CAudioPlayer::MapcPlayComplete(TInt aError)
{
    iState = aError ? ENotReady : EReadyToPlay;

    switch(aError)
    {
    case KErrNone:
        ... Actions required after a successful playback...
        break;
    case KErrCorrupt:
        ... Actions required after a KErrCorrupt error...
        break;
    case KErrInUse:
        ... Actions required after a KErrInUse error...
        break;
    case KErrNoMemory:
        ... Actions required after a KErrNoMemory error...
        break;
    default:
        ... Default actions required...
        break;
    }
}
```

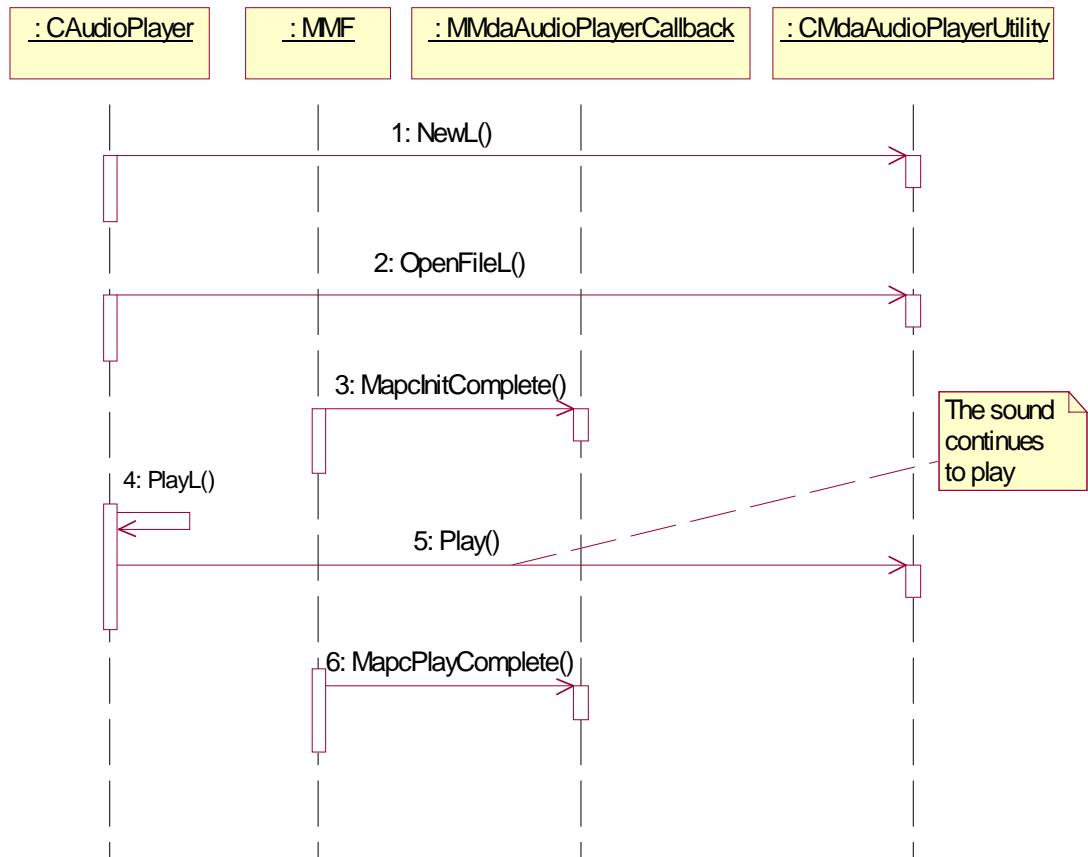
The play methods are asynchronous. This means that the application is not blocked while these operations complete. For the `CMdaAudioPlayerUtility` classes, the play operation completes either when the utility has played the whole of the sampled sound in the specified file or descriptor, or when an error occurs in the middle of playback.

When an asynchronous operation completes, the audio utility object can signal the event to the application object, or any other observer object, by calling a callback function on the observer object.

Remember that before an audio file is played, the initialization process needs to be done and the audio file opened. The next sequence diagram shows the first scenario, when the initialization and open processes are done in just one step. The `NewFilePlayerL()` function is used for this example, but it is possible to use any of the functions already described (see Section 3.3.1, "Initialization"):



The second scenario (see Section 3.3.1, “Initialization”), where initialization and opening of the audio file processes are done in two steps, is reflected in the following sequence diagram:



In this example, the file is opened with the `OpenFileL()` function, but it can also be done with the `OpenDesL()` function (see Section 3.3.1, "Initialization.").

Once all playing has been completed and/or the loaded audio file will not longer be used, the cleanup operation `CMdaAudioPlayerUtility::Close()` [3] should be performed. This method closes the current audio clip, allowing another clip to be opened, therefore the `iState` variable has to be set to `ENotReady` since the audio file will not be available for playback. In the example client application, this is done within the `Close()` function, as illustrated in the next code snippet, but it can be placed whenever the client application needs it.

```

void CAudioPlayer::Close()
{
    iMdaAudioPlayerUtility->Close();
    iState = ENotReady;
}
  
```

### 3.3.3 Pause

It is possible to pause the playback of the audio file just by calling the method `CMdaAudioPlayerUtility::Pause()` [3]; the position within the audio file is maintained in case a subsequent `PlayL()` is issued.

```

TInt CAudioPlayer::PauseL()
{
    TInt aError = iMdaAudioPlayerUtility->Pause();
    if(aError != KErrNone)
    {
  
```

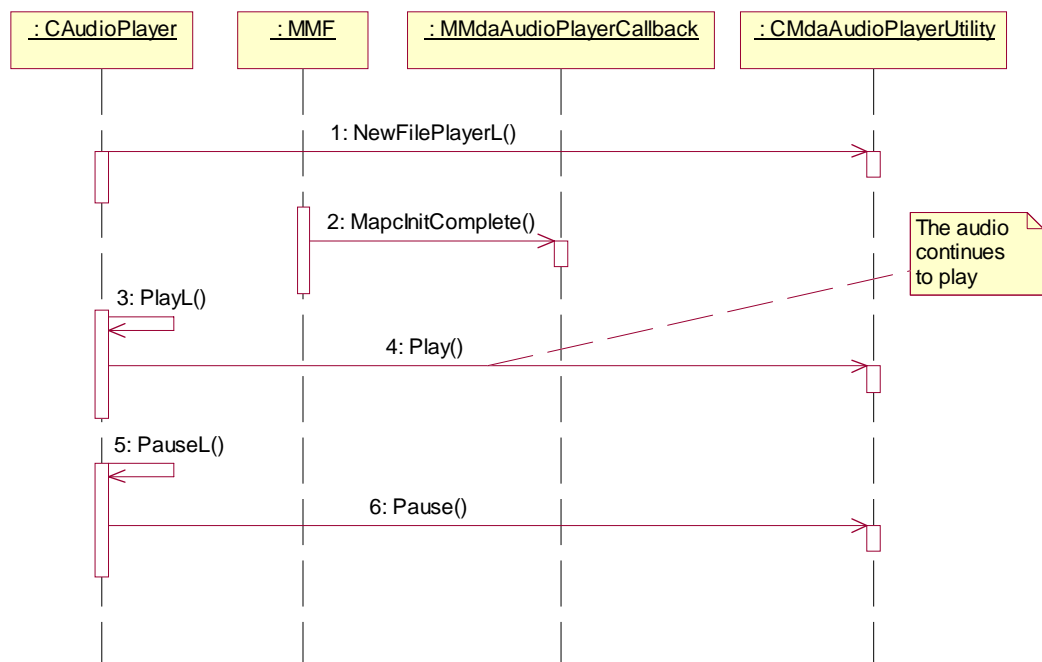
```

    ... Actions required for handling a general error ...
  }
  iState = EReadyToPlay;
}

```

As with the previous operations, there is a possibility that errors may occur during the pause action. The `Pause()` function will return a `Tint` to signal any possible errors, and therefore you can use this to handle any errors for this operation. Also, as can be seen in the code snippet, the `iState` variable is set to `EReadyToPlay` only if there was not an error—this is done because after a pause the audio file is still available for playing back.

The following sequence diagram shows the pause operation. Note that the initialization process can also be performed in any of the ways explained in Section 3.3.1, “Initialization.”



### 3.3.4 Stopping

Stopping playback of an audio file is as simple as calling the `CMdaAudioPlayerUtility::Stop()` [3] method as it is shown in the next code snippet. This will stop the audio sample as soon as possible. After a successful stop operation, the play position is moved to the beginning of the clip.

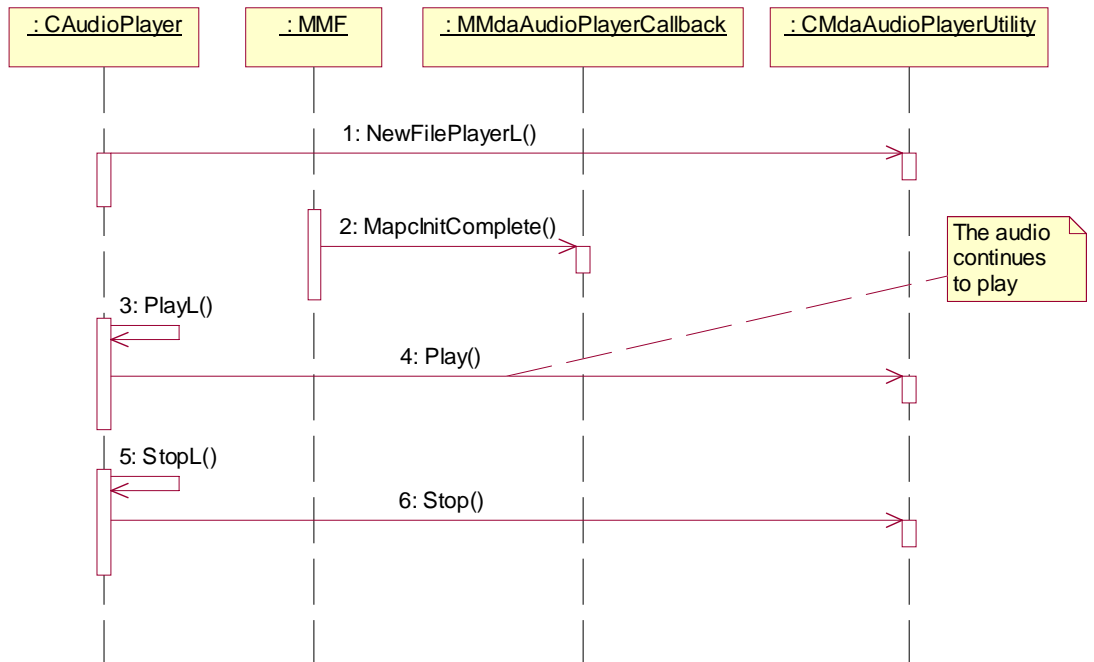
```

void CPlayerAdapter::StopL()
{
    iMdaAudioPlayerUtility->Stop();
    iState = EReadyToPlay;
}

```

This function has no effect if it is called after the playback is completed; under these circumstances the callback function `MapcPlayComplete()` is not called either. It is important to validate that this function will not be called within the client application before the audio player utility is initialized, otherwise it will raise a `CMdaAudioPlayerUtility` panic. One way to guarantee this is to set the stop functionality to “unable” until the initialization process has taken place.

By having the play operations asynchronous, the application object can stop the operation before it has completed by calling an appropriate function on it. This is illustrated in the UML sequence diagram below. Remember that the initialization process on the sequence diagram can be changed to any of the ways already described (see Section 3.3.1, "Initialization.").



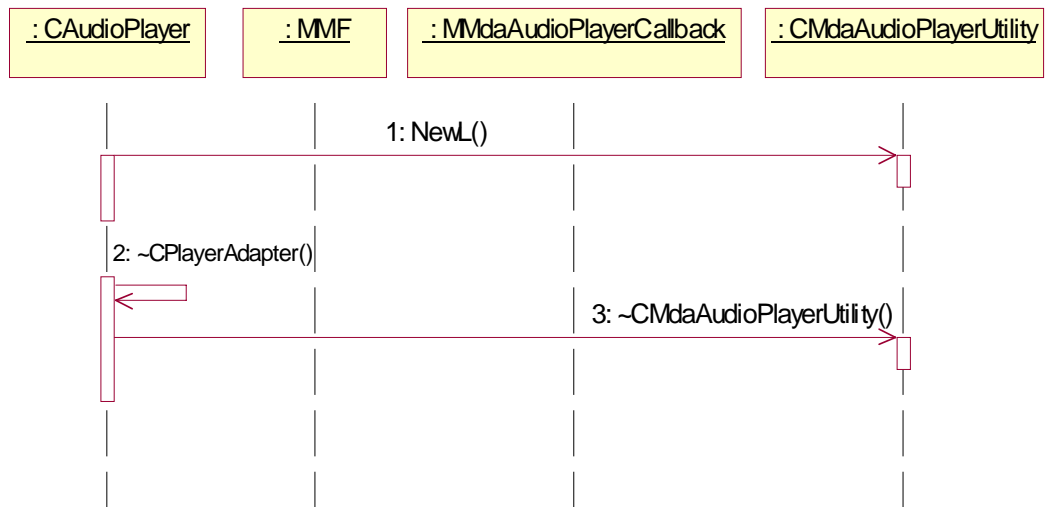
### 3.3.5 Destruction

The destruction process is very simple. You should make sure to destroy the `CMdaPlayerUtility` object in the destructor.

```

CPlayerAdapter::~CPlayerAdapter ()
{
    delete iMdaAudioPlayerUtility;
}
  
```

The sequence diagram below shows this process:



In this example, the `NewL()` function is used to initialize the object, just to show that it is all that is needed to be able to destroy the object. However, it is also possible to initialize the object in any of the valid ways already described (see Section 3.3.1, "Initialization.").

### 3.4 Seeking

Unlike the other audio actions examined so far, the seeking process does not consist of a single function like `Play()`, `Stop()`, or `Pause()`. Instead, the forwarding and rewinding process is a three-step operation.

First, the actual playing position of the file must be known; this can be accomplished with the `CMdaAudioPlayerUtility::GetPosition()` function.

```
iMdaAudioPlayerUtility ->GetPosition( aPosition );
```

After the position has been obtained, the next step is to increase or decrease that position in order to implement the behavior of the rewind or forward. Increasing the value of `aPosition` will result in a Forward, while decreasing it will result in a Rewind. In this case, skipping time is one second, as the position in the audio data is measured in microseconds. The seeking operation can vary in speed according to the codec used. Some codecs are faster than others.

```
aPosition = aPosition.Int64() - (TInt64) 1000000 ;
```

```
aPosition = aPosition.Int64() + (TInt64) 1000000 ;
```

Once the value of the position has been changed, you must tell the `CMdaAudioPlayerUtility` object to set the current playing position to the new playing position.

```
iMdaAudioPlayerUtility ->SetPosition(aPosition);
```

The next code snippet shows an example of how to implement the seeking functionality:

```
void CAudioPlayer::ForwardL()
{
    TTimeIntervalMicroSeconds aPosition;
    TInt aError;

    aError = iMdaAudioPlayerUtility->GetPosition( aPosition );
    if(aError != KErrNone)
    {
```

```

    ... Actions required for handling a general error ...
}

aPosition = aPosition.Int64() + 10000000;

aError = iMdaAudioPlayerUtility->SetPosition( aPosition );
if(aError != KErrNone)
{
    ... Actions required for handling a general error ...
}
}

void CAudioPlayer::RewindL()
{
    TTimeIntervalMicroSeconds aPosition;
    TInt aError;

    aError = iMdaAudioPlayerUtility->GetPosition( aPosition );
    if(aError != KErrNone)
    {
        ... Actions required for handling a general error ...
    }

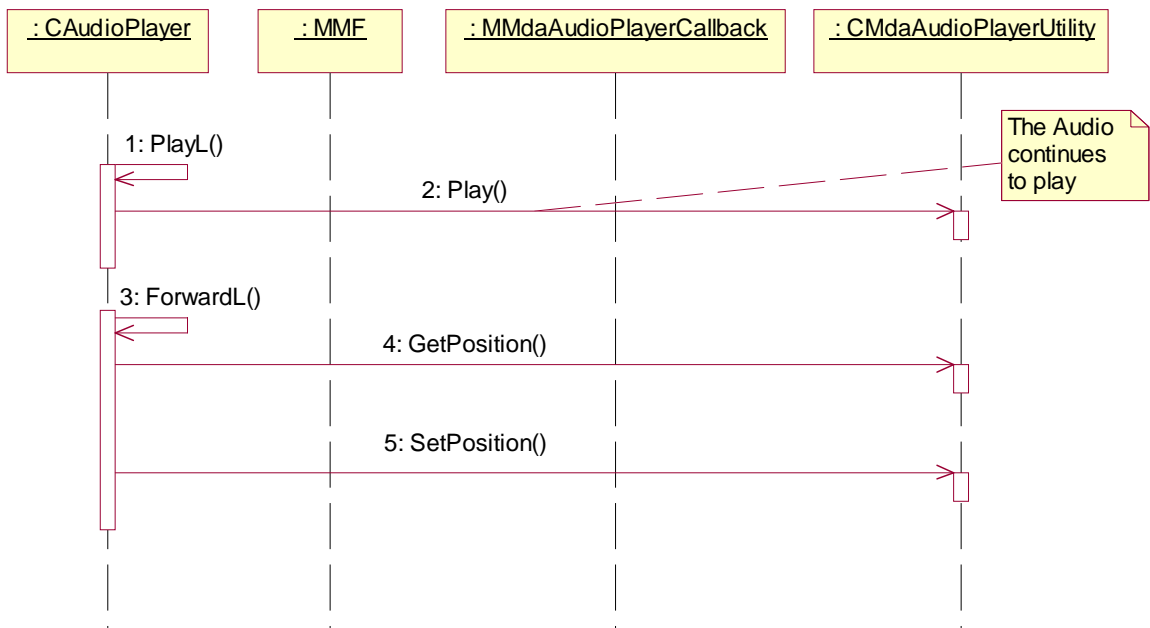
    aPosition = aPosition.Int64() - 10000000;

    aError = iMdaAudioPlayerUtility->SetPosition( aPosition );
    if(aError != KErrNone)
    {
        ... Actions required for handling a general error ...
    }
}
}

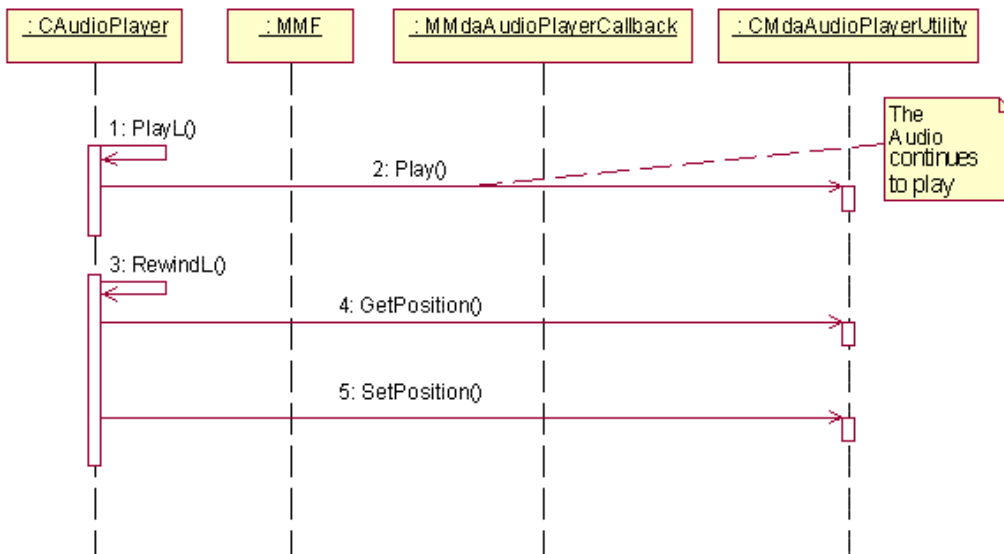
```

Both functions `GetPosition()` and `SetPosition()` return a `TInt` to signal any errors that might have occurred during their process. This can be used to take care of the error handling that might be needed for the application.

The next sequence diagram shows the forwarding functionality:



The following sequence diagram shows the rewinding functionality:



### 3.5 Playing a series of files

The following section offers examples of several scenarios for playing a series of audio files.

#### 3.5.1 Playing a new file after the previous file completes playing

Since you are already familiar with how to play a file, this is a very simple process. After `MapcPlayComplete()` [3] callback is called indicating that the playback was completed, close the audio file by calling function `Close()` [3]. This method closes the current audio clip, allowing another clip to be opened.

```

void CAudioPlayer::Close()
{
    iMdaAudioPlayerUtility->Close();
    iState = ENotReady;
}
  
```

Once the audio file has been closed, the next step is to open the file you want to play. Any of the available methods described in Section 3.3.1, "Initialization" can be used to open a file.

##### Option 1

```
iMdaAudioPlayerUtility ->OpenFileL( aFilename );
```

##### Option 2

```
iMdaAudioPlayerUtility ->OpenDesL( aDescriptor );
```

After you've opened the new file, you are ready to play it by simply calling the `PlayL()` method.

```

void CAudioPlayer::PlayL()
{
    if(iState==EReadyToPlay)
    {
        iMdaAudioPlayerUtility ->Play();
    }
}
  
```

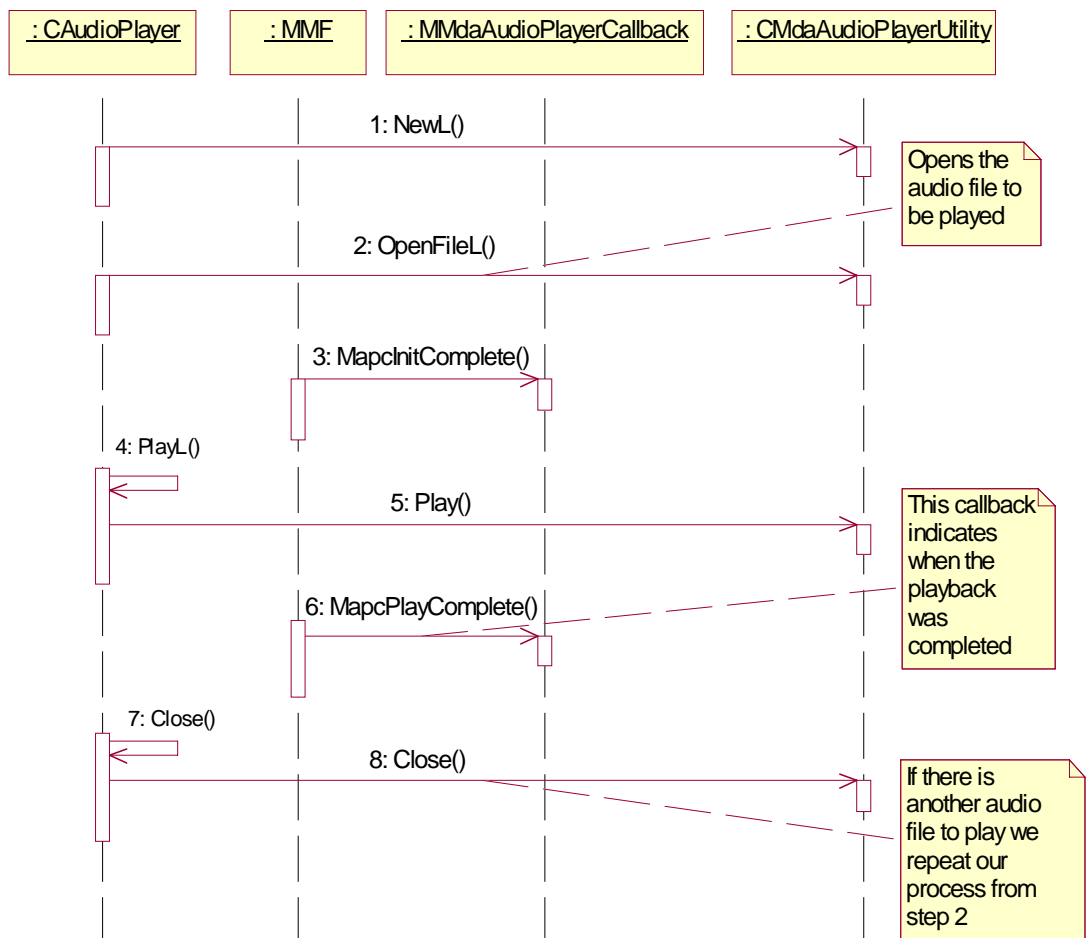
```

        iState=EPlaying;
    }
}

```

With regard to handling error cases, the same method described in Section 3.3, “Play utility life cycle for file playing,” applies to this section since the same functions are used for initialization, playing, etc. Just remember to check exactly which functions are involved and the error handling for those functions.

The next sequence diagram illustrates the functions and events involved when playing a new file after the previous file completes playing.



### 3.5.2 Playing a new file when another file is already playing

To play a new file while an audio file is already playing, first stop the audio file that is in playback.

```

void CPlayerAdapter::StopL()
{
    iMdaAudioPlayerUtility->Stop();
    iState = EReadyToPlay;
}

```

Next, close the audio file:

```

void CAudioPlayer::Close()

```

```

{
    iMdaAudioPlayerUtility->Close();
    iState = ENotReady;
}

```

After closing the file, open the new audio file to play, as described in Section 3.3.1, "Initialization."

Option 1

```
iMdaAudioPlayerUtility ->OpenFileL( aFilename );
```

Option 2

```
iMdaAudioPlayerUtility ->OpenDesL( aDescriptor );
```

Finally, call the `PlayL()` function to play the new audio file.

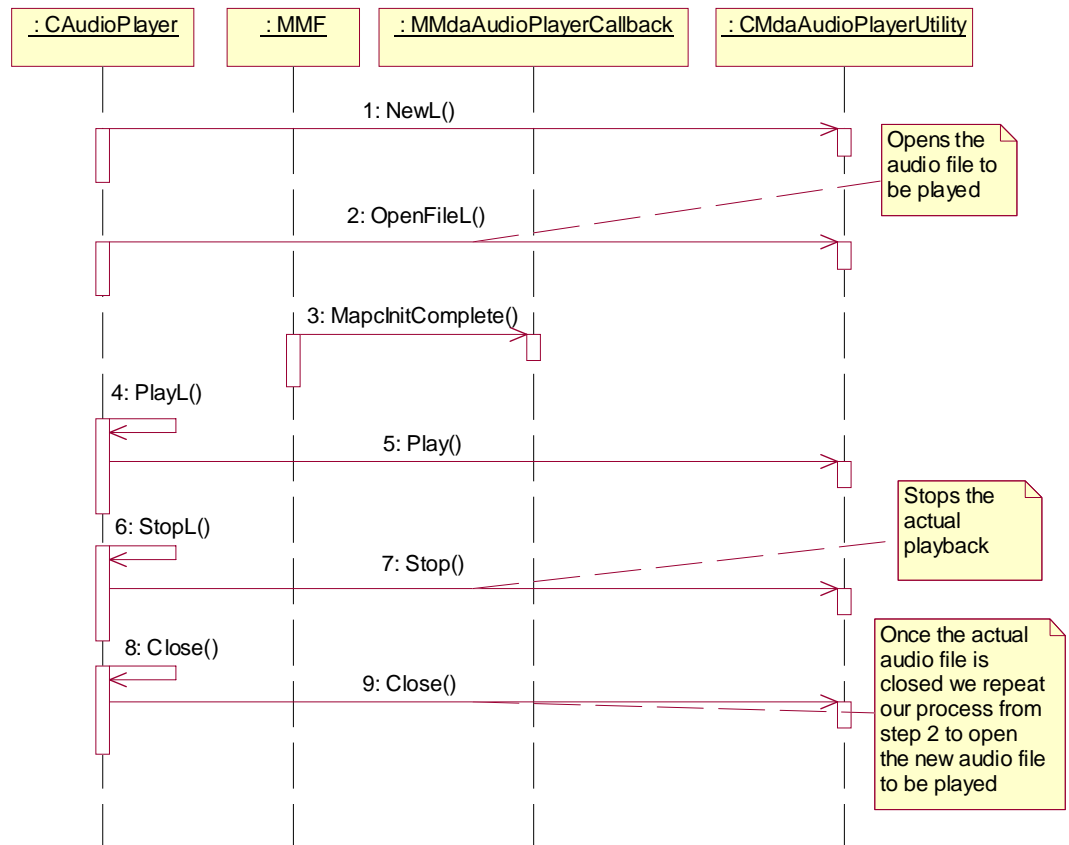
```

void CAudioPlayer::PlayL()
{
    if(iState==EReadyToPlay)
    {
        iMdaAudioPlayerUtility ->Play();
        iState=EPlaying;
    }
}

```

The same handling of error cases described in Section 3.3, "Play utility life cycle for file playing," applies to this section since the same functions are used for initialization, playing, etc. Simply check which functions are involved and check the error handling for those functions in Section 3.3, "Play utility life cycle for file playing."

This case is represented in the following sequence diagram:



### 3.5.3 Playing a new file after pausing a previous file playing

It is possible to play a new audio file after pausing the previous file that was playing. While the audio clip is still on playback, call the `PauseL()` function.

```

TInt CAudioPlayer::PauseL()
{
    TInt aError = iMdaAudioPlayerUtility->Pause();
    if(aError != KErrNone)
    {
        ... Actions required for handling a general error ...
    }
    iState = EReadyToPlay;
}
  
```

Then close the audio file that was playing by calling the `Close()` [3] function and changing the `iState` variable to `ENotReady`. This method closes the current audio clip, allowing another clip to be opened.

```

void CAudioPlayer::Close()
{
    iMdaAudioPlayerUtility->Close();
    iState = ENotReady;
}
  
```

By changing the `iState` variable, you specify to the application that the new audio file you will play is still not ready to play, since once the audio file is closed, it is no longer available for playback. This is a variable we declare in our

application in order to handle the possible states described in Section 3.3, "Play utility life cycle for file playing."

Now the application is ready to load a new audio file and play it. The next step is to open the new audio file you want to play, using any of the available methods described in Section 3.3.1, "Initialization."

Option 1

```
iMdaAudioPlayerUtility ->OpenFileL( aFilename );
```

Option 2

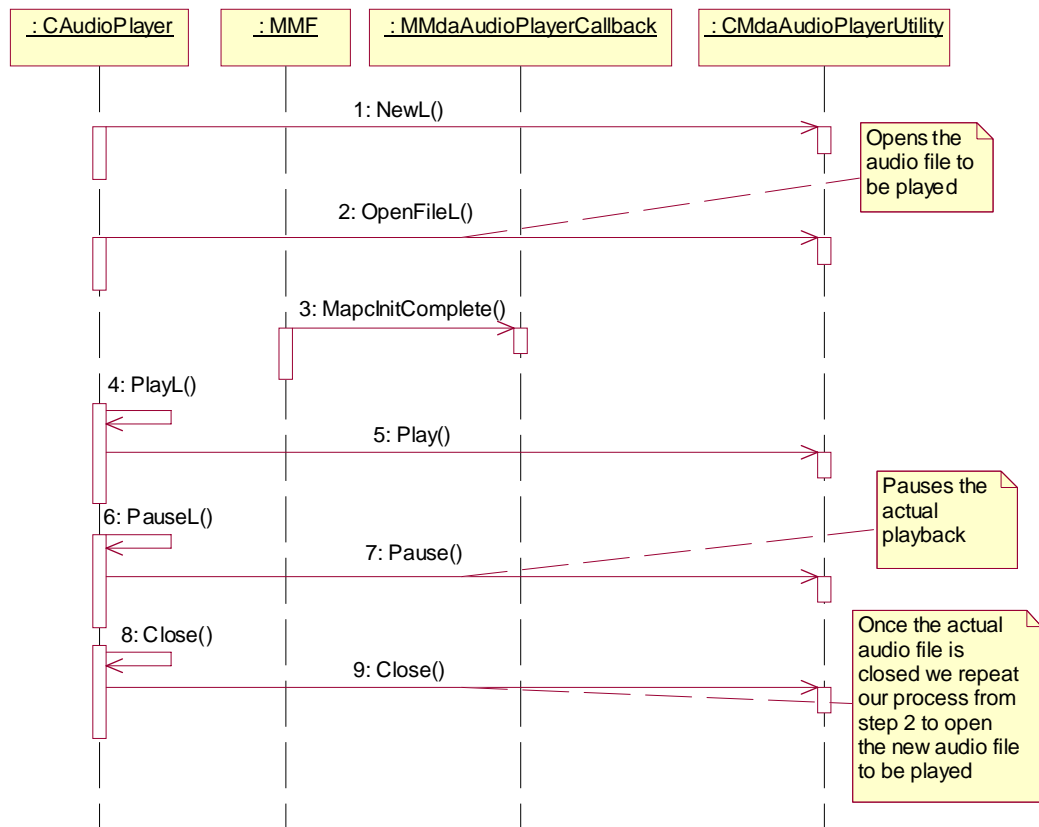
```
iMdaAudioPlayerUtility ->OpenDesL( aDescriptor );
```

After opening the new file, you are ready to play it by just calling the `PlayL()` method.

```
void CAudioPlayer::PlayL()
{
    if (iState==EReadyToPlay)
    {
        iMdaAudioPlayerUtility ->Play();
        iState=EPlaying;
    }
}
```

This is also the same situation we already described about handling error cases because the same way of handling error cases as described in Section 3.3, "Play utility life cycle for file playing," applies to this section since we use the same functions for initialization, playing, pause, etc. Simply check which functions are involved and check the error handling for those functions in Section 3.3, "Play utility life cycle for file playing."

The next sequence diagram illustrates the functions and events involved when playing a new file after pausing a previous file completes playing.



### 3.5.4 Playing a new file after stopping a previous file playing

This process is similar to pausing a playing file and then playing a new one, but instead of calling the `PauseL()` function you will call the `StopL()` function to stop the audio file that is playing.

```

void CPlayerAdapter::StopL()
{
    iMdaAudioPlayerUtility->Stop();
    iState = EReadyToPlay;
}
  
```

After you stop the audio file that was playing, you need to close it.

```

void CAudioPlayer::Close()
{
    iMdaAudioPlayerUtility->Close();
    iState = ENotReady;
}
  
```

The next step is to open the new audio file and play it. Use any of the available methods described in Section 3.3.1, "Initialization," to open an audio file.

Option 1  
`iMdaAudioPlayerUtility ->OpenFileL( aFilename );`

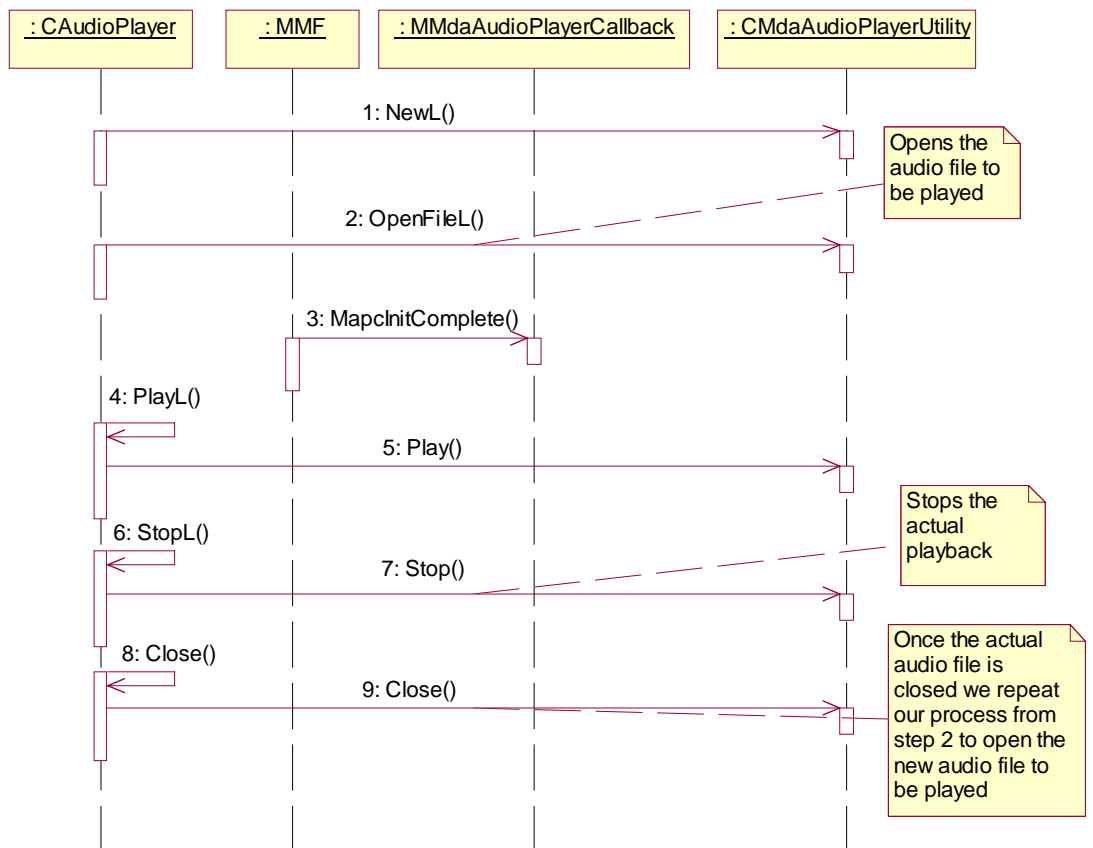
Option 2  
`iMdaAudioPlayerUtility ->OpenDesL( aDescriptor );`

Finally, after opening the new file you are ready to play it by just calling the `PlayL()` method.

```
void CAudioPlayer::PlayL()
{
    if (iState==EReadyToPlay)
    {
        iMdaAudioPlayerUtility ->Play();
        iState=EPlaying;
    }
}
```

The same error handling cases described in Section 3.3, “Play utility life cycle for file playing,” apply to this section, since you use the same functions for initialization, playing, stopping, etc. Just check which functions are involved and the error handling for those functions in Section 3.3, “Play utility life cycle for file playing.”

The next sequence diagram shows the functions involved:



## 4. Reading metadata

With the appearance of new audio file formats, new features have been added to these files. One of these features is support for metadata embedded in files. Metadata information is audio-track information embedded in some audio files. Metadata in audio files usually contains information like song title, name of singer or artist, and genre of music. Since Symbian v7.0, the Audio Play Utility [3] supports extraction of metadata information from files. The Audio Play Utility [3] and DRM Audio Play Utility [4] handle metadata retrieve operations in a very similar way—both require loading a file and they secure the resources needed to play that file.

Retrieving metadata information from a file with the Play Utility is a straightforward process. The first step is to initialize the Play Object and load an audio file into it, using the exact same process described in the initialization section of this document – within Section 3.3, “Play utility life cycle for file playing.”

```
CMdaAudioPlayerUtility* iMdaAudioPlayerUtility =
CMdaAudioPlayerUtility::NewL( *this );
```

As mentioned previously, it is possible to retrieve metadata from a DRM file, in which case you can use the DRM Audio Play Utility. All the functions provided by this API are handled in the same way as those provided by the Audio Play Utility API. The only difference is within the initialization process, where instead of creating an instance to the Audio Play Utility, you should create an instance to the DRM Audio Play Utility.

```
CDrmPlayerUtility* iMdaAudioPlayerUtility =
CDrmPlayerUtility::NewL( *this );
```

After this, all the functions are handled in the same way by both the DRM Audio Play Utility and the Audio Play Utility APIs. After creating the instance, the next step is to open the audio file.

```
iMdaAudioPlayerUtility ->OpenFileL( aFilename );
```

Once the object has been initialized and the audio data has been properly loaded into the object, you must count how many metadata entries the file has.

```
iMdaAudioPlayerUtility ->GetNumberOfMetaDataEntries( aNumEntries );
```

Now you can proceed to retrieve them. Each metadata entry is retrieved and matched against a predefined set of entry names. These are the most common entry names on ID3 tags [11]:

EntryName item	Description
KMMFMetaEntrySongTitle	Title of song.
KMMFMetaEntryAlbum	Name of album.
KMMFMetaEntryArtist	Name of singer or artist.
KMMFMetaEntryAlbumTrack	Track number of the track in the album.
KMMFMetaEntryYear	Year album was published.
KMMFMetaEntryGenre	Genre of music.
KMMFMetaEntryCopyright	Track copyright information.

EntryName item	Description
KMMFMetaEntryComment	Track comments.
KMMFMetaEntryComposer	Composer information of the track.
KMMFMetaOriginalArtist	Original artist information.

In order to determine which entry name the value retrieved belongs to and what information it represents, the name and value must be extracted from each entry. The following code cycles through each of the metadata entries contained in the file. Remember that you obtained `aNumEntries` from the previous step, and that this variable shows the number of entries in the file. In this code the other function that Play Utility provides for metadata handling is `iMdaAudioPlayerUtility->GetMetaDataEntryL(index) [3]`, which returns a `CMMFMetaDataEntry [12]` object that contains the name of the tag and its value. Note that in this code snippet `aMetadataName` represents any of the `EntryName` items from the table above. By retrieving the entry name you will know exactly which metadata entry you are getting.

```

HBufC* metadataValue = NULL;

for ( TInt j = 0; j < aNumEntries; j++ )
{
    CMMFMetaDataEntry* entry = NULL;
    TRAPD( errorValue, entry =
        iMdaAudioPlayerUtility->GetMetaDataEntryL(j) );

    CleanupStack::PushL(entry);
    if ( ( errorValue == KErrNone ) &&( entry ) )
    {
        TName name( entry->Name() );
        TName value( entry->Value() );
        CleanupStack::PopAndDestroy(entry);
        if ( name.CompareF( aMetadataName ) == 0 )
        {
            // Zero terminate the string for proper
            // displaying
            TInt i = value.LocateF('\0');
            if ( i != KErrNotFound )
            {
                value.SetLength(i);
            }
            metadataValue = value.AllocLC();
            break;
        }
    }
    else
    {
        CleanupStack::PopAndDestroy(entry);
        if ( ( errorValue != KErrNotFound ) &
            ( errorValue != KErrNotSupported ) )
        {
            User::LeaveIfError( errorValue );
        }
    }
}

```

Like most functions, `GetMetaDataEntryL()` and `GetNumberOfMetaDataEntries()` provide a way to check whether the operations were completed successfully. In the case of `GetMetaDataEntryL()`, it either responds with a `KErrNone` or with one of the

system-wide error codes. The function `GetMetaDataEntryL()` does respond with more specific error codes. It can leave with `KerrNotFound` to indicate that the selected index for the metadata entry does not exist; it can also leave with `KerrNotImplemented` if the controller does not support the metadata information for this format. This function can also leave with the other system-wide error codes [3].

```
aError = iMdaAudioPlayerUtility -> GetMetaDataEntryL() ( aPosition );

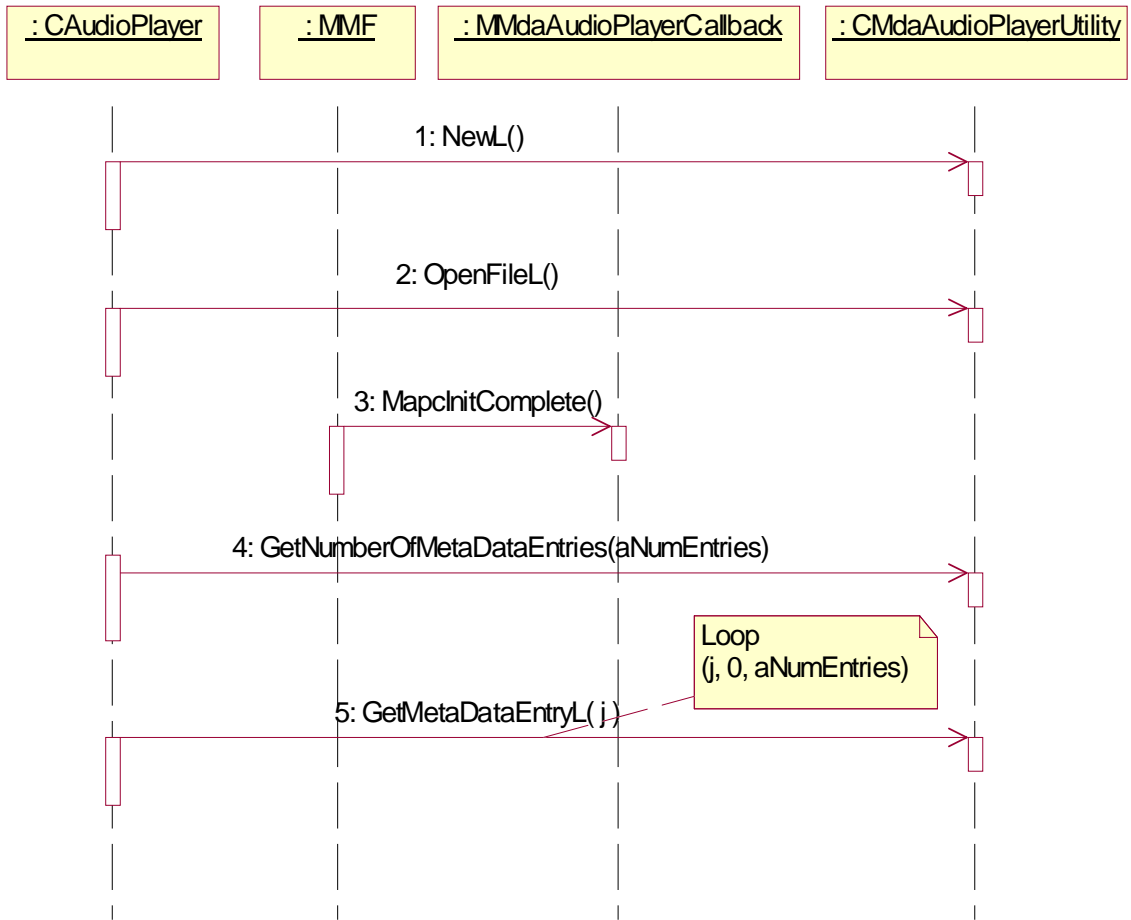
if(aError != KErrNone)
{
    ... Actions required for handling a general error ...
}

aError = GetNumberOfMetaDataEntries() ->
GetMetaDataEntryL() (aPosition);

if(aError != KErrNone)
{
    if(aError == KerrNotFound)
    {
        ... Actions required for an invalid index ...
    }
    else if(aError == KerrNotImplemented)
    {
        ... Actions required for invalid metadata format...
    }

    ... Actions required for handling a general error ...
}
```

The next sequence diagram illustrates the process of retrieving metadata with the Audio Player Utility API.



---

## 5. Streaming buffers from the application

The S60 platform allows streaming of buffers directly from the application. It should be noted that this type of buffer streaming differs from the comprehensive definition of Internet streaming. The mechanism is a means to stream buffers from one software component to another. This functionality has some advantages compared to local playback. First of all, stream audio will not be permanently stored locally, so it is not available for copying. Second, the end user cannot forward or send the content to another user. Finally, it is much easier to keep the content up to date. All of these reasons offer great advantages for service providers.

The transfer bandwidth for streaming depends on the access technology used and the configuration of the operator network. Care must be taken when a file is encoded at a high bit rate, typically 192 Kbps or more. These files may not stream well, resulting in poor playback, since the available bandwidth of the network is not high enough to support such files.

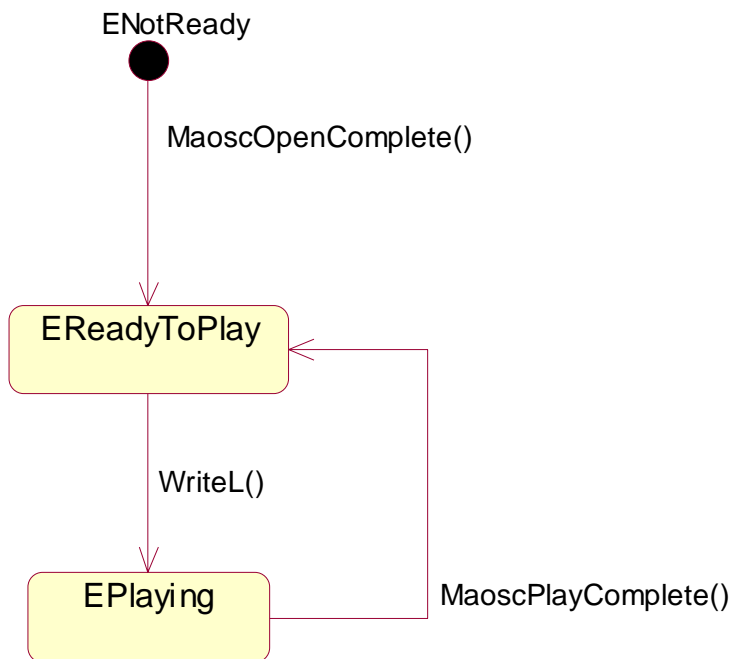
S60 devices can support several common audio formats when streaming buffers. For more details about the exact audio streaming formats supported by devices, consult the Audio Capabilities for each device [13].

For streaming buffers for audio playback, use the `CMdaAudioOutputStream` [14] API. The only difference between playing local content and streaming buffers from the application is that the playback methods use different methods for file opening and initialization.

Section 5.1 describes the `CMdaAudioOutputStream` API, which allows an interface to an audio stream player passing audio data from specified buffers to the audio hardware.

### 5.1 CMdaAudioOutputStream life cycle

The three states you need to handle in your client application are:



These states can be declared as:

```

enum TStreamAppStatus
{
    ENotReady,
    EReadyToPlay,
    EPlaying
};

TStreamAppStatus iStatus;
  
```

### 5.1.1 Initialization

Before exploring the initialization process, you will need to write a client class that implements `MMdaAudioOutputStreamCallback`. This class will provide you with three callback functions that notify the client of the progress of the audio output streaming, allowing you to handle possible errors as well. These callbacks are `MaoscOpenComplete()`, `MaoscBufferCopied()`, and `MaoscPlayComplete()`, and they must be implemented by the users of the `CMdaAudioOutputStream` class [14].

The description of the `MaoscOpenComplete()` callback occurs below, since it is related to the initialization process; the remaining callbacks will be described in subsequent sections.

The initialization process is quite simple. First it is necessary to create an instance of the `CMdaAudioOutputStream` class and initialize the `iStatus` variable.

The next code snippet shows how to create an instance:

```

CMdaAudioOutputStream* iAudioStream;
iAudioStream = CMdaAudioOutputStream::NewL( *this );

iStatus = ENotReady;
  
```

When creating the `CMdaAudioOutputStream` instance, it is possible to specify the priority and preference for the instance since the `NewL()` function accepts these as parameters. The function leaves if the audio streaming object cannot be created.

After the instance has been created, the last step is to open the output audio stream package. The following function does this.

```
iAudioStream->Open(&iSettings);
```

Note that the member variable `iSettings` (`TMdaAudioDataSettings`) does not have to contain any configuration settings at this point.

The Multimedia framework calls the `MaoscOpenComplete()` callback function after `CMdaAudioOutputStream::Open()` has completed, indicating that the audio output is ready for use. The parameter given by the framework is an error value that indicates if the `Open()` was completed successfully, with a `KErrNone` value if it succeeded. This is the place to set sample rate, volume, etc. [14]. The example uses a sampling rate of 16 khz. (In the following examples, a client class named `CStreamApp` is used for the client application; obviously developers may choose their own name.)

```
void CStreamApp::MaoscOpenComplete(TInt aError)
{
    if(aError==KErrNone)
    {
        //set stream properties
        iAudioStream -> SetAudioPropertiesL (
            TMdaAudioSettings::EsampleRate16000Hz,
            TMdaAudioDataSettings::EchannelsMono );

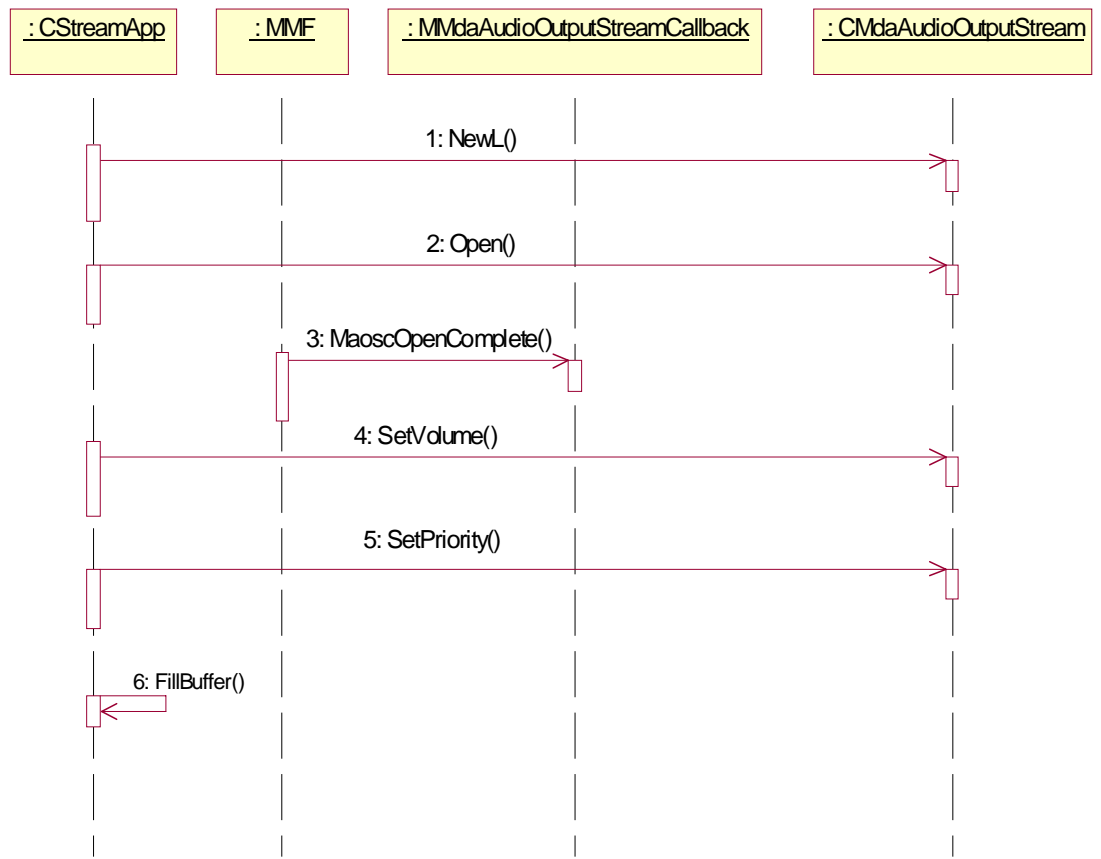
        //Set the appropriate volume. Note that the
        //MaxVolume () differs from the emulator to
        //the target device
        iAudioStream -> SetVolume(iAudioStream -> MaxVolume());
        iAudioStream -> SetPriority( EpriorityNormal,
            EMdaPriorityPreferenceNone);

        //Fill first buffer
        FillBuffer();

        iStatus = EReadyToPlay;
    }
    else
    {
        . . . Actions required for a general error . . .
    }
}
```

In the code snippet above, the `FillBuffer()` function is part of the client application. It is the application developer's job to fill the buffer with useful data. This is what has been done in the `FillBuffer()` function in the code snippet above. This has to be implemented by the developer of the client application. You need to call this function as done in the callback above, immediately after opening the stream, to have the buffer with information and ready to be played. That is why the `iStatus` variable is set to `EReadyToPlay` status at the end of the snippet above.

All the steps described above are represented in the following sequence diagram:



### 5.1.2 Playing

Once the initialization is complete, the `CMdaAudioOutputStream` object is ready to play audio data. You can do this by implementing a `PlayL()` function in the client application. As shown in the code below, the playback of the stream starts immediately after you write the buffer to `CMdaAudioOutputStream`. This is done by a call to `CMdaAudioOutputStream::WriteL(const TdesC8& aData) [14]`. Note that `aData` is the reference to the descriptor containing data to be played, set within the `FillBuffer()` function.

```

void CStreamApp::PlayL()
{
    if(iStatus==EReadyToPlay)
    {
        iAudioStream -> WriteL(aData);
        iStatus = EPlaying;
    }
}
  
```

`WriteL()` is an asynchronous function. After the data in the descriptor "aData" is copied, `MMdaAudioOutputStreamCallback::MaoscBufferCopied()` [14] callback will be called by the framework, notifying the client application that the `aData` has been received and copied into the stream. Once you receive this notification, the buffer can be reclaimed by the client application (you can reuse/delete the buffer). The client should not try to operate on the buffer `aData` after the buffer is written to `CMdaAudioOutputStream` but before receiving `MaoscBufferCopied` callback with the buffer reference. Remember that you

need to fill your buffer first with the next item and then write it into the stream. You can do that within this callback.

```
void CStreamApp::MaoscBufferCopied(TInt aError,
                                   const TdesC8& /*aBuffer*/)
{
    if(aError == KErrNone)
    {
        FillBuffer();
        iAudioStream -> WriteL(aData);
    }
    else if(aError == KErrAbort)
    {
        . . . Actions required when Stopping the playback . . .
    }
    else
    {
        . . . Actions required for a general error . . .
    }
}
```

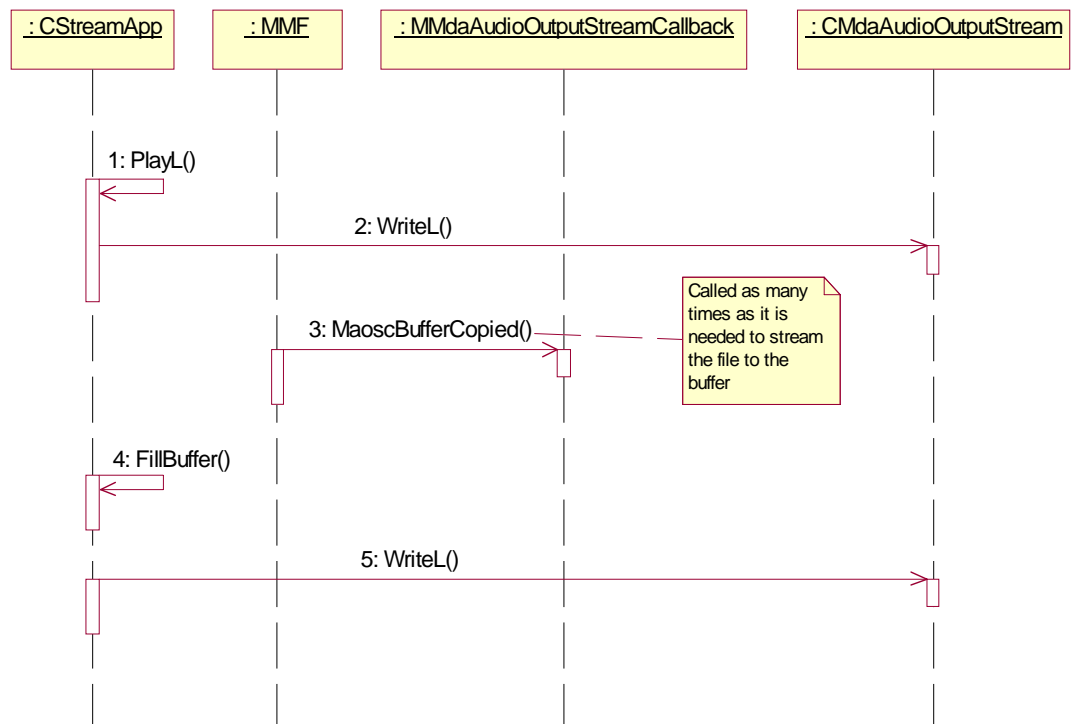


**Note:** Callbacks are notification messages from the framework. The framework will wait for the callback to finish before proceeding further. Client applications should not include CPU-intensive code in the callbacks and should return from the callback function as quickly as possible. The applications can implement CPU-intensive code in a separate active object and launch the active object from the callback.

Optionally, the client can create multiple buffers, fill data, and pass it to `CMdaAudioOutputStream`. Buffers containing data can be written to `CMdaAudioOutputStream` at any time.

This callback is also called either when the client has stopped the stream playing before the descriptor has been fully copied when calling `CMdaAudioOutputStream::Stop()` [14], or when an error has occurred. The parameters for this callback are an error value that will indicate if the copy succeeded and a reference to the buffer that has been copied. You will get a `KErrNone` value if the copy succeeded; otherwise, you'll receive one of the system error codes. `KErrAbort` indicates that the client has stopped the stream playing before the descriptor has been copied.

The steps performed for the playback process are shown in the following sequence diagram:



### 5.1.3 Stopping

Stopping audio stream playback is a very simple process—you just need to call the `CMdaAudioOutputStream::Stop()` [14] function to stop writing data to stream and stop playing audio. Any data remaining in the buffers will be discarded. Remember that this function will call the `MMdaAudioOutputStreamCallback::MaoscBufferCopied()` [14] callback, as was described in Section 5.1.2, “Playing,” with the error value indicating the buffer was not copied.

The next code snippet implements a function for the client application called `StopL()`, where the `CMdaAudioOutputStream::Stop()` [14] function is called.

```

void CStreamApp::StopL()
{
    if(iStatus == EPlaying)
    {
        iAudioStream -> Stop();
    }
}

```

After calling the `CMdaAudioOutputStream::Stop()` [14] function to stop the streaming, the `MMdaAudioOutputStreamCallback::MaoscPlayComplete()` [14] will be called as well. The next code snippet shows an example of how this callback can be implemented.

```

void CstreamApp::MaoscPlayComplete(TInt /*aError*/)
{
    if(aError == KErrNone)
    {
        . . . Actions required when close succeeded . . .
    }
    else if(aError == KErrUnderFlow)
    {
        iStatus == EReadyToPlay;
    }
}

```

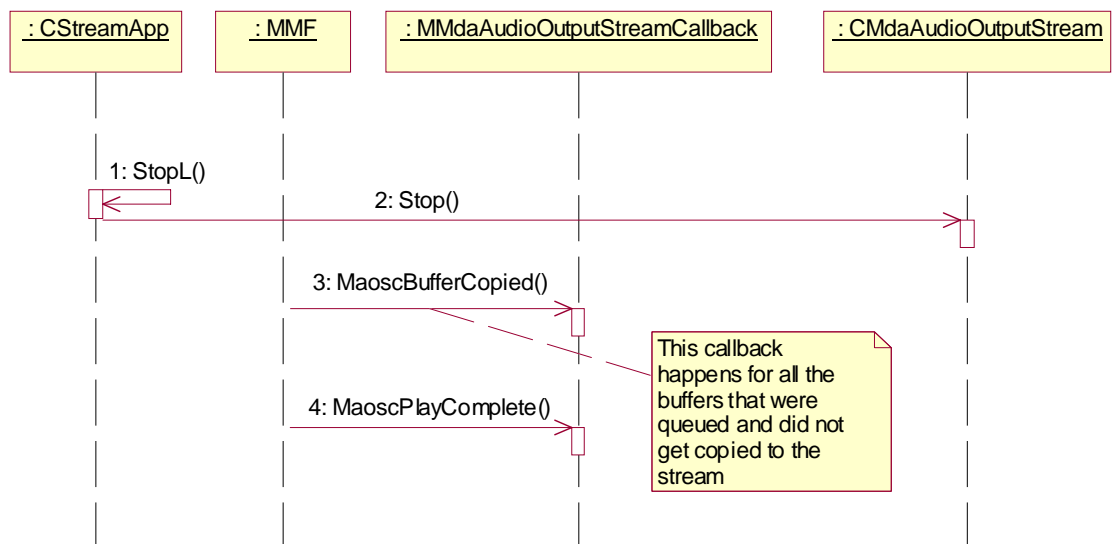
```

}
else if(aError == KErrDied)
{
. . . Actions required for KErrDied. . .
}
else
{
. . . Actions required for a general error . . .
}
}

```

As mentioned previously, this callback is called when the streaming is stopped, but it is also called when playing has stopped for some other reason. If the end of the sound data was reached, it may return a `KErrUnderFlow` value, a `KErrNone` value if the close succeeded, or an appropriate error value.

The next sequence diagram shows the stop process



#### 5.1.4 Destruction

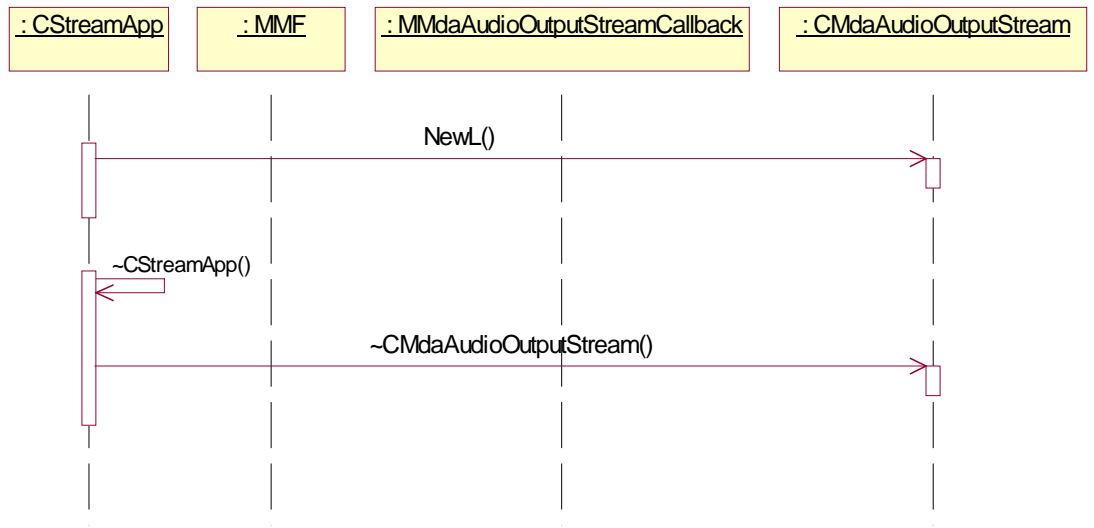
For this operation, the `CMdaAudioOutputStream` object needs to be destroyed, which can be done within the client application destructor. The next code snippet shows this:

```

CStreamApp::~CStreamApp()
{
    delete iAudioStream;
}

```

The following sequence diagram indicates how this is handled.



## 6. Internet music streaming service by streaming within an application

It is possible to create an application to stream music from the Internet by establishing a connection between the client application and a server application using sockets. The client application needs to implement the necessary multimedia streaming protocols to get the music from the server (e.g., Shoutcast, RTSP/RTP/SDP). When a connection is done it is possible to transfer audio data from the server to the client application, and the retrieved audio can be put into buffers and then sent to the `CMdaAudioOutputStream` API for playback.

For more information about how to create a connection between a client and server application by using sockets, see Reference [15] at the end of this document.

When streaming an audio file using the `CMdaAudioOutputStream` API, to send audio data to the platform, you can also specify the data type of the audio file, which is a good resource when using proprietary codec or for ensuring codec support across products. In order to do this, the API uses a class named `TFourCC` [16], which holds a four-character code representing supported data encodings. The four-character codes are packed into a single `TUint32`. These `FourCC` codes are a representation of the datatypes used to identify codecs.

The `CMdaAudioOutputStream` API provides function `SetDataTypeL()` [14], which takes a `TFourCC` [16] object as a parameter. This function lets you set the data type of the audio file.

Since this document has already described how to stream an audio file using the `CMdaAudioOutputStream` API, as well as all of the tasks involved in such a process in Chapter 5, this chapter will only describe how to use the `SetDataTypeL()` function.

The `CMdaAudioOutputStream::SetDataTypeL(TFourCC aAudioType)` function can be called immediately after the stream has been opened. This can be done within the `MaoscOpenComplete()` callback. For more information about the initialization process for streaming an audio file using the `CMdaAudioOutputStream` API, see Chapter 5, "Streaming buffers from the application."

The next code snippet shows the usage of this function:

```
void CStreamApp::MaoscOpenComplete(TInt aError)
{
    if(aError==KerrNone)
    {
        //sets the audio data type
        iAudioStream-> SetDataTypeL(KMMFFourCCCodeMP3);

        //set stream properties
        iAudioStream -> SetAudioPropertiesL (
            TMdaAudioSettings::EsampleRate16000Hz,
            TMdaAudioDataSettings::EchannelsMono );

        //Set the appropriate volume. Note that the MaxVolume ()
        //differs from the emulator to the target device
        iAudioStream -> SetVolume(iAudioStream -> MaxVolume());
        iAudioStream -> SetPriority( EpriorityNormal,
            EMdaPriorityPreferenceNone);

        //Fill first buffer
        FillBuffer();

        iStatus = EReadyToPlay;
    }
}
```

```

else
{
    . . . Actions required for a general error . . .
}
}

```

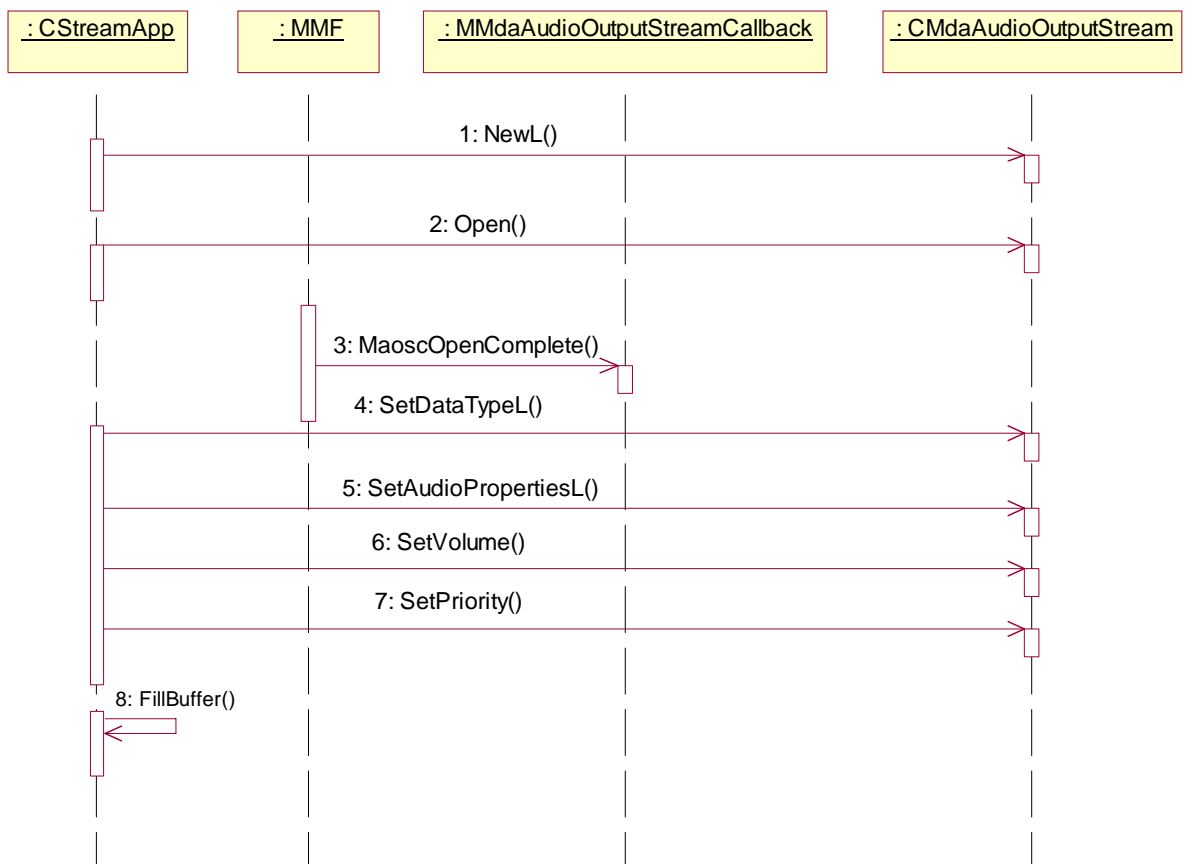
If the data type is not explicitly set, `CMdaAudioOutputStream` defaults to PCM16. While setting the data type for streaming encoded audio, if the hardware does not support the FourCC supplied as a parameter to the function, the function leaves with error code `KerrNotSupported`.

As mentioned before, the function receives, as a parameter, the FourCC code to specify the data type of the streamed audio. Some example codes are:

1. `KMMFFourCCCodePCM8`
2. `KMMFFourCCCodePCM16`
3. `KMMFFourCCCodeAMR`
4. `KMMFFourCCCodeMP3`

For a complete list of codes, see reference [17] at the end of this document.

The process is shown in the following sequence diagram:



## 7. References

[1]	Getting Started With C++ Development On The S60 SDK	<a href="http://www.symbian.com">www.symbian.com</a> or Symbian OS documentation
[2]	<code>Mmfbase.h</code>	<a href="http://www.symbian.com">www.symbian.com</a> or Symbian OS documentation
[3]	<code>MdaAudioSamplePlayer.h</code>	<a href="http://www.symbian.com">www.symbian.com</a> or Symbian OS documentation
[4]	<code>DrmAudioSamplePlayer.h</code>	<a href="http://www.symbian.com">www.symbian.com</a> or Symbian OS documentation
[5]	<code>MdaAudioTonePlayer.h</code>	<a href="http://www.symbian.com">www.symbian.com</a> or Symbian OS documentation
[6]	<code>MidiClientUtility.h</code>	<a href="http://www.symbian.com">www.symbian.com</a> or Symbian OS documentation
[7]	<code>DRMHelper.h</code>	<a href="http://www.symbian.com">www.symbian.com</a> or Symbian OS documentation
[8]	<code>DRMLicenseChecker.h</code>	<a href="http://www.symbian.com">www.symbian.com</a> or Symbian OS documentation
[9]	<code>DRMCommon.h</code>	<a href="http://www.symbian.com">www.symbian.com</a> or Symbian OS documentation
[10]	<code>e32std.h</code>	<a href="http://www.symbian.com">www.symbian.com</a> or Symbian OS documentation
[11]	<code>MmfMeta.h</code>	<a href="http://www.symbian.com">www.symbian.com</a> or Symbian OS documentation
[12]	<code>mmfcontrollerframeworkbase.h</code>	<a href="http://www.symbian.com">www.symbian.com</a> or Symbian OS documentation
[13]	Audio Capabilities	<a href="http://www.forum.nokia.com/audiovideo">www.forum.nokia.com/audiovideo</a>
[14]	<code>MdaAudioOutputStream.h</code>	<a href="http://www.symbian.com">www.symbian.com</a> or Symbian OS documentation
[15]	<a href="http://www.symbian.com/developer/techlib/papers/Sockets/sockets.html">http://www.symbian.com/developer/techlib/papers/Sockets/sockets.html</a>	<a href="http://www.symbian.com">www.symbian.com</a> or Symbian OS documentation
[16]	<code>mmf\common\MmfUtilities.h</code>	<a href="http://www.symbian.com">www.symbian.com</a> or Symbian OS documentation
[17]	<code>\mmf\common\mmffourcc.h</code>	<a href="http://www.symbian.com">www.symbian.com</a> or Symbian OS documentation

## 8. Terms and abbreviations

Term or abbreviation	Meaning
API	Application Programming Interface
AU	Default SUN Systems audio format
CODEC	Coder/Decoder. A codec is a software or hardware module that transforms (compresses/decompresses/transcodes) a particular file format into another format.
DLL	Dynamic Link Library
DRM	Digital Rights Management
DTMF	Dual Tone Multi-Frequency
METADATA	The information that is held within a clip. Metadata is usually used to store information such as copyright information, creator, creation date, etc. Only some audio formats enable the use of metadata.
MIDI	Musical Instrument Digital Interface
MMF	Multi Media Framework
PCM	Pulse Code Modulation. A common linear audio encoding format.
PLUG-IN	A DLL that is pluggable into the existing Multimedia Framework
RAM or ram	File extensions for RealNetworks' simple URL descriptor files
STREAMING	The transmission of real-time data, audio and video, from a server to a client, where the client decodes and plays the data as it is received.
UI	User Interface
URL	Uniform Resource Locator
WAV	Microsoft Windows standard sampled audio format.

---

## 9. Evaluate this resource

Please spare a moment to help us improve documentation quality and recognize the resources you find most valuable, by [rating this resource](#).