

S60 平台：音乐应用开发伙伴指南

版本 1.1

2007 年 1 月 31 日

S60
p l a t f o r m

法律提示

版权©（2005 年，2006 年）属于诺基亚公司，诺基亚公司保留全部权利

“诺基亚”及“诺基亚科技以人为本”是诺基亚公司的注册商标。**Java** 和所有基于 **Java** 的标志是 **Sun** 微系统有限公司的商标或注册商标。在此提到的其它产品和公司名称可能是其所有者的商标或商业名称。

声明

本文档中的信息基于其现有状况，不存在任何保证，包括销售保证、适用某一特殊用途的保证，或从任何建议、规范或范例中衍生出来的保证。此外，本文档所提供的并非最终信息，在其最终发布前会做较大改动。本文档仅用作信息通报。

诺基亚公司不承担所有因实施本文档中所表述的信息而产生的相关责任，包括侵犯任何知识产权的责任。诺基亚公司并不保证或认为使用这些信息不会构成对这些知识产权的侵犯。

诺基亚公司保留不预先通知而随时修改或撤销本规范的权力。

授权许可

本授权仅限于因个人应用而下载和打印本说明，除此之外，不存在对其它任何知识产权的授权许可。

目录

1.	简介	5
2.	音频优先级和播放选择	6
3.	从文件播放音乐	7
3.1	用于文件回放的 API	7
3.2	选择适当的 API.....	8
3.3	Play utility 文件播放的生命周期.....	8
3.3.1	初始化.....	9
3.3.2	播放.....	15
3.3.3	暂停.....	17
3.3.4	停止.....	18
3.3.5	析构.....	19
3.4	搜寻	20
3.5	播放系列文件.....	21
3.5.1	前一个文件播完后播放一个新文件	22
3.5.2	另一个文件在播时播放一个新文件	23
3.5.3	暂停播放前一个文件并开始播放新文件	24
3.5.4	停止播放前一个文件并开始播放新文件	26
4.	读取元数据 (metadata)	28
5.	从应用程序实现流式缓冲.....	32
5.1	CmdAudioOutputStream 的生命周期	32
5.1.1	初始化.....	33
5.1.2	播放.....	34
5.1.3	停止.....	36
5.1.4	析构.....	37
6.	在应用中通过流方式提供互联网音乐服务.....	39
7.	参考文献	41
8.	术语与缩略语.....	42
9.	对本文评分	43

修订记录

2005 年 12 月 16 日	版本 1.0	文档首次发布
2006 年 11 月 14 日	版本 1.1	删除了有关 Metadata Utility API （不再包括在 SDK 中）的信息
2007 年 1 月 31 日	中文版本 1.1	中文版本发布

1. 简介

S60 平台向开发伙伴们提供多种解决方案，用于为用户创建和分发丰富多彩的音乐应用。S60 终端强大的音频处理能力也确保了方案的实施。

本文介绍了如何使用 S60 多媒体 API，在 S60 3rd Edition 及后续版本的手机终端上进行音乐应用的开发。文档既包括讲解也包括范例，演示了如何使用各种多媒体 API 来完成音乐应用中的一些常规操作。此外，本文也谈及实现类似操作的其它一些技术并解释了它们各自在设计上的优缺点。

文档的讲解方式通过将某些应用用例转换成对应的多媒体服务，从而充分展示为满足用例要求而对各种 API 的恰当使用。在一些用例中也许只用到了单个的 API，而在另一些用例中，则可能要求使用多个 API。对后一种情况我们将讨论这些 API 的协同使用，同时详细讲述多 API 协调机制的背后原理。

阅读本文前，读者应该熟悉创建 S60 应用的一些基本方法。关于 S60 应用创建方面的更多信息请参阅《S60 SDK C++开发入门》[1]。因而，本文并不讨论有关 UI 框架、编译环境、及 Symbian 一般架构等相关问题。

如果您正在开发一个功能全面的音乐服务客户端，您可能也希望熟悉其他一些 S60 API，如浏览器控件和下载管理。这些内容被分别包含在一些具体的开发伙伴指南中，如，《S60 平台：Browser Control API 开发伙伴指南》和《S60 平台：Download Manager API 开发伙伴指南》。有关 S60 API 方面的更多信息请访问 www.forum.nokia.com。

2. 音频优先级和播放选择

常常有这种情况：某台终端的音频硬件同时收到来自多个应用（客户端）的请求。一个常见的例子就是：当用户正在使用一个多媒体应用时接到了一个电话呼叫。此刻，该终端既可以继续使用那个多媒体应用，也可以停止该应用转而振铃。**Audio Priority**（音频优先级）和**Audio Preference**（音频播放选择）就用于应付这类情况。**Audio Policy**（音频策略）组件通过分解那些来自不同终端的同时请求实现对终端音频硬件的访问优先级管理。当多个应用同时请求音频播放时，**Audio Policy**（音频策略）将决定允许哪一个播放，拒绝哪一个访问音频资源，以及是否要混合两种或多种音源。

设计一个应用需要考虑方方面面。产品性能，硬件能力，以及系统资源的可用性等不同需求都将对音频优先级的实现产生影响。

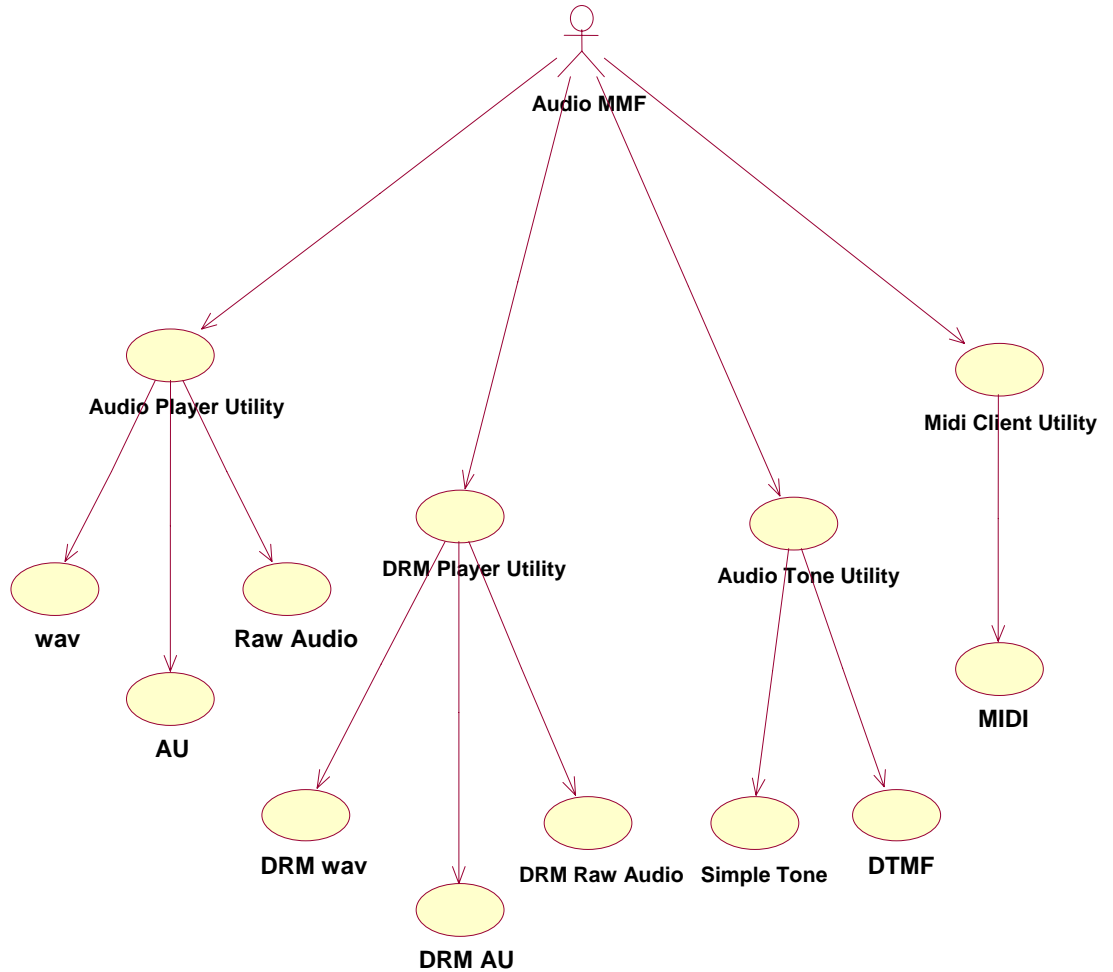
Audio Priority的范围从**EmdaPriorityMin**到**EmdaPriorityMax**。前者是最低优先级，任何其它客户端都能中断该客户端；而后者则相反，其它客户端无法中断该客户端。

EmdaPriorityNormal优先级意味着：该客户端能够被中断，但只能被更高级别的客户端中断。这些值在**TMdaPriority**枚举结构中定义[2]。

Audio Priority 决定哪些事件更重要，因而需要首先处理。但其功能却不仅止于此，当更高优先级的客户端抢占终端时，音频播放选择有助于您定义一些行为，以便让正使用音频终端的客户端采用。

3. 从文件播放音乐

S60 平台用 **multimedia framework (MMF) API** 为应用程序提供多媒体服务。它作为一个多媒体处理单元的插件集，提供操作终端硬件的通用接口。这个 API 内部有多个接口用于从文件播放音乐，下面的用例演示了这一点。



3.1 用于文件回放的 API

S60 MMF 提供了一个客户端 API，包括用于音频操作的不同接口。MMF 既能本地回放，也能流式回放不同的音乐作品（音频格式不同）。下面简要说明了用于从文件播放音乐的各个接口。

- Audio Player Utility – 通用接口，用于播放受数字版权管理（DRM）保护的内容，如果应用本身具有 DRM capability。这个接口提供了一些方法，用于创建、播放，及处理储存在各种文件和描述符中的音频数据。
CMdaAudioPlayerUtility 类 [3] 提供这些功能。基于插件机制的音频类，支持的

输入输出音频格式并无限制。MMF 所支持的标准音频文件格式是 AU 和 WAV。而输入输出音频数据则可以是插件所支持的任何格式。

- **DRM Player Utility** –通用接口，允许不具有 DRM capability 的应用播放受 DRM 保护的内容，但性能表现有少许降低。它也提供了一些方法用于访问被储存在各种文件（加密和未加密的）及描述符（未加密）中的音频数据。**CDrmPlayerUtility [4]** 类提供该功能。由于音频类基于插件，所以获支持的回放音频格式并无限制。MMF 所支持的标准音频文件格式是 AU，WAV，及原始音频数据。
- **Audio Tone Utility** – 一个接口，它所提供的一些方法用于播放和配置单序列音调 and 双音多频（DTMF）串。播放功能是由 **CmdaAudioToneUtility 类[5]** 提供的，这个类可用于播放：
 - 特定长度和频率的单音（需要终端支持）
 - 双音（需要终端支持）
 - 双音多频串
 - 储存在文件或描述符中的音调序列（不同开发商可能支持不同的格式/类型）。
 - 移动设备中预定义（固定）的音调序列（不同开发商是否支持各有不同。诺基亚终端目前并不支持）。

Midi Client Utility – 提供一个接口，用于打开、播放，及获取MIDI格式的信息。既可以用文件形式也可以用描述符形式来提供MIDI数据。**CMIDIClientUtility类[6]**提供了Midi播放器功能。这个API也支持实况midi事件的播放，且为midi的一些回放属性（如节拍等）提供了更强的控制。

3.2 选择适当的 API

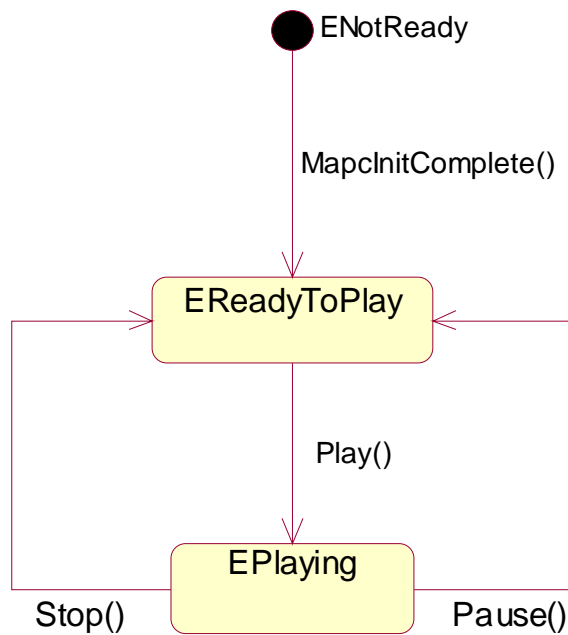
如上所述，**Audio Player Utility**是通用API，但它要求使用这个API的应用能提供DRM capability，以播放受DRM保护的内容。使用**Audio Player API**的应用必须使用**DRMHelper**类[7]实现对DRM内容的出错处理。也可以使用**CDRMLicenseChecker [8]**类来实现数据检查和解密。**DRMCommon [9]**提供了大部分常用的DRM函数。这个类提供了广泛的功能，如读取文件中的过期信息或获取文件具体的播放、显示、打印，及执行权限。

CDrmPlayerUtility [4] 提供了与**CmdaAudioPlayerUtility [3]** 相同的接口，用于打开、播放采样的音频数据，或从中读取信息。可以向**CDrmPlayerUtility [4]** 提供音频数据，方法与**CmdaAudioPlayerUtility [3]** 完全相同。两者的主要区别是**CDrmPlayerUtility [4]** 允许不具备DRM capability的应用播放受DRM保护的内容。使用**CDrmPlayerUtility [4]** 播放受DRM保护内容的代价是：应用程序的性能会有少许下降。

CDrmPlayerUtility [4] 与**CmdaAudioPlayerUtility [3]** 提供相同的接口，这意味着：只需改变导入的类，大部分代码可以保持不变，因为**CDrmPlayerUtility [4]** 中的一些同样方法可以以相同的方式被从**CmdaAudioPlayerUtility [3]** 中调用。

3.3 Play utility 文件播放的生命周期

由开发伙伴定义的用于某个文件播放应用的三大状态：



可以在音频播放器客户端应用内对这些状态做如下声明：

```

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

```
TState iState;
```

MMF 为音频回放所提供的这些 API 独立于所播放音频的格式和类型。同一 API 既可用于回放本地内容，也可用于回放流内容，并得到下列 S60 类的支持：

- **CMdaAudioPlayerUtility** –支持音频回放操作和简单元数据读取操作的 Audio Player 类。这也是实现解码器功能的类[3]。
- **MMdaAudioPlayerCallback** –针对 **CMdaAudioPlayerUtility** 的回调类，它报告错误并向应用提示文件打开操作或播放已经完成等状态。这是一个 observer 类。基本上这个类需要实现 **MapcInitComplete()** 和 **MapcPlayComplete()**，3.3.1 节将讲解这两个方法[3]。

3.3.1 初始化

对音频采样进行初始化是相当简单的操作，但对某些问题也必须深思熟虑。由于 **CMdaAudioPlayerUtility** 中的所有操作都是异步的，所以有必要让一个客户端类来侦听音频播放操作。这个类必须继承自 **MMdaAudioPlayerCallback** 接口类，它提供了两个方法，即 **MapcInitComplete()** 和 **MapcPlayComplete()**。当打开及初始化一个音频采样的工作结束之后，不管其成功与否，**MapcInitComplete()** 方法将定义所需的客户端行为。而当播放某段音频采样操作完成后，不管其成功与否，**MapcPlayComplete()** 也将定义所需的客户端行为。

当调用 **MapcInitComplete()** [3] 时，把对象的状态从 **ENotReady** 改为 **EReadyToPlay** 是一个好习惯。这样做可以通知应用：初始化操作已成功完成，可以播放音频文件了。如果在初始化过程中出错，也可以采取一些必要的措施。

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

```



注意：如客户端应用需要，可以在 GUI 上显示音频片段的持续时间。

另一个好习惯是：使用 `MapcPlayComplete ()` [3] 通知应用，播放操作已经完成，客户端可以接收新的请求。回放音频文件时如遇出错可以通过这个函数来实施必要的措施。

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

```

一旦从 `MMdaAudioPlayerCallback` [3] 实现了这两个回调函数，就可以对 `CMdaAudioPlayerUtility` 对象进行初始化。初始化过程首先要做的就是：创建 `CMdaAudioPlayerUtility` 类的一个实例。`CMdaAudioPlayerUtility` 类为初始化该对象提供了若干选项。初始化有两种情形，即所播放的音频文件既可以指定，也可以不指定。

正如上面所提到的，在第一种情形中，可以对播放器对象进行实例化，同时可以加载（打开）音频文件供播放，两个操作一步完成。在这种情况下，`CMdaAudioPlayerUtility` 提供了下列函数：

- `NewFilePlayerL ()` [3] 函数构造并初始化音频播放器用例的一个新实例，用于播放文件（`aFileName`）中的采样音频数据。下面的代码片段说明了如何创建这个实例：

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

```

- `NewDesPlayerL ()` [3] 函数构造并初始化音频播放器用例的一个新实例，用于播放描述符（`aDescriptor`）中的采样音频数据。下面的代码片段说明了如何创建这个实例：

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

```

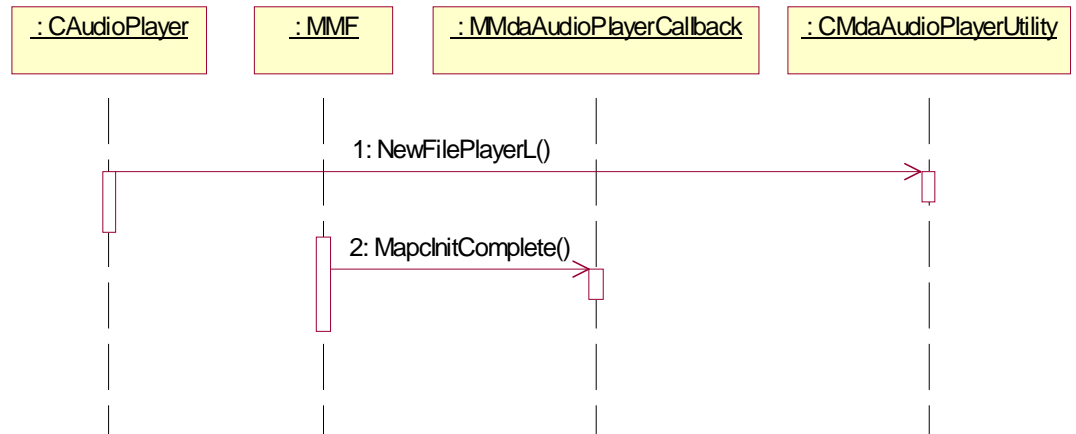
- 最后，`NewDesPlayerReadOnlyL ()` [3] 函数构造并初始化音频播放器用例的一个新实例，用于播放只读描述符（`aDescriptor`）中的被采样音频数据。下面的代码片段说明了如何创建这个实例：

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

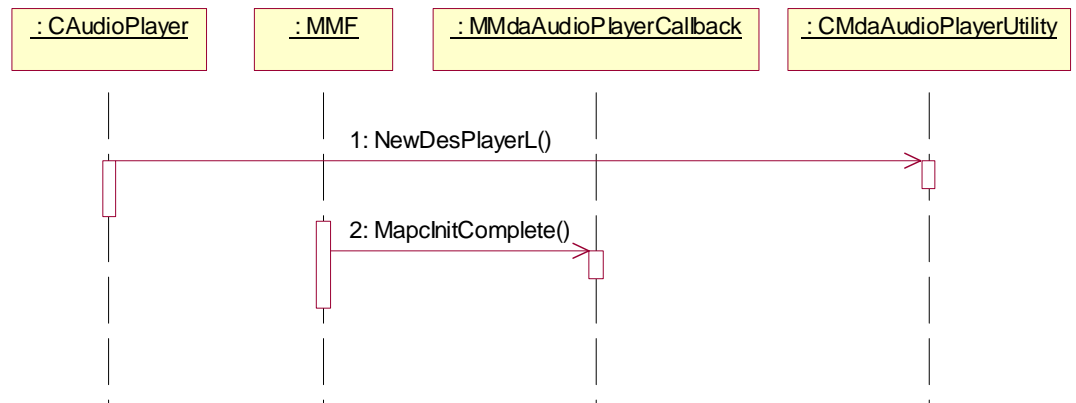
```

所有这些函数都必须得到获支持格式的音频数据，且当不能创建该对象时抛出异常。

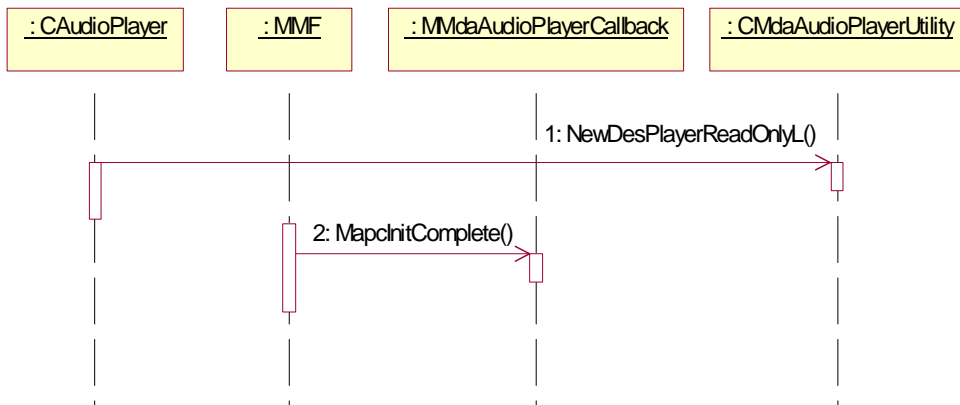
下面的时序图显示了用 `NewFilePlayerL ()` 方法进行初始化的过程：



下面的时序图显示了用 `NewDesPlayerL ()` 函数进行初始化的过程：



下面的时序图显示了用 `NewDesPlayerReadOnlyL ()` 函数进行初始化的过程：



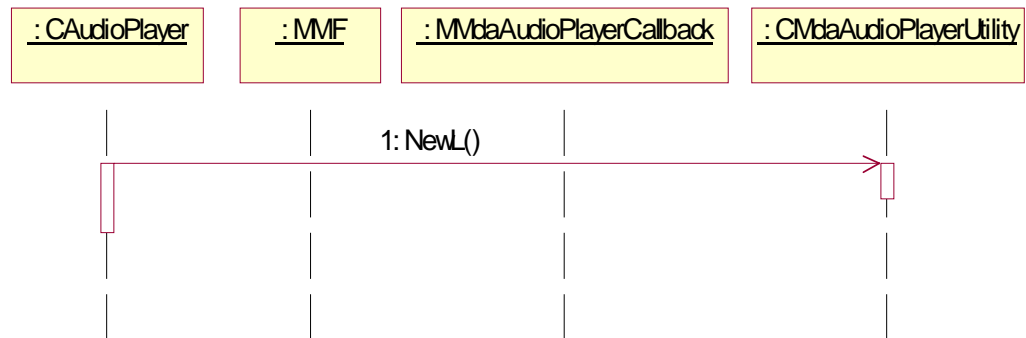
很明显，对第一种初始化情形来说，三种初始化过程各自非常相似。您可能还注意到，在音频播放器对象被初始化之后，调用了 `MapcInitComplete ()` 方法；这是因为：函数初始化音频播放器对象的同时也打开一个音频文件。

在第二种情形中，如果您仅仅希望实例化播放器对象，而不指定具体的音频片段，就必须使用以下的 `CMdaAudioPlayerUtility` 成员函数：

- **NewL()** 函数构造并初始化音频播放器用例的一个新实例，并当创建失败时抛出异常。下列代码片段表示如何创建一个实例：

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

下面的时序图显示了用 **NewL()** 函数的初始化进程。请注意当使用这个函数初始化该对象后，并没有出现回调。



对于上述各个函数，当创建 **CMdaAudioPlayerUtility** 实例时，可以为这个实例指定优先权及优先级。这些参数都是可选的，其默认值是 **EPriorityNormal** 和 **RMdaPriorityPreferenceTimeAndQuality**。

用于对实例进行初始化的所有方法都是异步的。因此在处理该操作时应用本身并不会被阻塞。

再来看第二种情形，不管何时，当您需要播放一个音频文件时，您一旦用 **NewL()** 函数创建了该实例，就需要打开该文件以便可以回放该文件。请记住，在第一种情形中并不需要该步骤，因为可以在同一个初始化函数内打开那个音频文件。可以通过向其传递文件的完整路径信息或该文件的描述符来打开一个文件，请使用下述方法之一：

- **OpenFileL()** [3] 函数从某个文件 (**aFileName**) 中打开一个音频片段。音频数据必须具备某种获支持的格式。下面的代码片段说明了该函数的一个使用范例：

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

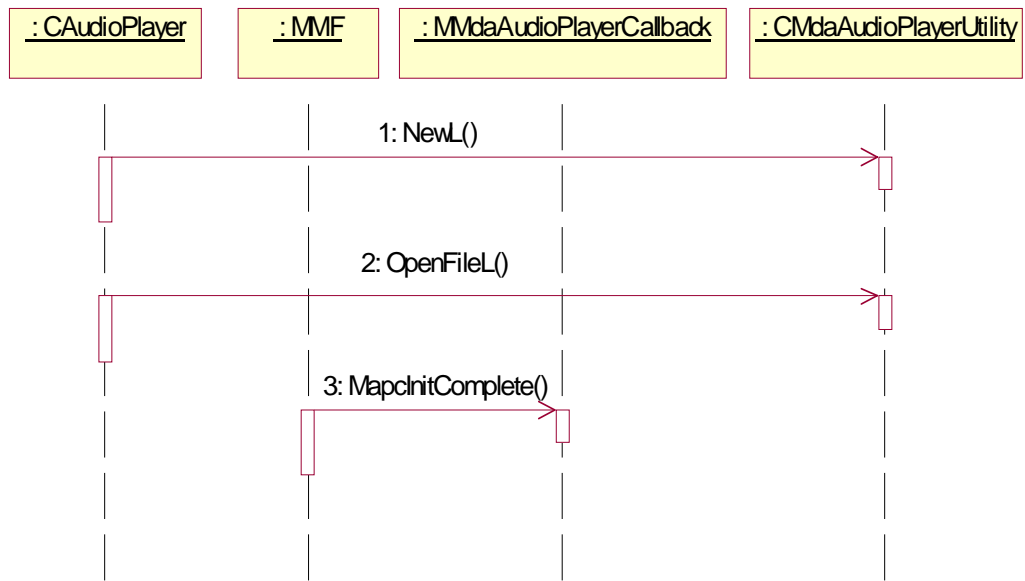
- 打开某个音频片段的第二种方法是使用 **OpenDesL()** [3] 函数，它从一个描述符 (**aDescriptor**) 中打开这个音频片段。音频文件必须具备某种获支持的格式。下面的代码片段说明了该函数的一个使用范例：

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

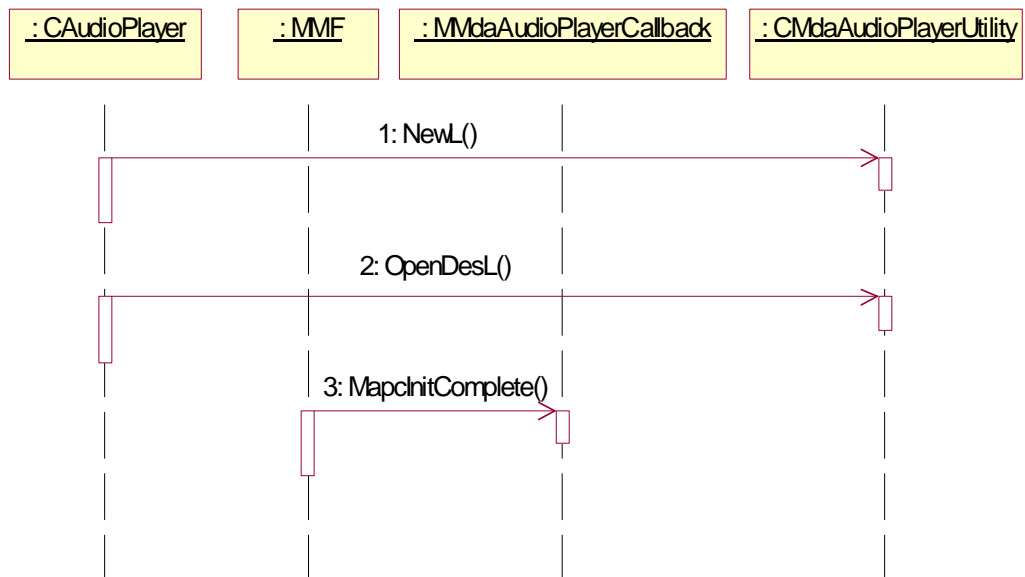
针对这两个函数的音频文件必须具备获支持的格式。如果之前有未完成的打开操作，这些方法将抛出 **KerrInUse** 异常。请注意，**MP3** 描述符需要在数据开头处包含 **mime** 类型信息。

下面两幅时序图说明了当使用 **NewL()** 函数创建播放器的一个实例时的初始化过程，及使用 **OpenFileL()** 函数打开音频文件或通过 **OpenDesL()** 函数打开音频文件描述符时的初始化过程。

当使用 **OpenFileL()** 时：



当使用 `OpenDesL()` 时:



如您所见，每次打开文件后，都会调用 `MapchritComplete()` [3] 方法，得到相关出错信息，如 `KerrNone`, `KerrNotSupported`, 或 `KErrNoMemory` [10]，告知应用

打开文件操作的状态。应用确定这些状态后播放器才能播放。如果在文件打开操作中并没有出错 (`KerrNone`)，就可以启动回放操作。这符合第一种情形 (`OpenFileL`) 中的任何初始化函数，因为音频文件是在这些函数内部被打开的；同时也符合音频文件被打开之后的第二种情形 (`OpenDesL`)。另外的一些状态信息与具体的 EPOC 平台有关。

为了处理可能的出错，您只需判断 `MapcInitComplete()` [3] 返回的出错信息是否等于 `KerrNone`，然后根据该出错信息继续处理。

```
void CAudioPlayer::MapcInitComplete(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;
    }
}
```

如前所述，`NewFilePlayerL()` 和 `OpenFileL()` 函数要求把音频文件作为参数来启动播放。下面的代码片段演示了为这些函数获取音频文件的范例。在这个范例中，在音频客户端应用中实现了一个名为 `GetFile()` 的函数。用于该函数的参数是含有要播放的音频文件名的描述符，它将返回该音频文件及其路径。

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

下面的范例演示了如何使用 `GetFile()` 函数：

```
//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 ));
```

请记住，对于 `OpenFileL()` 函数，打开该文件前需要用 `NewL()` 方法先执行初始化。

3.3.2 播放

完成初始化且打开音频文件后，您就可以播放这个音频文件了。如下面的代码所示，文件回放是通过调用 `CMdaAudioPlayerUtility::Play()` [3] 实现的，这个方法将从音频文件的当前位置开始回放音频数据。

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

在上面的代码片段中，您注意到：首先要做的事情是去验证 `iState` 变量。请记住，每当打开音频文件后，在 `MapcInitComplete()` 回调中要把 `iState` 变量设为 `EReadyToPlay`，并在开始播放文件后立即把 `iState` 变量设为 `EPlaying`。

回放过程一旦顺利完成或被 `Audio Policy` 叫停，`MapcPlayComplete()` [3] 方法将会得到 `KerrNone`（表示操作完全成功）或某个可能的出错代码。`KErrCorrupt` 告诉您采样数据已损坏。`KErrInUse` 说明音频设备正在被具有更高优先级的客户使用。如果当前的音频任务正被具有较高优先级的其他音频任务所抢占时，也会在回放期间返回这个出错。`KErrNoMemory` 表示没有足够的内存来播放这段音频采样。还有一些其他的值，表示打开音频采样时遇到的问题。这些值随设备不同而不同。正如初始化时那样，某个唯一标识码有助于您处理回放期间的各种出错。

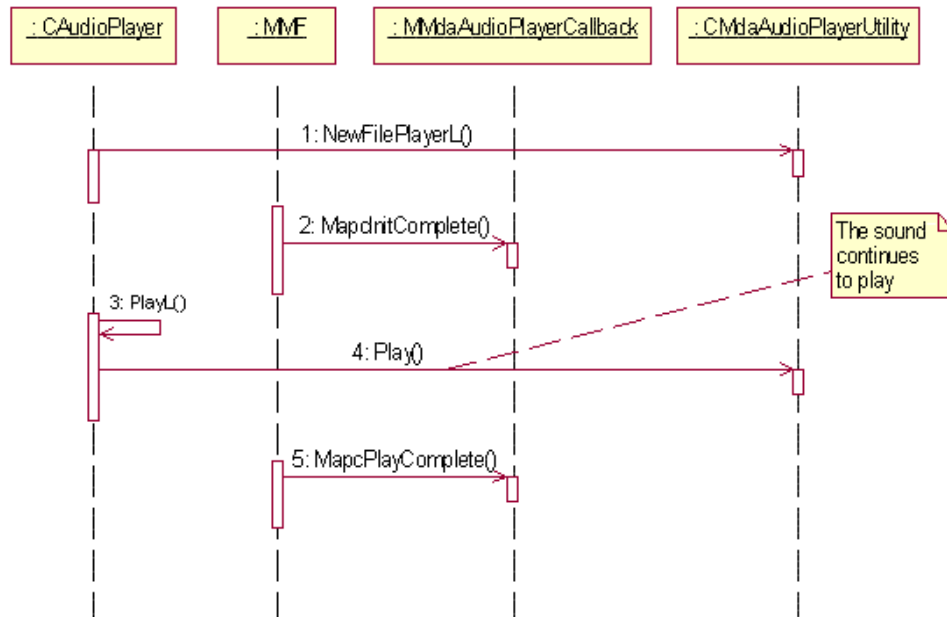
```
void CAudioPlayer::MapcPlayComplete(TInt aError)
{
    iState = aError ? ENotReady : EReadyToPlay;

    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;
    }
}
```

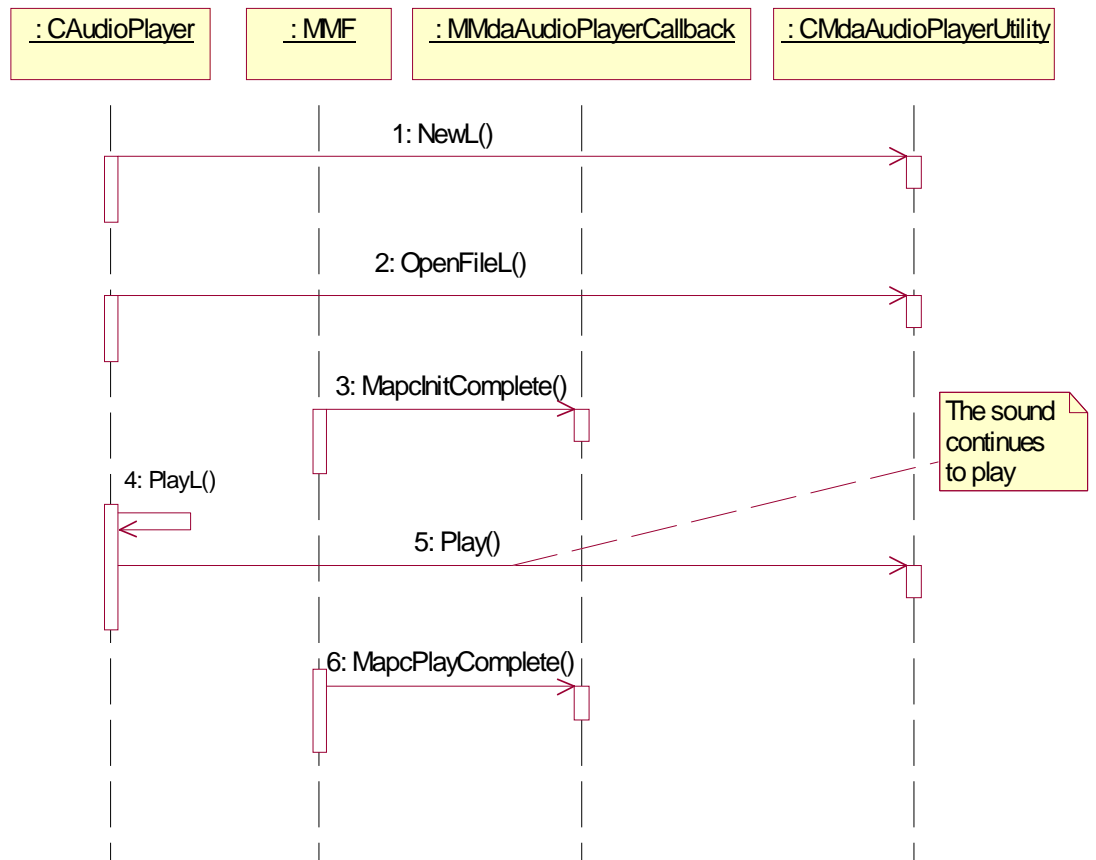
各种播放方法都是异步的。这意味着：这些操作完成时应用并不会被阻塞。对于各种 `CMdaAudioPlayerUtility` 类，会在两种情况下完成播放操作：该用例(`utility`)已经播完了特定文件或描述符中的整段被采样音频，或在播放中途出错。

当完成了某个异步操作时，音频用例对象会调用侦听器对象上的某个回调函数，以便向该应用对象或任何其他侦听器对象发出一个事件消息。

请记住：音频文件播放之前需要先完成初始化过程并打开音频文件。下面的时序图展示了第一种情形，即初始化和打开操作一步完成。这个范例使用了 `NewFilePlayerL()` 函数，但也可以使用我们之前讨论过的那些函数（请参阅 3.3.1，初始化）



第二种情形（请参见 3.3.1，初始化），即初始化和打开操作分两步完成，反映在下面的时序图中：



在这个范例中，文件是用 `OpenFileL()` 方法打开的，但也可以用 `OpenDesL()` 函数打开。（请参阅 3.3.1，初始化）

完成所有播放及/或不再使用已加载音频文件时，应该执行 `CMdaAudioPlayerUtility::Close()` [3] 清除操作。这个方法关闭当前的音频片段，允许打开别的片段，因而 `iState` 变量必须被设定为 `ENotReady`，因为该音频文件已不可再回放。在这个范例客户端应用中，这个过程由 `Close()` 函数完成，如下面的代码片段所示，可以将它放置到客户端应用所需要的任何地方。

```

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

```

3.3.3 暂停

只要使用 `CMdaAudioPlayerUtility::Pause()` [3] 方法就可以暂停音频文件的回放；该音频文件的当前播放位置会被记录下来，以便调用 `PlayL()` 继续播放。

```

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

```

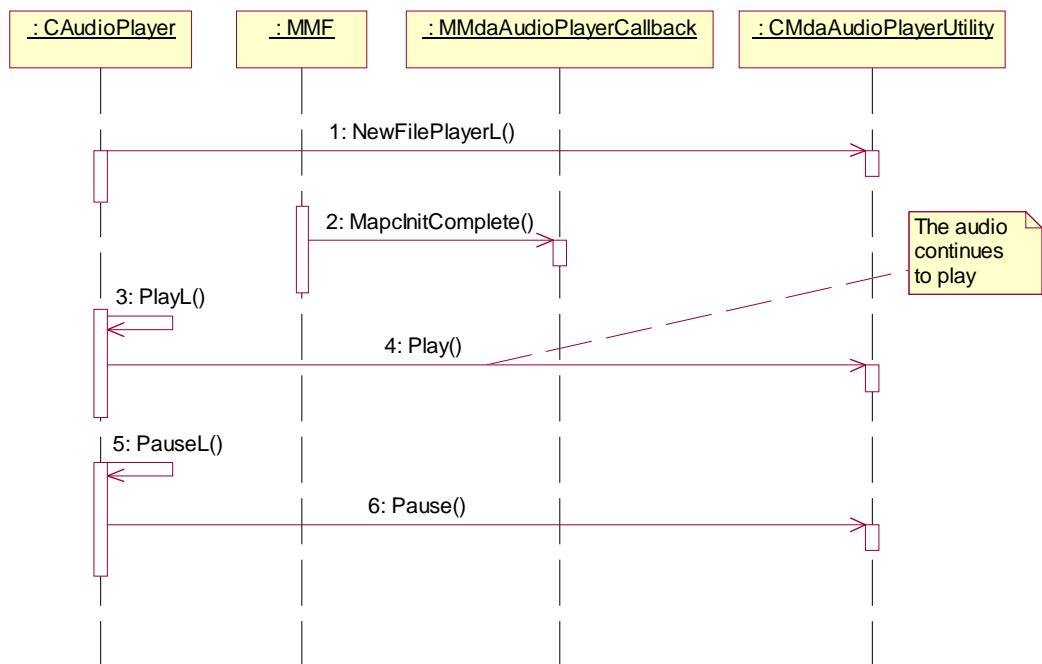
```

    iState = EReadyToPlay;
}

```

与之前的几个操作一样，暂停期间可能会出错。`Pause()` 函数会返回一个 `Tint` 值表示任何可能的出错，供您处理。此外，如代码片段所示，当没有任何出错时，`iState` 变量才被设定为 `EReadyToPlay`，这么做的原因是：暂停后音频文件仍然处于可回放状态。

下面的时序图展示了这个暂停操作。请注意，还可以使用在 3.3.1 初始化章节中讨论过的任意一种方式来完成初始化进程。



3.3.4 停止

停止回放音频文件很简单，只需调用 `CMdaAudioPlayerUtility::Stop()` [3] 方法，如下一段代码片段所示。这将尽可能快地停止音频采样。成功完成操作后，播放位置被移到该音频片段的起始处。

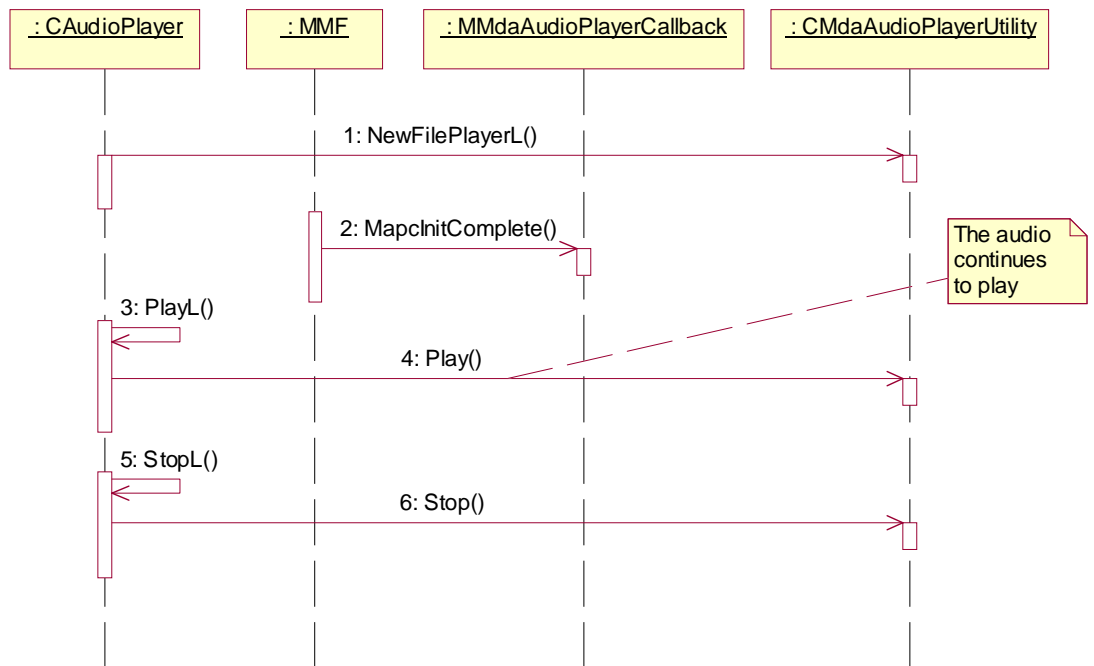
```

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

```

完成回放后再调用该方法就不会有任何效果；在这种情况下，也不会去调用 `MapcPlayComplete()` 回调方法。重要的是：在尚未初始化音频播放器用例之前不调用该方法，否则会造成 `CMdaAudioPlayerUtility` 的异常出错。为确保安全，一种做法是把停止功能设为“失效”，直到初始化进程开始。

将各种播放操作定为异步，应用对象可通过调用合适的函数在操作结束前停止其操作。下面的 UML 时序图显示了这一过程。请记住，时序图中的初始化进程可以被改为之前讨论过的任何一种方法（请参阅 3.3.1，初始化）。



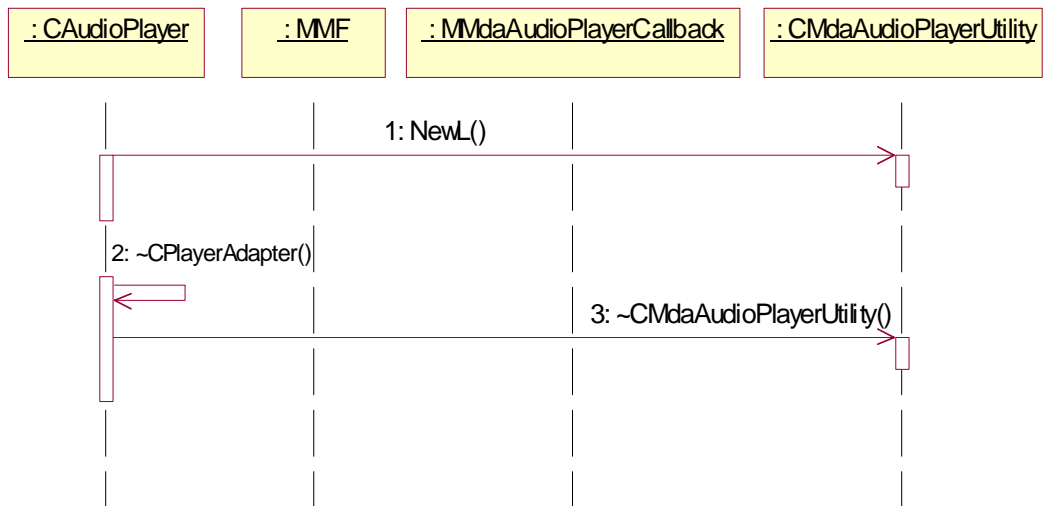
3.3.5 析构

析构过程非常简单。您要确保在析构函数中销毁 `CMdaPlayerUtility` 对象。

```

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

下面的时序图展示了这个过程：



在这个范例中，`NewL()` 函数用于初始化该对象，这里仅仅演示了用其销毁该对象。然而，也可以用之前讨论过的任何一种有效方法来初始化该对象（请参阅 3.3.1，初始化）。

3.4 搜寻

不同于已讨论过的其它音频操作，搜寻操作并非由一个单一的函数（如 `Play()`，`Stop()`，或 `Pause()`）组成。相反，快进和后退过程都是三步操作。

首先，必须知道该文件的实际播放位置；可以用 `CMdaAudioPlayerUtility::GetPosition()` 方法完成。

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

获得位置后，下一步就是对这个位置进行增/减以实现后退或快进等动作。增加 `aPosition` 的值将导致快进，而减少其值则导致后退。在这种情况下，步进时间为一秒，因为音频数据中的位置以微妙计。搜寻操作的速度根据所使用的编码解码器而有所不同。某些编码解码器拥有更快的速度。

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

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

一旦修改了位置值，您必须告诉 `CMdaAudioPlayerUtility` 对象将当前播放位置设为新值。

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

下面的代码片段给出了实现搜寻功能的一个范例：

```
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;

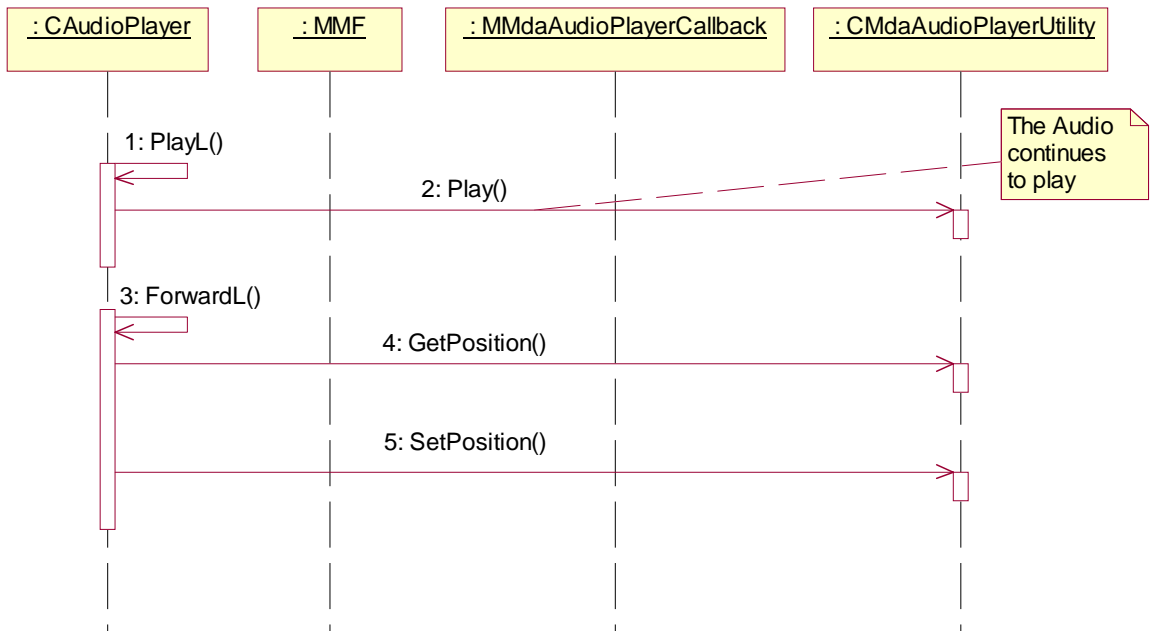
    aError = iMdaAudioPlayerUtility->GetPosition( aPosition );
    if(aError != KErrNone)
    {
        ... Actions required for handling a general error ...
    }

    aPosition = aPosition.Int64() - 10000000;

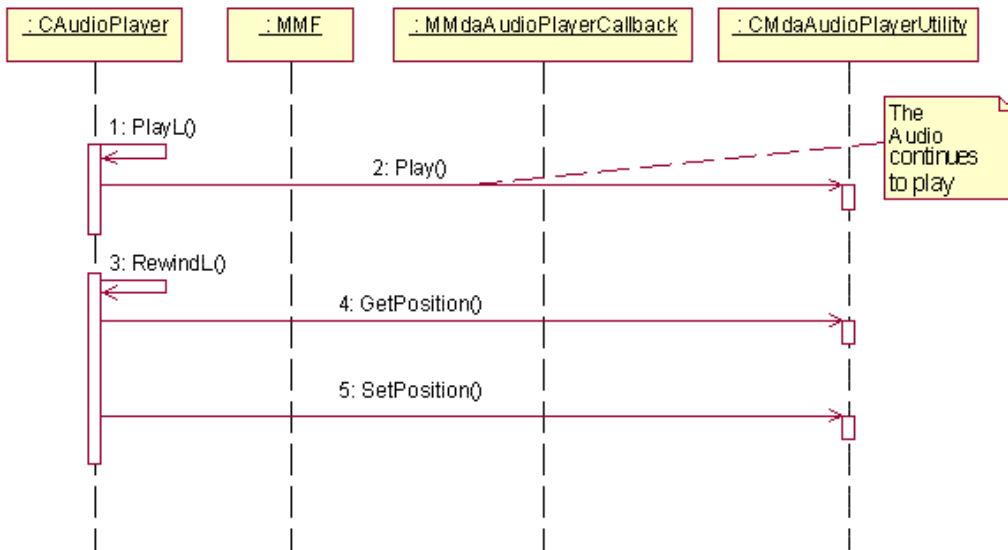
    aError = iMdaAudioPlayerUtility->SetPosition( aPosition );
    if(aError != KErrNone)
    {
        ... Actions required for handling a general error ...
    }
}
```

`GetPosition()` 和 `SetPosition()` 这两个函数都返回一个 `TInt` 变量，提示其处理过程中可能出现的错误。这能用于应用所需要的出错处理。

以下的时序图展示了快进功能：



以下的时序图展示了后退功能：



3.5 播放系列文件

下节提供了播放系列文件若干情况的范例。

3.5.1 前一个文件播完后播放一个新文件

因为您已经熟悉如何播放一个文件，本节内容就十分简单了。当触发了 `MapcPlayComplete()` [3] 回调方法后，即表示回放过程已完成，这时通过调用 `Close()` 函数关闭该音频文件。这个方法关闭当前的音频片段，并允许打开另一个片段。

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

一旦关闭了音频文件，下一步就是打开您希望播放的文件。3.3.1 “初始化”一节所讲述的任何方法都可用于打开一个文件。

Option 1 选项 1

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

Option 2 选项 2

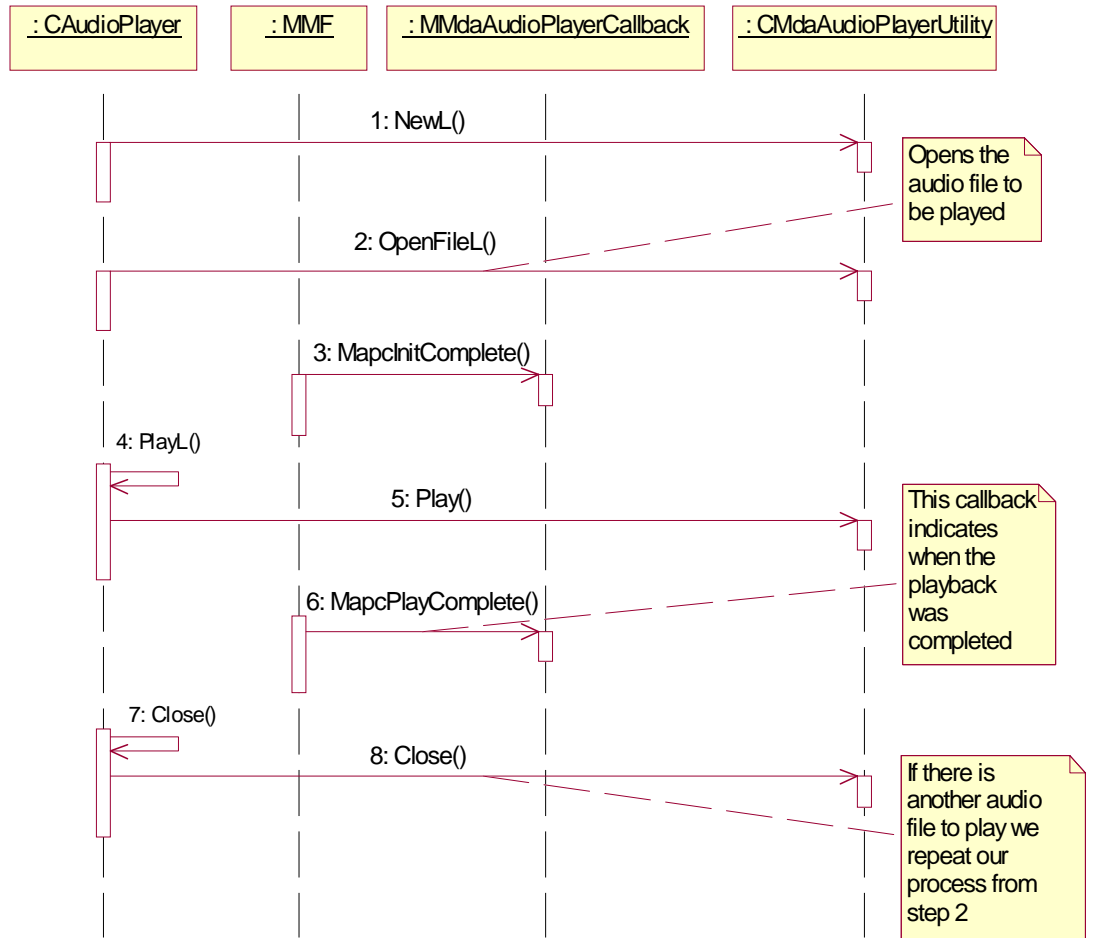
```
iMdaAudioPlayerUtility ->OpenDesL( aDescriptor );
```

打开新文件后，您就能调用 `PlayL()` 方法来播放这个文件。

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

关于出错处理，3.3 节“Play utility 文件播放的生命周期”中所讲述的那些方法同样也适用本节，因为初始化、播放等都使用了一些相同的函数。切记：检查一下用到了哪些函数及针对这些函数的出错处理等。

下面的时序图展示了前一个文件播完后播放新文件时所涉及到的一些方法和事件。



3.5.2 另一个文件在播时播放一个新文件

当正在播放某个音频文件时试图播放一个新文件，请先停止正在回放的文件。

```

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

接着请关闭那个音频文件：

```

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

关闭文件后，打开新文件以供播放，如 3.3.1 “初始化” 所述。

Option 1 选项 1

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

Option 2 选项 2

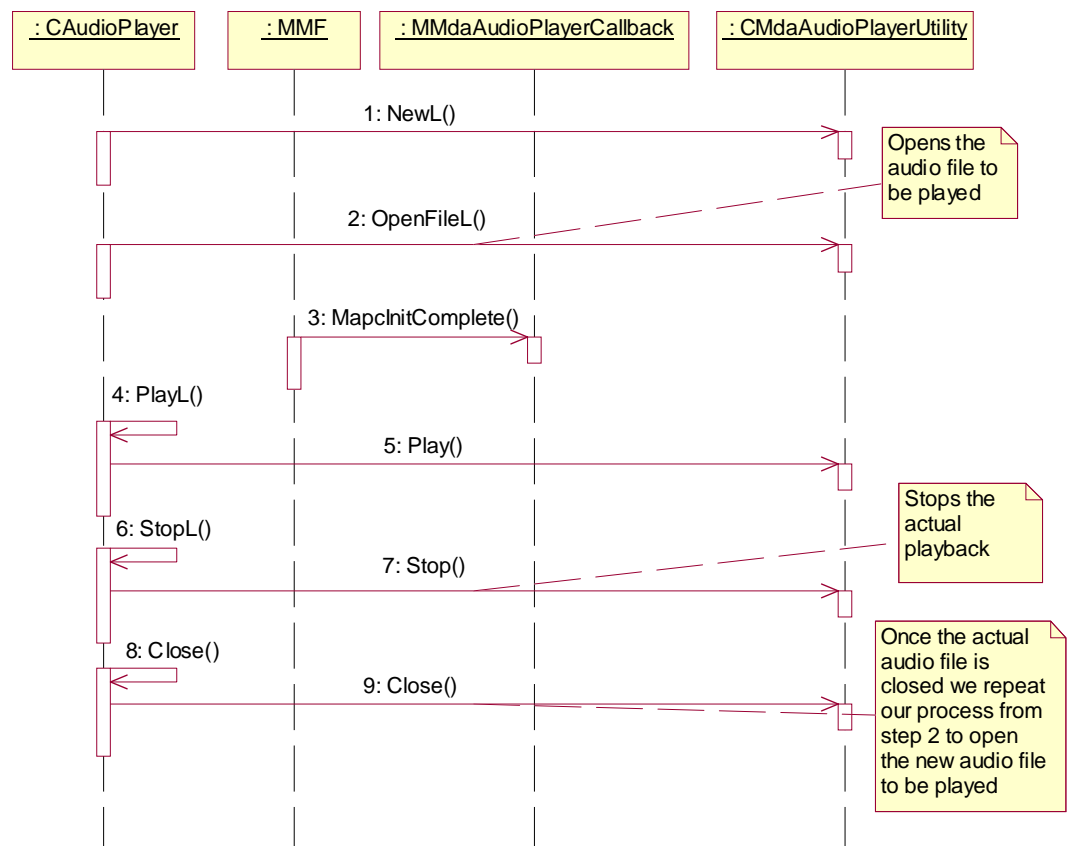
```
iMdaAudioPlayerUtility ->OpenDesL( aDescriptor );
```

最后调用 `PlayL()` 函数播放新音频文件。

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

在3.3节“Play utility 文件播放的周期性”中所讲述的对于出错事例的一些处理方法同样也适用本节，因为初始化，播放等都用到了一些相同的函数。请到3.3节“Play utility 文件播放的周期性”中查阅所使用的函数及针对这些函数的出错处理。

下面的时序图展示了这种情况：



3.5.3 暂停播放前一个文件并开始播放新文件

可以暂停播放前一个文件并开始播放一个新文件。当音频片段正处在回放过程中时调用 `PauseL()` 函数。

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

```

    }
    iState = EReadyToPlay;
}

```

然后调用 `Close()` 函数关闭那个音频文件，并把 `iState` 变量设为 `ENotReady`。这个方法关闭当前的音频片段，同时允许打开另一个片段。

```

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

```

通过修改 `iState` 变量，您告诉应用：您将播放的新音频文件还处于不可播放状态，这是因为：如果关闭了音频文件，就不能再回放它了。在应用中声明的这个变量用于处理 3.3 “Play utility 文件播放的生命周期” 一节中所讲述的那些可能状态。

现在应用可以加载一个一个新音频文件并播放了。下一步是打开您想播放的那个新文件。请使用 3.3.1, “初始化” 一节中讲述过的那些方法。

Option 1 选项 1

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

Option 2 选项 2

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

打开新文件后您只要调用 `PlayL()` 方法就可以播放了。

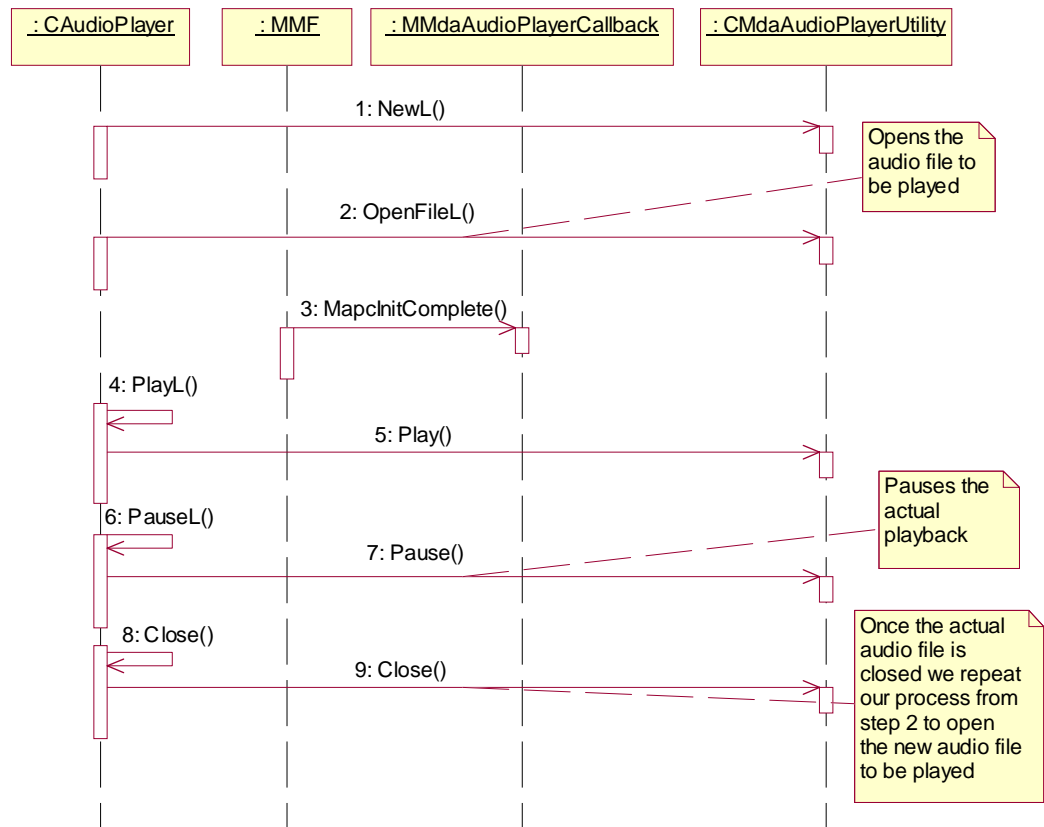
```

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

```

这种情形与我们曾经讲述过的出错处理情形相同，因为在 3.3 “Play utility 文件播放的生命周期” 一节中有关出错处理的方法同样适用于这一节中所讨论的情形，我们使用相同的一些函数来进行初始化、播放、暂停等。您可以查一下用到了哪些函数，以及 3.3 “Play utility 文件播放的生命周期” 一节中所讲述的针对那些函数的出错处理方式。

下一个时序图展示了暂停播放前一个文件并开始播放一个新文件所涉及的一些函数和事件。



3.5.4 停止播放前一个文件并开始播放新文件

这个过程类似于暂停播放前一个文件然后开始播放一个新文件，所不同的只是：并非调用 `PauseL()` 函数，而是调用 `StopL()` 函数来停止正处与播放中的音频文件。

```

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

停止播放前一个音频文件后，您还需要关闭它。

```

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

下一步是打开新音频文件并播放。请使用在 3.3.1 “初始化” 一节中讲述过的任何一种方法来打开一个音频文件

Option 1 选项 1

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

Option 2 选项 2

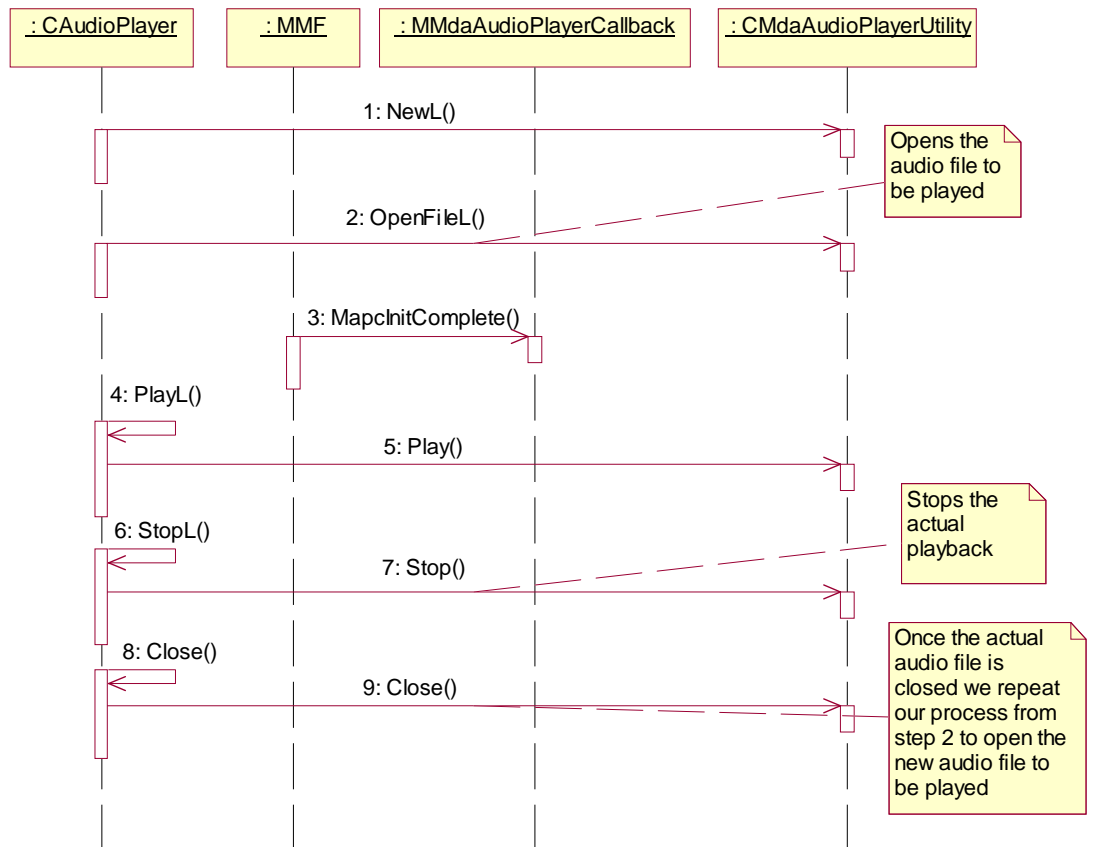
```
iMdaAudioPlayerUtility ->OpenDesL( aDescriptor );
```

最后，打开新文件后，您就可以通过调用 `PlayL()` 方法开始播放。

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

3.3 “Play utility 文件播放的生命周期”一节中所讲述过的出错处理方式同样适用于本节，因为您使用相同的一些函数进行初始化、播放、停止等。您可以查一下用到了哪些函数，以及 3.3 “Play utility 文件播放的生命周期”一节中所讲述的针对那些函数的出错处理方式。

下一个时序图展示了相关的一些函数：



4. 读取元数据 (metadata)

随着音频文件一些新格式的出现，一些新功能也被添加到这些文件中。其中一项功能就是支持嵌入在文件中的元数据。**Metadata** 信息是内嵌在某些音频文件中的音轨信息，通常包括歌曲标题，歌手或艺术家的名字，及音乐流派信息等。从 **Symbian v7.0** 版本开始，**Audio Play Utility [3]**支持从文件中提取元数据信息。**Audio Play Utility [3]**和 **DRM Audio Play Utility [4]**以非常类似的方法处理元数据的提取操作：它们都要求加载一个文件，并且都需要确保播放该文件所必须的安全资源。

用 **Play Utility** 从某个文件读取元数据信息的过程直截了当。第一步是初始化 **Play Object** 并向其加载一个音频文件，使用的方法与本文有关初始化章节中所讲述过的完全相同，请参见 3.3 “**Play utility** 文件播放的生命周期”。

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

如前所述，可以从某个 **DRM** 文件中提取元数据。在这种情况下您可以使用 **DRM Audio Play Utility**。这个 **API** 所提供的全部函数其处理方式都与 **Audio Play Utility API** 所提供的相同。唯一的不同在于初始化过程中，您并不去创建 **Audio Play Utility** 的一个实例，而是去创建 **DRM Audio Play Utility** 的一个实例。

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

此后 **DRM Audio Play Utility** 与 **Audio Play Utility API** 用相同的方法处理全部函数。实例创建之后，下一步就是打开该音频文件。

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

一旦对那个对象进行了初始化并把音频数据正确加载到这个对象后，您必须考虑该文件中包含多少项元数据。

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

现在您开始提取信息。提取出每一项元数据，并将其与一组预先设定的条目名进行匹配。下面列出了 **ID3 tag [11]**最常见的一些条目名称。

条目名	说明
KMMFMetaEntrySongTitle	Title of song. 歌曲标题
KMMFMetaEntryAlbum	Name of album. 专辑名称
KMMFMetaEntryArtist	Name of singer or artist. 歌手或艺术家名字
KMMFMetaEntryAlbumTrack	Track number of the track in the album. 专辑的曲目数量
KMMFMetaEntryYear	Year album was published. 专辑发行年份
KMMFMetaEntryGenre	Genre of music. 音乐流派

条目名	说明
KMMFMetaEntryCopyright	Track copyright information. 乐曲版权信息
KMMFMetaEntryComment	Track comments. 乐曲标注
KMMFMetaEntryComposer	Composer information of the track. 乐曲作者信息
KMMFMetaOriginalArtist	Original artist information. 原创艺术家信息

为确定所提取的值属于哪个条目名，以及它所代表的是什么信息，名字和值必须提取自各个条目。下面这段代码遍历了文件中所包含的每个元数据条目。我们记得在前面的步骤中获得过 `aNumEntries`，这个变量表示了该文件中条目的数量。在这段代码中，`Play Utility` 为操作元数据而提供的函数是 `iMdaAudioPlayerUtility->GetMetaDataEntryL(index)` [3]，它返回一个 `CMMFMetaDataEntry` [12] 对象，其中含有标记名及其值。请注意，在这段代码片段中，`aMetadataName` 代表之前表格中的任何一个 `EntryName` 项目。通过读取条目名您将准确地了解到您所获取的是哪个元数据。

```

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 );
        }
    }
}

```

与大多数函数类似，`GetMetaDataEntryL()` 和 `GetNumberOfMetaDataEntries()` 提供了一种方法，用于判断操作是否成功。其中的 `GetNumberOfMetaDataEntries()` 要么给出一个 `KerrNone`，要么给出一个系统级出错代码。`GetMetaDataEntryL()` 函数则给出更为具体的出错代码。它能用 `KerrNotFound` 抛出异常，表示所选择的元数据条目并不存在；它也能用 `KerrNotImplemented` 抛出异常，如果控制器并不支持这种格式的元数据信息。这个函数也能以其它系统级出错代码抛出异常。

```

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

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

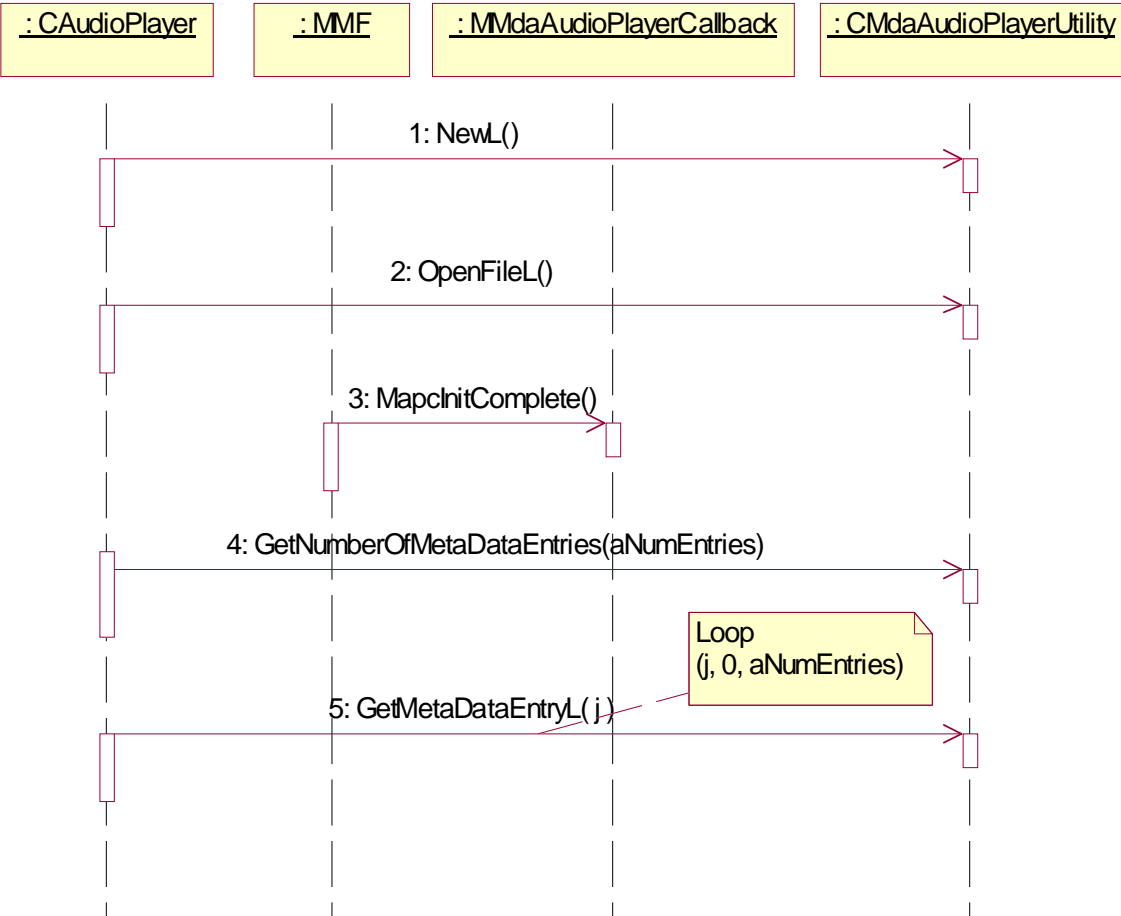
aError = GetNumberOfMetaDataEntries() ->
GetMetaDataEntryL() (aPosition);

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 ...
}

```

下面的时序图展示了用 Audio Player Utility API 读取元数据的过程。



5. 从应用程序实现流式缓冲

S60 平台允许直接从应用程序中实现流式缓冲。必须注意的是，这种类型的流式缓冲不同于互联网上对流的广泛定义。这种机制是一种手段，实现从一个软件部件向另一个软件部件的流式缓冲。与本地回放相比，这种功能拥有一定的优势。首先，流式音频并不会被永久地保存在本地，所以它不能被复制。第二，最终用户并不能向其他用户转发或直接发送该内容。最后，能更方便地保持内容更新。所有这些都为服务供应商们创造了更大的优势。

流媒体的传输带宽取决于所使用的接入技术和运营商网络的设置。当某个文件被以高码流（通常是 192 Kbps 或更高）编码时，您必须注意。这些文件也许不能很好地得到流处理，结果是回放效果不佳，原因就是可用的网络带宽没有高到能足以支持这类文件。

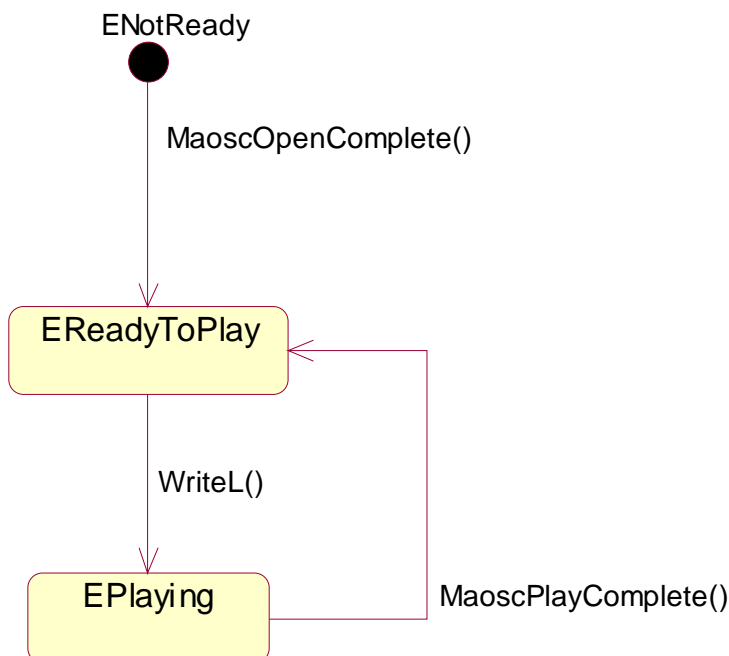
S60 终端能支持若干种常见的音频格式用于流式缓冲。有关终端所支持的音频流格式方面的更详细信息请参阅各款终端的 [Audio Capabiltiy \[13\]](#)。

对于流式缓冲用于音频播放，请使用 [CmndaAudioOutputStream \[14\]](#) API。播放本地内容和从应用程序实现流式缓冲的唯一不同是：使用了不同的打开文件和初始化方法。

5.1 节讲述了 [CmndaAudioOutputStream](#) API，它提供流式音频播放器一个接口将音频数据从特定缓冲区传递给音频硬件。

5.1 CmndaAudioOutputStream 的生命周期

在客户端应用需要处理的三种状态是：



这三种状态如下声明：

```

enum TStreamAppStatus
{
    ENotReady,

```

```

    EReadyToPlay,
    EPlaying
};

TStreamAppStatus iStatus;

```

5.1.1 初始化

在深入探讨初始化过程之前，您需要编写一个实现了 **MMdaAudioOutputStreamCallback** 的客户端类。这个类提供三个回调函数，向客户端提示音频输出的流式过程结果，让您能处理可能的出错。这些回调函数是 **MaoscOpenComplete()**，**MaoscBufferCopied()**，及 **MaoscPlayComplete()**，它们都必须由 **CMdaAudioOutputStream** 类[14]的使用类实现。

MaoscOpenComplete() 回调方法在下面讲解，因为它与初始化处理有关；其余的回调方法将在后续章节讲解。

初始化过程十分简单。首先要创建 **CMdaAudioOutputStream** 类的一个实例并对 **iStatus** 变量进行初始化。

下面的代码片段展示了如何创建一个实例：

```

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

iStatus = ENotReady;

```

当创建 **CMdaAudioOutputStream** 实例时，可以为该实例规定优先权和优先级，因为 **NewL()** 函数将这些作为参数接受。如果不能创建流式音频对象，该函数会抛出例外。

创建该实例后，最后一步是打开输出音频流包。请见下面的函数。

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

请注意，成员变量 **iSettings** (**TMdaAudioDataSettings**) 这时并不需要含有任何配置设定。

CMdaAudioOutputStream::Open() 完成后 **Multimedia** 框架会调用 **MaoscOpenComplete()** 回调函数，指出音频输出已经可用。该框架所给出的参数是一个出错值，它指出是否成功完成了 **Open()** 方法。如果成功，则给出 **KErrNone** 值。此处可以设置采样率和音量等[14]。该范例中使用了 **16khz** 的采样率。（在后面的范例中，某个名为 **CStreamApp** 的客户端类被用于该客户端程序；很显然，开发伙伴们可以另行选名。）

```

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);

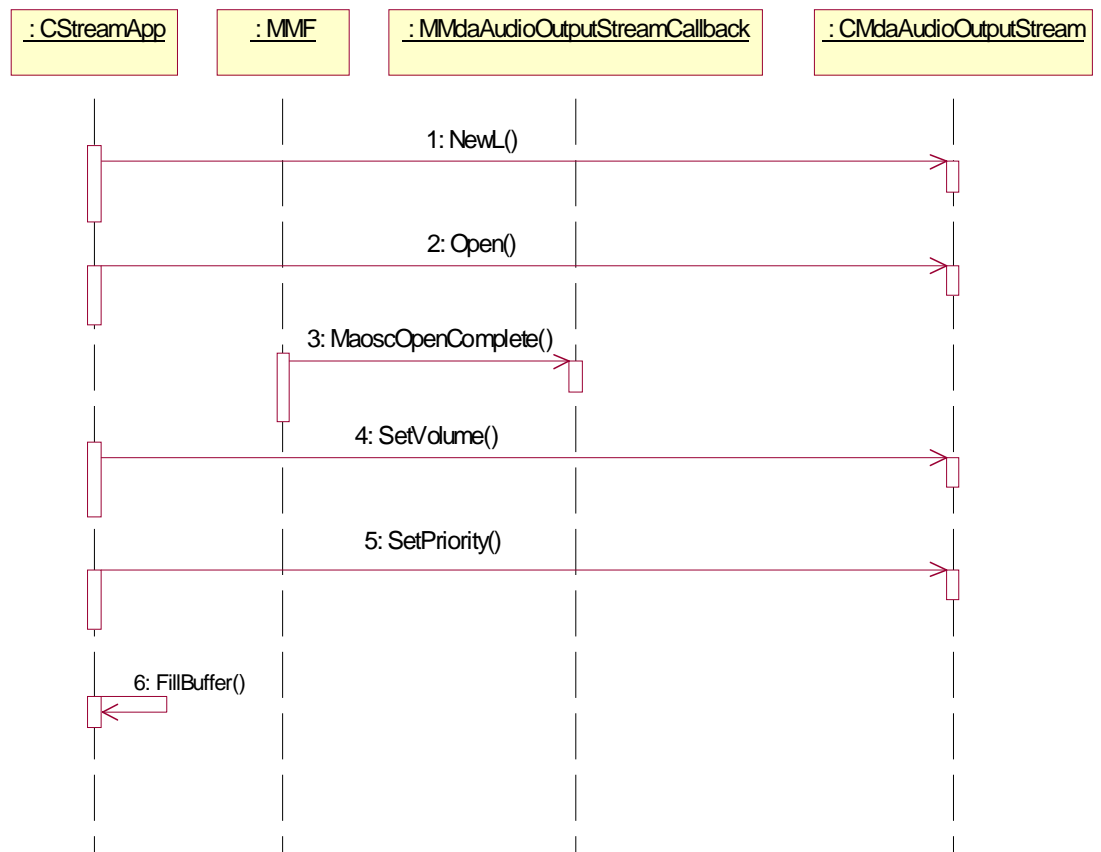
//Fill first buffer
FillBuffer();

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

```

在上面的代码片段中，`FillBuffer()` 函数是该客户端应用的一部分。应用开发伙伴将向该缓冲区填入有用的数据，由上面的代码片段中的 `FillBuffer()` 函数所完成。这必须由客户端应用的开发者实现。在打开流的同时，您需要立即调用该函数，如上面的回调所做的。上面代码片段中最后把 `iStatus` 变量设为 `EReadyToPlay` 状态就是这个原因。

上面所讲解的所有步骤都表示在以下的时序图中：



5.1.2 播放

初始化一旦完成，`CMdaAudioOutputStream` 对象就可以播放音频数据了。您可以在客户端应用中通过实现一个 `PlayL()` 函数来实现这一点。如下面的代码所示，当向 `CMdaAudioOutputStream` 中写入缓冲后，流的播放立即开始。这是通过调用 `CMdaAudioOutputStream::WriteL(const TdesC8& aData) [14]` 实现的。请

注意，`aData` 变量是对含有所要播放数据的描述符的引用，在 `FillBuffer()` 函数内部设置。

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

`WriteL()` 是一个异步函数。当复制了描述符“`aData`”中的数据到音频硬件之后，该框架将调用 `MMdaAudioOutputStreamCallback::MaoscBufferCopied()` [14] 回调方法，通知客户端：应用已收到 `aData` 并将其复制到了流中。一旦您收到了这个通知，客户端应用就可以回收该缓冲（可以重用或释放该缓冲）。在将该缓冲写入到 `CMdaAudioOutputStream` 但尚未收到 `MaoscBufferCopied` 回调和缓冲引用前，客户端不能对该缓冲的 `aData` 进行操作。请记住，您需要先向您的缓冲中加入下一个项目，然后再将其写入到流中。您可以在这个回调中完成这一操作。

```
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 . . .
    }
}
```

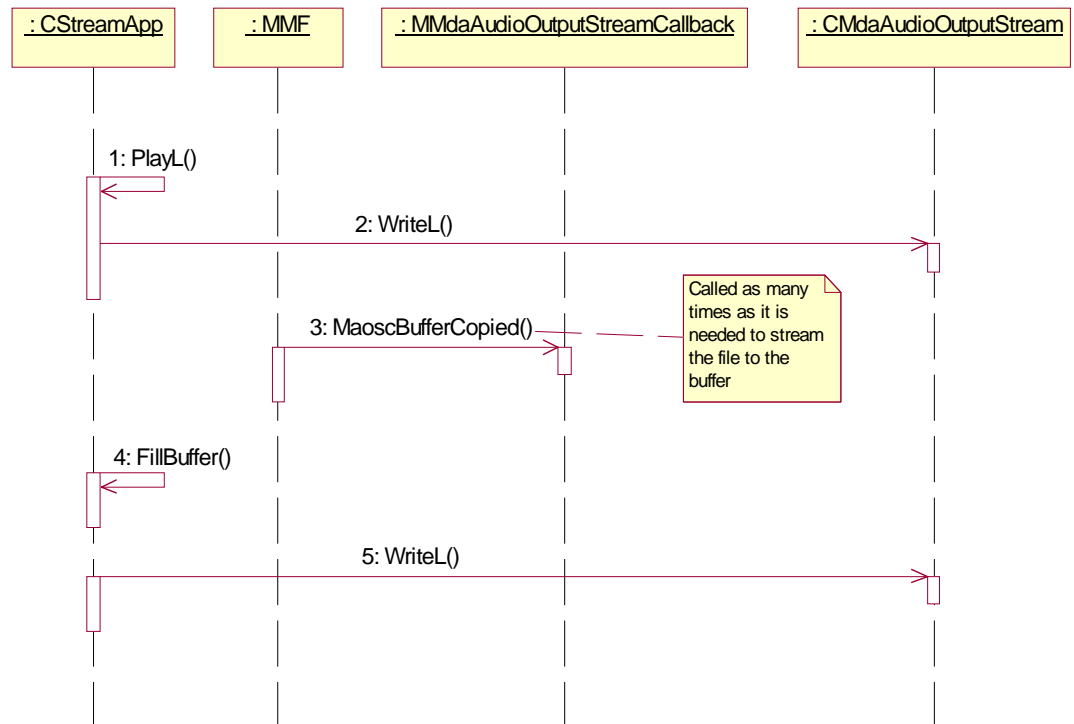


注意：回调是来自多媒体框架的通知消息。该框架会等待回调结束后才继续下一步处理。客户端应用不应在回调方法中包括会消耗 CPU 时间的代码。应尽可能快地从回调函数中返回。各种应用可以在一个独立的活动对象（**active object**）中去实现消耗 CPU 时间的代码，并从该回调方法中启动那个活动对象。

此外，该客户端可以创建多个缓冲，填入数据，并将其传递给 `CMdaAudioOutputStream`。含有数据的缓冲可以随时被写入到 `CMdaAudioOutputStream` 中。

当调用 `CMdaAudioOutputStream::Stop()` [14] 在客户端完全复制描述符之前停止了流式播放，或当发生出错时，也会触发该回调方法。用于这个回调的参数是一个出错值，它指出复制是否成功，以及对所复制的缓冲的引用。如果复制成功，您将得到一个 `KErrNone` 值；否则将收到一个系统出错码。`KErrAbort` 表示该客户端在描述符被复制之前停止了流式播放。

用于回放过程的实现步骤如下面的时序图所示：



5.1.3 停止

停止音频流的播放是一个极其简单的过程：您只需调用 `CMdaAudioOutputStream::Stop()` [14] 函数停止向流中写入数据并停止播放音频。缓冲中余下的任何数据都被丢弃。请记住：该函数会触发 `MMdaAudioOutputStreamCallback::MaoscBufferCopied()` [14] 回调方法，如 5.1.2，“播放”一节中所讨论的，给出的出错代码指出：缓冲未被复制。

下面的代码片段为客户端应用实现了一个 `StopL()` 函数，其中调用了 `CMdaAudioOutputStream::Stop()` [14] 函数。

```

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

```

调用 `CMdaAudioOutputStream::Stop()` [14] 函数，并停止了流之后，还会调用 `MMdaAudioOutputStreamCallback::MaoscPlayComplete()` [14]。下面的代码片段展示了如何实现该回调方法的一个范例。

```

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

```

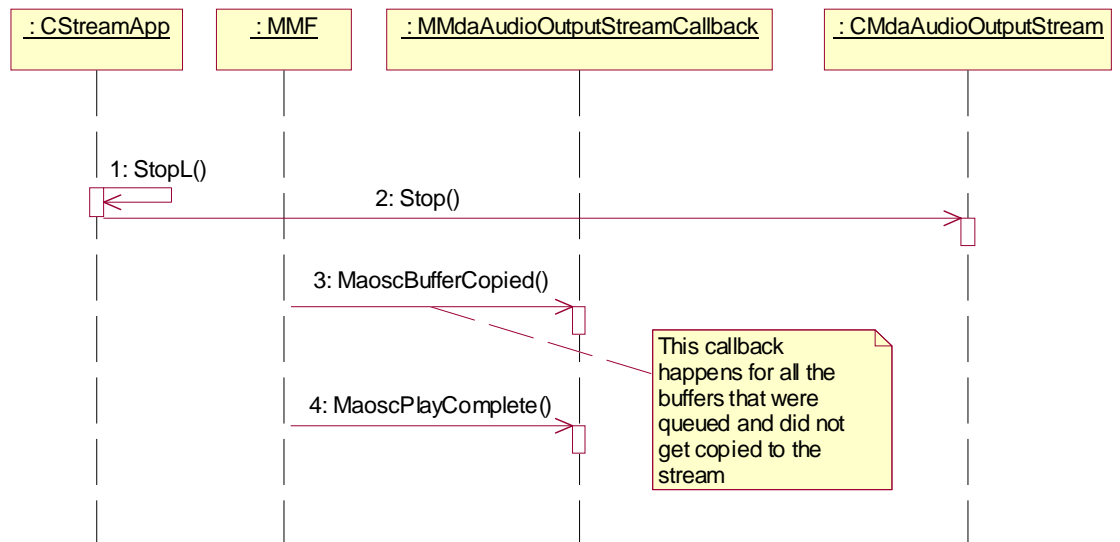
```

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

```

如前所述，停止流时会触发这个回调方法，但如果因其它一些原因而停止了播放也会触发该回调方法。如果到达了声音数据的尾部，它会返回一个 **KErrUnderFlow** 值；如果成功关闭，会返回 **KErrNone**，其他情况下则会返回一个适当的出错值。

以下时序图展示了停止过程：



5.1.4 析构

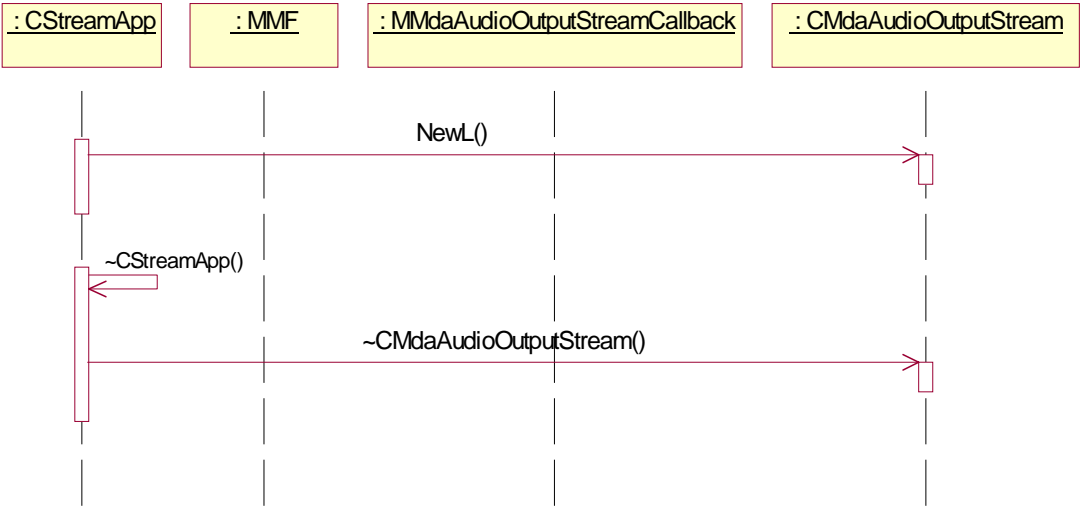
针对这个操作，需要销毁 **CMdaAudioOutputStream** 对象，可以在客户端应用的析构函数中完成。以下代码片段展示了这一过程：

```

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

```

下面的时序图说明了处理过程。



6. 在应用中通过流方式提供互联网音乐服务

可以创建通过流方式提供互联网音乐服务的应用程序，客户端应用和服务器基于 `socket` 协议建立连接。客户端应用需要实现必要的多媒体流式协议来从服务器（例如，`Shoutcast`，`RTSP/RTP/SDP`）上获得音乐。连接建立之后可以把音频数据从服务器传送到客户端应用，所读取的音频可以被放入缓存然后通过 `CMdaAudioOutputStream` API 进行回放。

关于如何用 `sokcet` 建立客户端与服务器端应用之间的连接，请查阅本文列出的参考文献 [15]。

当使用 `CMdaAudioOutputStream` API 对某个音频文件进行流处理以向平台发送音频数据时，您还可以规定该音频文件的数据类型，以便使用专门的编解码器，或确保编解码器能实现跨产品支持。为了实现这个目的，该 API 使用了名为 `TFourCC` [16] 的一个类，它具有代表着所支持的数据编码的四字符代码。这个四字符代码被压缩到一个单一的 `TUint32` 类型变量中。这些 `FourCC` 代码用于标识各种编解码器的数据类型表现。

`CMdaAudioOutputStream` API 提供了 `SetDataTypeL()` 函数，它将一个 `a TFourCC` [16] 对象作为参数传入。该函数帮助您设置该音频文件的数据类型。

由于本文第五章中讲解了如何用 `CMdaAudioOutputStream` API 对一个音频文件进行流处理，以及该处理过程所涉及到的所有任务，本章将只讲解如何使用 `SetDataTypeL()` 函数。

打开流以后可立即调用 `CMdaAudioOutputStream::SetDataTypeL(TFourCC aAudioType)` 函数。可以在 `MaoscOpenComplete()` 回调方法内部完成这一点。有关使用 `CMdaAudioOutputStream` API 对音频文件进行流处理的初始化过程请参见第五章，“从应用程序实现流式缓冲。”

以下代码片段展示了该函数的使用方法：

```
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 requiered for a general error . . .
    }
}
```

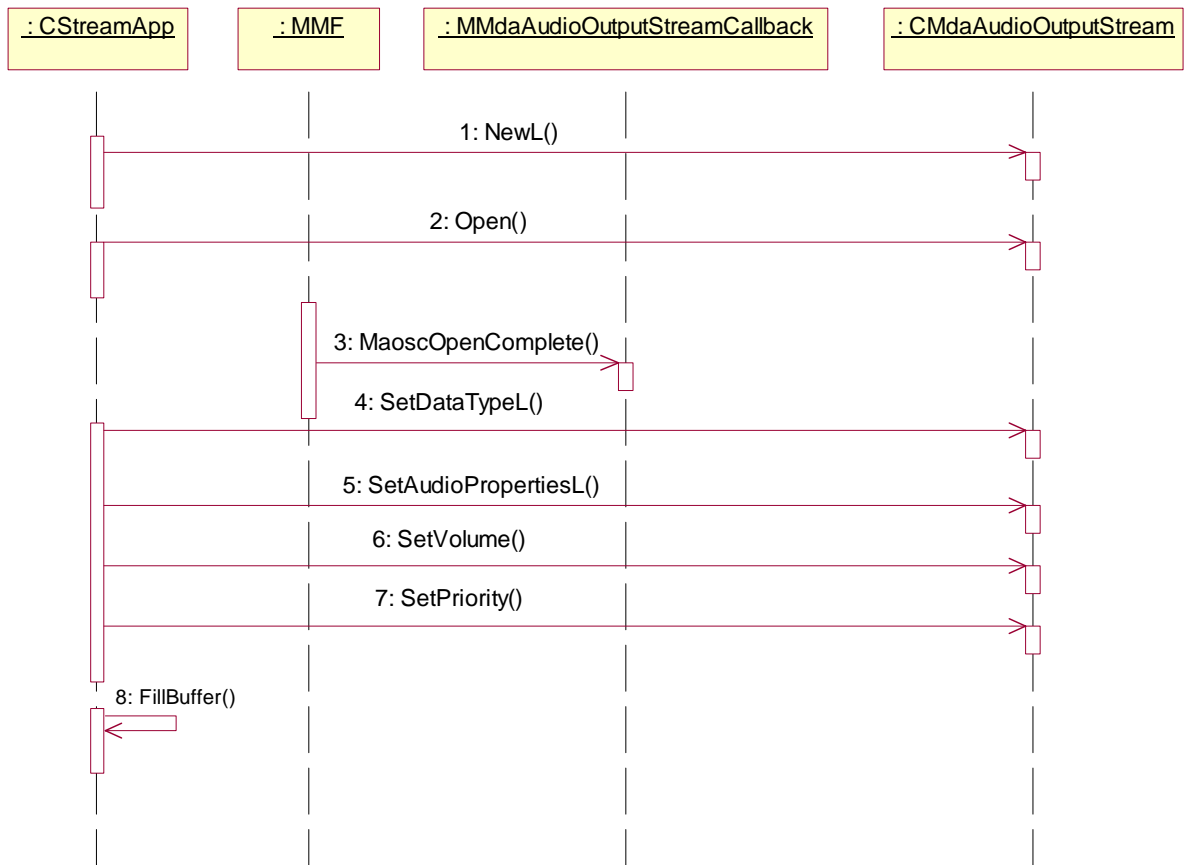
如果没有明确设置数据类型，`CMdaAudioOutputStream` 的默认设置是 `PCM16`。当为流式编码解码音频设置数据类型时，如果硬件并不支持将 `FourCC` 作为一个参数向该函数传入，该函数会抛出一个出错代码为 `KerrNotSupported` 的异常。

如前所述，该函数收到作为参数传入的 `FourCC` 代码，以便规定流式音频的数据类型。一些范例代码是：

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

这些代码的完整列表请参阅本文最后列出的参考文献[17]。

以下时序图展示了处理过程：



7. 参考文献

[1]	S60 SDK C++开发入门	www.symbian.com 或Symbian OS文档
[2]	Mmfbase.h	www.symbian.com 或Symbian OS文档
[3]	MdaAudioSamplePlayer.h	www.symbian.com 或Symbian OS文档
[4]	DrmAudioSamplePlayer.h	www.symbian.com 或Symbian OS文档
[5]	MdaAudioTonePlayer.h	www.symbian.com 或Symbian OS文档
[6]	MidiClientUtility.h	www.symbian.com 或Symbian OS文档
[7]	DRMHelper.h	www.symbian.com 或Symbian OS文档
[8]	DRMLicenseChecker.h	www.symbian.com 或 Symbian OS 文档
[9]	DRMCommon.h	www.symbian.com 或Symbian OS文档
[10]	e32std.h	www.symbian.com 或Symbian OS文档
[11]	MmfMeta.h	www.symbian.com 或Symbian OS文档
[12]	mmfcontrollerframeworkbase.h	www.symbian.com 或Symbian OS文档
[13]	Audio Capabilities	www.forum.nokia.com/audiovideo
[14]	MdaAudioOutputStream.h	www.symbian.com 或Symbian OS文档
[15]	http://www.symbian.com/developer/techlib/papers/Sockets/sockets.html	www.symbian.com 或Symbian OS文档
[16]	mmf\common\MmfUtilities.h	www.symbian.com 或Symbian OS文档
[17]	\mmf\common\mmffourcc.h	www.symbian.com 或Symbian OS文档

8. 术语与缩略语

术语或缩略语	说明
API	Application Programming Interface, 应用编程接口
AU	Default SUN Systems audio format, SUN Systems 支持的音频格式
CODEC	Coder/Decoder, 编码器/解码器。编码解码器是一个软件或硬件模块, 它将某个特定的文件格式转换 (压缩/解压缩/码流转换) 到另一种格式。
DLL	Dynamic Link Library, 动态链接库
DRM	Digital Rights Management, 数字版权管理
DTMF	Dual Tone Multi-Frequency, 双音多频
METADATA	在某个片段中保留的信息。Metadata (元数据) 通常用于储存诸如版权、作者、创作日期等信息。仅有某些音频格式能使用元数据。
MIDI	Musical Instrument Digital Interface, 音乐乐器数字接口
MMF	Multi Media Framework, 多媒体框架
PCM	Pulse Code Modulation, 脉码调制。一种通用的线性音频编码格式。
PLUG-IN	A DLL that is pluggable into the existing Multimedia Framework, 能被插入到现有多媒体框架中的一个 DLL。
RAM or ram	针对 RealNetworks 的简单 URL 描述符文件的文件扩展名
STREAMING	从服务器向客户端传输实时数据、音频和视频。客户端将对数据进行解码并播放。
UI	User Interface, 用户界面
URL	Uniform Resource Locator, 统一资源定位器
WAV	微软 Windows 标准采样音频格式。

9. 对本文评分

请花一点时间帮助我们改进文档质量，也帮助我们了解您认为最有价值的资源。请[对本文打分](#)。