PowerVR also publishes SDKs and code samples:

- ***www.imgtec.com/PowerVR/insider/index.asp***.

The Symbian Developer Library contains information about OpenGL ES in the Graphics section of the Symbian OS Guide. You can find this documentation in your development SDK, or read it online on the Symbian Developer Network:

- ***developer.symbian.com/main/oslibrary/osdocs***.

Both S60 and UIQ SDKs can be upgraded to support OpenGL ES 1.1:

- ***www.developer.sonyericsson.com/getDocument.do?docId=84947***.
- ***www.forum.nokia.com/info/sw.nokia.com/id/36331d44-414a-4b82-8b20-85f1183e7029/OpenGL_ES_1_1_Plug_in.html***.

The Symbian Press book *Games on Symbian OS* discusses OpenGL ES and other Khronos standards. It also includes further discussion of the factors to consider when creating smartphone games:

- ***developer.symbian.com/gamesbook***.

4.7 Multimedia

These recipes discuss aspects of working with multimedia files, such as playing and recording audio or video.

Symbian OS provides a framework called MMF (Multimedia Framework) which supports the following:

- audio playing, recording and conversion
- audio streaming
- tone playing
- video playing and recording.

The MMF is an extensible framework that allows phone manufacturers and third parties to add plug-ins to provide support for audio and video formats. For the application developer, it provides APIs that abstract the underlying hardware, thereby simplifying the code needed to record and play the content. The streaming APIs provided by the framework, which bypass large parts of the MMF, offer a lower-level interface that allows streaming of audio data to and from the audio hardware.

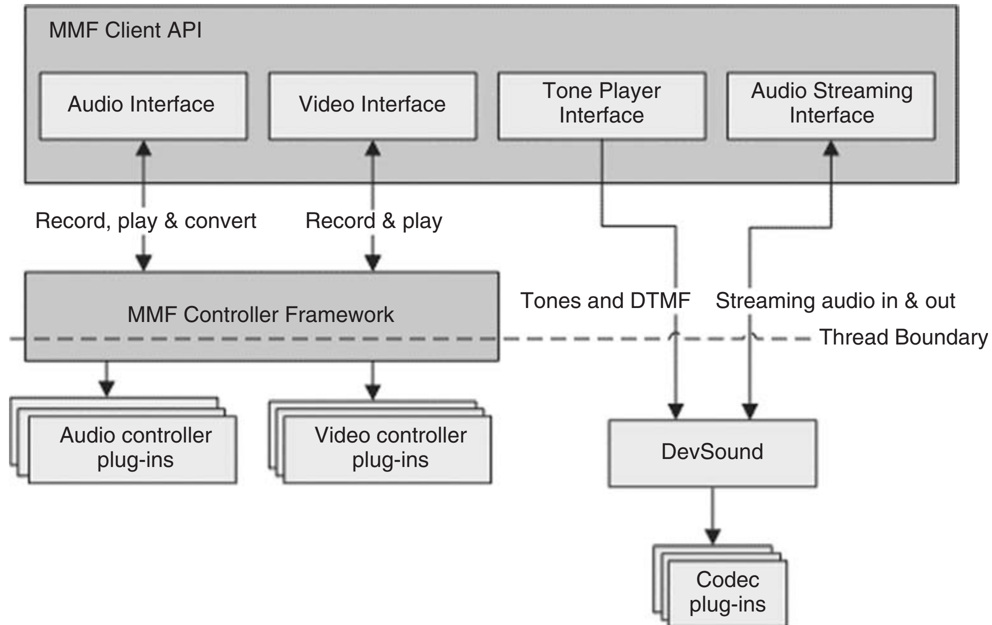

Figure 4.7.1 Architecture of MMF

Figure 4.7.1 shows the architecture of MMF.

The default Symbian OS MMF implementation supports basic audio formats. For example, for audio playback, it supports AU and WAV files and provides codecs for pulse code modulation (PCM) format. Furthermore, in addition to basic formats, most Symbian OS phones provide support for playback of other popular formats. For audio, this includes MP3, advanced audio coding (AAC), and adaptive multi-rate (AMR). Phone manufacturers supply additional controller plug-ins only to selected devices because of dependencies on specific accelerated hardware components, licensing issues, DRM requirements, or other business factors.

In this set of recipes, we will also discuss the Onboard Camera API, which is a generic and extensible API for controlling an onboard digital camera on devices. It supports either still images or videos. Please refer to the recipes in Section 4.5 for a discussion about image processing and general 2D graphics.

As a client of the Symbian OS multimedia APIs, you'll find a lot of the code you write will be implementation of the observer interfaces that are passed to the API and called by the MMF to signal an event or change of state. There is quite a lot to discuss in these recipes and, for clarity, we are going to quote the minimum of source code possible. You can find the full set of example code available for download at ***developer.symbian.com/symbian_press_cookbook***.

Further information about MMF and working with multimedia content on Symbian OS is available from the Symbian Developer Library documentation in your SDK, or online on the Symbian Developer Network (at ***developer.symbian.com/main/oslibrary/osdocs***).

4.7.1 Easy Recipes

4.7.1.1 Play an Audio Clip

Amount of time required: 15 minutes
Location of example code: `\Multimedia\AudioPlaying`
Required libraries: `mediaclientaudio.lib`
Required header file(s): `MdaAudioSamplePlayer.h`
Required platform security capability(s): None

Problem: You want to play an audio clip from a file, such as a WAV, MP3 or AAC file.

Solution: The client API to play an audio clip from a file is the `CMdaAudioPlayerUtility` class. `CMdaAudioPlayerUtility` requires an observer class, which implements `MMdaAudioPlayerCallback`. The observer will be notified when the audio clip has been initialized or played completely.

The following steps explain how to play an audio clip using `CMdaAudioPlayerUtility`:

1. Create an observer class, which implements `MMdaAudioPlayerCallback`.
2. Create an instance of `CMdaAudioPlayerUtility` and pass it a reference to the observer class.
3. Open the audio clip by calling `CMdaAudioPlayerUtility::OpenFileL()`.
4. Wait until `MMdaAudioPlayerCallback::MapcInitComplete()` is called, with a result indicating either that the file has been opened successfully, or that an error occurred.
5. Call `CMdaAudioPlayerUtility::Play()` to start audio playback.
6. When the audio has been played completely, or if an error occurs, `MMdaAudioPlayerCallback::MapcPlayComplete()` will be called.
7. Call `CMdaAudioPlayerUtility::Close()` to close the file. If you don't close the audio file, there will be a memory leak.

You'll see in the example code for the recipe that the CSimple-AudioPlayer class not only creates an instance of CMdaAudio-PlayerUtility but also serves as observer, by derivation from MMda-AudioPlayerCallback. Figure 4.7.2 shows a sequence diagram for the steps above.

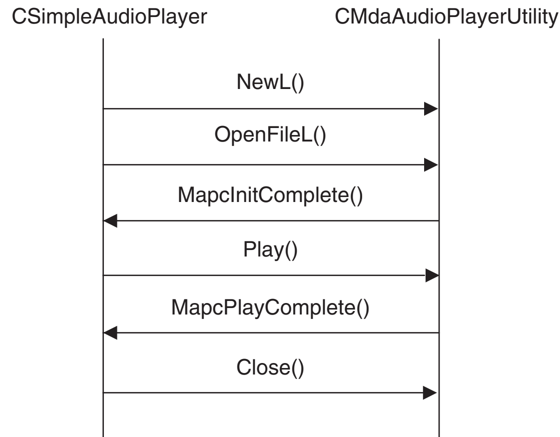

Figure 4.7.2 Sequence Diagram for Playing an Audio Clip from File, Showing the CMda-AudioPlayerUtility Class and the Client Class, which Implements the MMdaAudio-PlayerCallback Observer

Discussion: If your application has the MultimediaDD capability (see Chapter 3 for further discussion of platform security capabilities) you can successfully override the default priority and priority preference parameters of the CMdaAudioPlayerUtility::NewL() method. However, none of the examples in this book require this capability.

Besides NewL(), class CMdaAudioPlayerUtility offers some other factory methods:

- NewFilePlayerL(), to construct an audio player to play a file specified by filename.
- NewDesPlayerL(), to construct an audio player utility and play data from a descriptor.
- NewDesPlayerReadOnlyL(), to construct an audio player utility and play from a read-only descriptor.

Note that the three methods both open and initialize the audio clip. It means MMdaAudioPlayerCallback::MapcInitComplete() will be called once the initialization is complete. This is different from the NewL() method, because the latter does not initialize the audio player utility, and requires a separate call to OpenFileL(). The OpenFileL()

method opens an audio clip, which can either be specified by filename or by passing a handle to the file.

Note that we cannot play the file right after `CMdaAudioPlayerUtility::OpenFileL()` has returned. We have to wait until `MMdaAudioPlayerCallback::MapcInitComplete()` is called on the observer.

There are two other variants of open methods in `CMdaAudioPlayerUtility`:

- `OpenDesL()`, to open an audio clip from a descriptor.
- `OpenUrlL()`, to open an audio clip from a URL.

> **What may go wrong when you do this:** Note that not all devices support all variants of the `OpenXyzL()` methods. For example, S60 devices don't support `OpenUrlL()`.

Once the player utility has been initialized, we are able to start the audio playback. It is done by calling `CMdaAudioPlayerUtility::Play()`. In most cases, you may want to adjust the volume before playing anything by calling `CMdaAudioPlayerUtility::SetVolume()`.

> **Tip:** If you get an extraneous sound at the start of your audio clip, use the `SetVolumeRamp()` function to get rid of it.

> **Tip:** The emulator normally supports only basic formats and codecs, such as AU and WAV. Other advanced formats and codecs, such as MP3, are currently not supported. This means you have to perform testing on the real devices.

> **What may go wrong when you do this:** Some audio clips may be protected using digital rights management (DRM), in formats such as OMA DRM or WMDRM. They cannot be played using `CMdaAudioPlayerUtility`.
>
> You can play DRM-protected files using a specific API for DRM-protected content, such as `CDrmPlayerUtility` in S60, which does not require the DRM capability.
>
> Alternatively, you can use the content access framework (CAF) to read DRM-protected files and pass them to MMF client APIs. Unfortunately, this requires your application to have DRM capability, which is one of the manufacturer-approved capabilities. It requires a

high level of negotiation to win the trust of the phone manufacturer before they will grant it to you. For details of how to apply for the DRM capability, see ***www.symbiansigned.com***.

Further discussion of how to play DRM-protected files is outside the scope of this book. However, you can find an example on the Forum Nokia developer wiki, at ***wiki.forum.nokia.com/index.php/Playing_DRM-protected_audio_file_using_CDrmPlayerUtility***.

4.7.1.2 Perform Basic Audio Operations

Amount of time required: 15 minutes
Location of example code: `\Multimedia\AudioPlaying`
Required libraries: `mediaclientaudio.lib`
Required header file(s): `MdaAudioSamplePlayer.h`
Required platform security capability(s): None

Problem: You want to do basic operations on the audio playback, such as stop, play, rewind and fast forward.

Solution: The `CMdaAudioPlayerUtility` class provides all methods needed to perform basic audio operations, such as:

- `Stop()`, to stop audio playback.
- `Pause()`, to pause audio playback.
- `SetPosition()`, to set the current playback position from the start of the clip.
- `GetPosition()`, to return the current playback position from the start of the clip.

Discussion: The audio playback can be stopped or paused at any time. To resume the audio playback, simply call `CMdaAudioPlayerUtility::Play()`.

Performing rewind and fast forward is done by calling `CMdaAudioPlayer::SetPosition()`. It requires a parameter in the type of `TTimeIntervalMicroSeconds`, which is the interval of the new position in microseconds from the start of the clip (not from the current position).

The following example shows how to rewind the current audio playback by one second:

```
const TInt KOneSecond = 1000000; // 1 second in microseconds
void CSimpleAudioPlayer::Rewind(TInt aIntervalInSeconds)
  {
  iPlayerUtility->Pause();

  // Get the current position of the playback.
  TTimeIntervalMicroSeconds position;
  iPlayerUtility->GetPosition(position);

  // Subtract the interval from the current position
  position = position.Int64() -
                      (aIntervalInSeconds*KOneSecond);

  // Set the new position.
  iPlayerUtility->SetPosition(position);
  iPlayerUtility->Play();
  }
```

What may go wrong when you do this: When you call `SetPosition()`, `CMdaAudioPlayerUtility` does not move the position immediately. It will continue playing for a couple of seconds until its internal buffer is empty. Sometimes, you may notice that the playback continues for a while before it moves to the new position. In order to have immediate rewind/fast forward effect, you must call `Pause()` before setting the new position, and then call `Play()` after that.

4.7.1.3 Play an Audio Tone

Amount of time required: 15 minutes
Location of example code: `\Multimedia\AudioTone`
Required libraries: `mediaclientaudio.lib`
Required header file(s): `MdaAudioTonePlayer.h`
Required platform security capability(s): None

Problem: You want to play an audio tone, such as a beep sound, or a sound from a certain frequency.

Solution: The `CMdaAudioToneUtility` class is used to play an audio tone. It uses an observer class, which implements `MMdaAudioToneObserver`. This observer will receive events and error notifications, for example when it has been configured, when the tone has been played completely, or when an error occurs.

The following steps explain how to play an audio clip using `CMdaAudioToneUtility`:

- Create an observer class, which implements `MMdaAudioToneObserver`. In the sample code, we use class `CAudioTonePlayer`.

- Create an instance of `CMdaAudioToneUtility` and pass it a reference to the observer class.
- Open the audio clip by calling one of the variants of the `PrepareToPlay()` method (`CMdaAudioToneUtility::PrepareToPlayXyz()`). See the Discussion section for further details.
- Wait until `MMdaAudioToneObserver::MatoPrepareComplete()` is called.
- Within the callback, call `CMdaAudioToneUtility::Play()` to start audio tone playback.
- When the audio has been played completely, or if an error occurs, `MMdaAudioToneObserver::MatoPlayComplete()` will be called.

Discussion: There are several variants of the `CMdaAudioToneUtility::PrepareToPlayXyz()` methods, for instance:

- `PrepareToPlayTone()` – used to play a single tone.
- `PrepareToPlayDualTone()` – used to play a dual tone, which is a combination of two frequencies.
- `PrepareToPlayDTMFString()` – used to play dual-tone multi frequency (DTMF) tones. These are used in the telephony signalling system.

> **Tip:** You cannot use `PrepareToPlayDTMFString()` to play DTMF tones to the telephony uplink. It only plays DTMF on the local speaker.

Please check the Symbian Developer Library documentation to see the complete list of methods supplied by `CMdaAudioToneUtility`.

4.7.1.4 Play a MIDI File

Amount of time required: 15 minutes
Location of example code: `\Multimedia\MidiPlaying`
Required libraries: `midiclient.lib`
Required header file(s): `midiclientutility.h`
Required platform security capability(s): None

Problem: You want to play a MIDI file on the device.

Solution: The client API to play a MIDI file is `CMidiClientUtility` class. Like the audio player utility, the MIDI client utility class requires an

observer, which must implement the `MMidiClientUtilityObserver` `interface`.

The following steps explain how to play a MIDI file using `CMidi-ClientUtility`:

- Create an observer class, which implements `MMidiClientUtilityObserver`.
- Create an instance of `CMidiClientUtility` and pass it a reference to the observer class.
- Open the audio clip by calling `CMidiClientUtility::OpenFile()`.
- Wait until `MMidiClientUtilityObserver::MmcuoStateChanged()` is called. The `aNewState` parameter has the value `EOpen` if the file has been opened successfully.
- Call `CMidiClientUtility::Play()` from within the observer to start MIDI playback.
- When the audio has been played completely, wait until `MMidiClientUtilityObserver::MmcuoStateChanged()` is called again. The `aOldState` parameter will be set to `EPlaying` and the `aNewState` parameter will be set to `EOpen`.

In our recipe, the `CMidiPlayer` class implements `MMidiClientUtilityObserver` and is used to play a MIDI file, `\data\sample.mid`, which is located at the same drive as the application. For example, if the sample application is installed on the C: drive, then the file location is `c:\data\sample.mid`.

Tip: The Windows emulator does not support MIDI playback. You can only test this example on the device.

4.7.2 Intermediate Recipes

4.7.2.1 Get the Default Multimedia Storage Location

Amount of time required: 25 minutes
Location of example code: `\Multimedia\AudioRecording` and `\Multimedia\CameraImage`
Header files: `pathinfo.h` (S60), `QikMediaFileFolderUtils.h` (UIQ)
Required libraries: `platformenv.lib` (S60), `qikutils.lib` (UIQ)
Required platform security capability(s): None

Problem: You need to retrieve the default path for storing multimedia files.

S60 and UIQ store multimedia files in different default folder locations. For example, the default path for audio files on S60 on the C: drive is `c:\data\sounds`, while on UIQ it is `c:\Media files\Music`.

The location used to store files on the emulator may also be different from that used on smartphone hardware. For example, the UIQ's emulator uses `c:\Media files\audio` for audio files, compared to `c:\Media files\Music` when storing files on the device.

And that's not all! Even when just considering phone hardware, the location in which files are stored in the phone memory may be different to that used on the memory card. For example, in S60 the default video location in phone memory is `c:\data\videos`, while on the memory card it is `e:\videos`.

How can we write a code that is able to return the media path independent from the platform and media type?

Solution: Unfortunately, there is no single solution to this problem. S60 and UIQ use different ways of getting media paths. You need to create two different implementations for each platform.

Discussion:

S60

The class to handle various media paths on S60 is `PathInfo`. It is declared in the `pathinfo.h` header file, and the library to link against is `platformenv.lib`.

Some of the methods related to media path information are as follows:

- `PathInfo::PhoneMemoryPath()`,
- `PathInfo::MemoryCardPath()`,
- `PathInfo::VideosPath()`,
- `PathInfo::ImagesPath()`,
- `PathInfo::SoundsPath()`.

The `PhoneMemoryPath()` method returns the root path in the phone memory, for example '`c:\data\`'. The `MemoryCardPath()` method returns the root path on the memory card, for example '`e:\`'. Note that there is a backslash character at the end of the returned path.

The other methods return the path of multimedia files. For example, `VideosPath()` returns '`Videos\`'. Again, there is a backslash character at the end of the returned path.

The following code shows how to get an absolute path of the audio folder including the drive letter:

```
void CAudioRecordingAppUi::GetAudioPathL(TChar aDriveLetter,
                                          TDes& aPath)
  {
  aPath.Zero();
  if ((aDriveLetter == 'c') ||  (aDriveLetter == 'C'))
    {
    aPath.Append(PathInfo::PhoneMemoryRootPath());
    }
  else if ((aDriveLetter == 'e') ||  (aDriveLetter == 'E'))
    {
    aPath.Append(PathInfo::MemoryCardRootPath());
    }
  aPath.Append(PathInfo::SoundsPath());
  }
```

The following example shows how to call the method above:

```
TFileName audioPath;
GetAudioPathL('c', audioPath);
```

The audioPath will have the value of 'c:\data\sounds\' after GetAudioPathL() is called.

Similarly, if you call it using the statement:

```
GetAudioPathL('e', audioPath);
```

the audio path will have the value of 'e:\sounds'.

UIQ

Now, let's take a look at UIQ. The class to handle various media paths is CQikMediaFileFolderUtils. It is declared in the QikMediaFileFolderUtils.h header file. The library name is qikutils.lib.

The methods used to retrieve paths related to multimedia files are GetDefaultPathForMimeType() and GetDefaultPathForMimeTypeL().

The only difference between them is that the latter method can leave. The methods require three parameters. The first parameter, aMimeType, is the MIME type to convert to a folder path. It uses only the first part of MIME type. For example, if the MIME type is 'audio/wav', only 'audio' will be used for matching.

The second parameter, aDriveLetter, is the drive to query for the default location. The final parameter, aFolderPath, is a reference parameter which receives the absolute path of the media files excluding drive letter. For example, it returns ':\Media files\audio\' for audio files.

> **Tip:** You can also retrieve the root path of the media folder (i.e., ':\Media files\') using the `CQikMediaFileFolderUtils::GetMediaFilesRootL()` method.

The following code shows how to get the audio path in UIQ:

```
void CAudioRecordingAppUi::GetAudioPathL(TChar aDriveLetter,
                                           TDes& aPath)
  {
  CQikMediaFileFolderUtils* mediaUtils =
                  CQikMediaFileFolderUtils::NewL(*iEikonEnv);
  CleanupStack::PushL(mediaUtils);

  // Get the path for image
  TFileName mediaPath; // To receive the path
  mediaUtils->GetDefaultPathForMimeTypeL(KAudioMimeType,
                              aDriveLetter, mediaPath);

  // Construct a full path that contains drive letter.
  aPath.Zero();
  aPath.Append(aDriveLetter);
  aPath.Append(mediaPath);

  CleanupStack::PopAndDestroy(mediaUtils);
  }
```

The following example shows how to call the method above:

```
_LIT(KAudioMimeType, "audio/wav");
TFileName audioPath;
GetAudioPathL('c', audioPath);
```

The `audioPath` will have the value of 'c:\Media files\music\' on the UIQ device and 'c:\Media files\audio\' on the emulator.

Similarly, if you call it using the statement below,

```
GetAudioPathL('d', audioPath);
```

the audio path will have the value of 'd:\music\' on the UIQ device and 'd:\Media files\audio\' on the emulator.

4.7.2.2 Play a Video Clip

Amount of time required: 30 minutes
Location of example code: `\Multimedia\VideoPlaying`
Required libraries: `mediaclientvideo.lib`
Required header file(s): `VideoPlayer.h`
Required platform security capability(s): None

Problem: You want to play a video clip from a file, such as a 3GP or MP4 file.

Solution: The MMF class that is used to play video is `CVideoPlayerUtility`, which requires you to pass an implementation of the `MVideoPlayerUtilityObserver` observer class.

Using `CVideoPlayerUtility` is similar to `CMdaAudioPlayerUtility`, except there is an additional step; that is, preparing the video clip to be played. It is not possible to play the video clip before `CVideoPlayerUtility` finishes the preparation.

The following steps explain how to play a video clip using `CMdaVideoPlayerUtility`:

- Create an observer class, which implements `MVideoPlayerUtilityObserver`.
- Create an instance of `CVideoPlayerUtility` and pass it a reference to the observer.
- Open the video clip by calling `CVideoPlayerUtility::OpenFileL()`.
- Wait until `MVideoPlayerUtilityObserver::MvpuoOpenComplete()` is called.
- Prepare the video clip to be accessed by calling the `CVideoPlayerUtility::Prepare()` method.
- Wait until `MVideoPlayerUtilityObserver::MvpuoPrepareComplete()` is called.
- Call `CVideoPlayerUtility::Play()` to start video playback.
- When the video has been played completely, `MVideoPlayerUtilityObserver::MvpuoPlayComplete()` will be called.

In the full sample code for this recipe, the `CSimpleVideoPlayer` class implements `MVideoPlayerUtilityObserver`.

Discussion: The factory methods of `CSimpleVideoPlayer` require one parameter to be passed. The type is `CCoeControl&`, and the parameter is the control where the video is to be displayed.

Let's take a look at the constructor of `CVideoPlayerUtility` to see how it is going to be used. The constructor of `CVideoPlayerUtility` is defined as follows:

```
CVideoPlayerUtility* NewL(
      MVideoPlayerUtilityObserver& aObserver,
      TInt aPriority, TMdaPriorityPreference aPref,
      RWsSession& aWs,
```

```
    CWsScreenDevice& aScreenDevice,
    RWindowBase& aWindow,
    const TRect& aScreenRect,
    const TRect& aClipRect);
```

The aObserver parameter is the reference to the observer which will receive notifications.

The aPriority and aPref parameters are the video client's priority and preference, respectively. Like CMdaAudioPlayerUtility, they require MultimediaDD capability. The discussion of MultimediaDD is not in the scope of this book.

The aWs parameter is the reference to the window server session. You can use the shared window server session defined in CCoeEnv::WsSession(). For example:

```
RWsSession& wsSession = aControl.ControlEnv()->WsSession();
```

The aScreenDevice is the reference to the software device screen. You can usually use the default screen device owned by CCoeEnv, which is CCoeEnv::ScreenDevice(). For example:

```
CWsScreenDevice* screenDevice = aControl.ControlEnv()->ScreenDevice();
```

The aWindow parameter is the handle of the window for the video. You can use CCoeControl::DrawableWindow() to get the client-side handle of a control:

```
RWindowBase& windowBase = aControl.DrawableWindow();
```

The aScreenRect parameter is the rectangle where the video is displayed on the screen. The position is relative to the origin of the screen, not to the origin of the control.

The aClipRect parameter is the area of the video clip to be displayed. In most cases, this parameter has the same value as aScreenRect, which means the whole area of the video is displayed. The same as aScreenRect, the position is relative to the origin of the screen, not to the origin of the control.

The supported codecs and formats of CVideoPlayerUtility depend on the installed plug-in on the device. For example, most Symbian OS devices support the H.263 codec, but only some support the H.264 codec.

Note that there are some other methods to open video clips from other sources, such as CVideoPlayerUtility::OpenUrlL().

The CVideoPlayerUtility::SetDisplayWindowL() method changes the display window of the video. It can be used to display the

video playback to another control. It can also be used to change the area where the video is played. Its parameters are the same as those used in the constructor of `CVideoPlayerUtility`.

What may go wrong when you do this: There is one additional thing that you need to take care of; that is, the possibility of a screen orientation change. For example, some S60 devices allow orientation change by opening the cover of the device. If you don't respond to the orientation change, your video will not be displayed properly.

What you need to do is override the `CEikAppUi::HandleResourceChangeL()` method and then call `CVideoPlayerUtility::SetDisplayWindowL()` to update the position and size of the video playing area.

4.7.2.3 Audio Streaming

Amount of time required: 30 minutes
Location of example code: `\Multimedia\AudioStreaming`
Required libraries: `mediaclientaudiostream.lib`
Required header file(s): `MdaAudioOutputStream.h`
Required platform security capability(s): None

Problem: You want to play an audio clip in streaming mode. The audio clip is read chunk by chunk incrementally. The audio streaming may be needed; for example, you want to process the audio from the file before you play it or you are getting the audio clip from the network. An Internet radio is an example of an application that needs audio streaming.

Solution: The class to stream audio is `CMdaAudioOutputStream`.

The following lists the steps to stream audio:

- Create an instance of `CMdaAudioOutputStream`, which implements `MMdaAudioOutputStreamCallback`.
- Open the audio stream package by calling `CMdaAudioOutputStream::Open()`.
- Once the stream has been opened, the callback method, `MMdaAudioOutputStream::MaosOpenComplete()`, is called.
- Start streaming audio by calling `CMdaAudioOutputStream::WriteL()`.
- Once the buffer has been copied to the lower layers of MMF, the callback method, `MMdaAudioOutputStream::MaosBuffer-`

Copied(), is called. Now, we can call WriteL() to copy the next buffer.

In our recipe, the CAudioStreamPlayer class implements MMdaAudioOutputStream.

Discussion: The CMdaAudioOutputStream class plays the audio using the sound driver. It relies on hardware DSP (Digital Signal Processing) codecs. No MMF controller plug-in is involved. The supported formats depend on the device's DSP. All Symbian OS devices support PCM formats. Some of them support other compressed formats, such as AMR and MP3.

The CMdaAudioOutputStream::Open() requires one parameter with the type of TMdaPackage*. You can ignore this parameter because it is maintained for historical reasons only.

After MMdaAudioOutputStreamCallback::MaoscOpenComplete() is called, you are ready to stream the audio clip. There are several properties that need to be set to match the data you are streaming, such as format, sampling rate and number of channels.

The format is set by calling the CMdaAudioOutputStream::SetDataTypeL() method. It requires one parameter in the type of TFourCC, which is the FourCC (Four-Character Code) of the audio:

```
TRAP(err, iAudioStream->SetDataTypeL(KMMFFourCCCodePCM16));
```

Note that we use a TRAP because we call this method inside a non-leaving method, MaoscOpenComplete().

The list of FourCC constants can be found in \epoc32\include\mmf\common\MmfFourCC.h.

Here are some examples of the predefined FourCC constants:

- KMMFFourCCCodePCM8 = (' ', ' ', 'P', '8')
- KMMFFourCCCodePCM16 = (' ', 'P', '1', '6')
- KMMFFourCCCodeAMR = (' ','A','M','R')
- KMMFFourCCCodeAAC = (' ','A','A','C')
- KMMFFourCCCodeMP3 = (' ','M','P','3').

You can use CMMFDevSound::GetSupportedOutputDataTypesL() to get the list of supported FourCCs on a particular device. Note that some SDKs may not distribute the header file of CMMFDevSound.

The sampling rate and number of channels can be set by calling CMdaAudioOutputStream::SetAudioPropertiesL().

The possible values for the sampling rate and number of channels are defined in `TMdaAudioDataSettings`. There are several sampling rates, starting from `ESampleRate8000Hz` to `ESampleRate64000Hz`. There are two supported channels: `EChannelsMono` and `EChannelsStereo` (no surround sound).

After all the properties have been set up, you can start writing the data stream to the audio device. This is done by calling `CMdaAudio-OutputStream::WriteL()`. It requires a parameter with the type of `TDesC8&`:

```
TRAP(err, iAudioStream->WriteL(iBuffer));
```

The optimal size of the buffer depends on your needs and the audio format. Ideally, you want to use the smallest possible amount of memory without risking a buffer underflow situation.

Tip: In order to avoid an out-of-data situation, you may want to use more than one buffer. For example, you may use two buffers. You pass one buffer to the audio stream and use the other buffer to read from the file. When the one buffer is passed to the audio device, you can replace it with the next data. The example in this book uses one buffer for simplicity reasons.

In some applications that have a heavy thread, such as games, a large buffer may not solve the underflow situation. In this case, you may consider running the audio stream in a separate thread with higher priority.

Some audio-intensive applications may need to create proper adaptive buffer management algorithms to deal with different performance situations.

You can stop the streaming by calling `CMdaAudioOutputStream::Stop()`. Note that calling this method will cause the callback method, `MaoscBufferCopied()`, to be called with `KErrAbort` error code, and `MaoscPlayComplete()` with `KErrCancel`. If you are handling the error code in those methods, you have to exclude this situation. Otherwise, you will get an error message when stopping the audio streaming.

What may go wrong when you do this: After you have finished playing the stream, you have to stop the stream or destroy the instance of `CMdaAudioOutputStream`. If you don't do this, the audio device is actually still on. This will drain the battery.

4.7.3 Advanced Recipes

4.7.3.1 Record Audio

Amount of time required: 20 minutes
Location of example code: `\Multimedia\AudioRecording`
Required libraries: `mediaclientaudio.lib`
Required header file(s): `MdaAudioSampleEditor.h`
Required platform security capability(s): `UserEnvironment`

Problem: You want to record audio from the device microphone or telephone downlink.

Solution: The class to record audio is `CMdaAudioRecorderUtility`.

Here are the steps to record audio:

- Create a new instance of `CMdaAudioRecorderUtility`, which implements `MMdaObjectStateChangeObserver`.
- Open the file by calling `CMdaAudioRecorderUtility::OpenFileL()`.
- Wait until the observer method, `MMdaObjectStateChangeObserver::MoscoStateChangeEvent()`, is called.
- Start recording by calling `CMdaAudioRecorderUtility::RecordL()`.
- Stop recording by calling `CMdaAudioRecorderUtility::Stop()`.

The recording method, `CMdaAudioRecorderUtility::RecordL()`, requires the `UserEnvironment` capability.

Like other audio APIs, the supported formats and codecs vary between devices.

In our recipe, the `CSimpleAudioRecorder` class implements `MMdaObjectStateChangeObserver`.

Discussion: `CMdaAudioRecorderUtility::OpenFileL()` opens the file where the audio sample data will be recorded. It selects the MMF controller automatically based on the file extension. For example, when you specify `.amr` as the file extension, the controller plug-in for AMR will be loaded.

When the audio file has been opened, the observer method, `MoscoStateChangeEvent()`, is called. This callback method has four parameters:

```
void CSimpleAudioRecorder::MoscoStateChangeEvent(CBase* aObject,
    TInt aPreviousState, TInt aCurrentState, TInt aErrorCode)
```

The `aObject` parameter is the object of the recorder utility. It is useful when you have more than one recorder utility object.

The `aPreviousState` and `aCurrentState` parameters are the previous state and the current state of the audio sample, respectively. There are several values defined in `CMdaAudioClipUtility` that can be assigned to them:

- `ENotReady`. The audio clip recorder has been constructed but no file has been opened.
- `EOpen`. The file is opened but no recording or playing is in progress.
- `ERecording`. New audio sample data is being recorded.
- `EPlaying`. Audio sample is being played.

`aErrorCode` is the system-wide error code, as defined in `\epoc32\include\e32err.h`.

What may go wrong when you do this: The `RecordL()` method appends the new recorded audio to an existing file. There is a method, `CMdaAudioRecoderUtility::CropL()`, that is supposed to discard any existing data before recording. Unfortunately, it does not work with AMR on some S60 devices. In order to support compatibility with as many devices as possible, it is recommended not to use `CropL()`. If you want to delete existing data, you can use `BaflUtils::DeleteFile()` before recording audio.

What may go wrong when you do this: If you don't specify the `UserEnvironment` capability in the MMP file, you will get the `KErrAccessDenied (-21)` error code in the `MoscoStateChangeEvent()` observer.

If your recorded clip is distorted, try a lower gain value by calling `CMdaAudioRecoderUtility::SetGain()`.

There are several recording settings that can be adjusted, such as codec, sampling rate, bit rate and number of channels. The recipe uses the default settings.

Tip: It is possible to set the maximum length of the file that is being recorded by calling `CMdaAudioRecorderUtility::SetMaxWriteLength()`. It requires one parameter, which is the maximum file size in **bytes** (not kilobytes, as indicated by some SDK documentation).

> **Tip:** The `CMdaAudioRecorderUtility` class can also be used to play an audio file, using the `PlayL()` method. Take a look at `CSimpleAudioRecorder::PlayRecordedL()` in the recipe to see how to use it.

4.7.3.2 Record a Phone Call

The previous recipe, which showed audio recording, can be used to record a phone call. Once you have called `CMdaAudioRecorderUtility::RecordL()`, audio from the local speaker and telephony downlink will be recorded. Unfortunately, it is not possible to record solely from one source (local speaker or telephony downlink only) because the API, `CMdaAudioRecorderUtility::SetAudioDeviceMode()`, has been deprecated.

It is also possible to create an answering machine that automatically records a caller's message from the telephony downlink. To create such an application, you need to use `CMdaAudioRecorderUtility` and `CTelephony`. Please see the recipes in Section 4.8 for a discussion about `CTelephony`. The idea is to start recording once an incoming call is detected

The audio recording recipe would have to be modified to monitor telephony events to decide when to start and stop recording.

> **Tip:** Recording a phone conversation is illegal in some countries. Some of them require the person to be notified that the conversation is recorded. That is why some Nokia phones output a beep sound every few seconds when recording is happening. This is to notify the other party on the line that the conversation is being recorded.

4.7.3.3 Display a Camera Viewfinder

Amount of time required: 30 minutes
Location of example code: `\Multimedia\CameraImage`
Required libraries: `ecam.lib`
Required header file(s): `ecam.h`
Required platform security capability(s): `UserEnvironment`

Problem: You want to display the viewfinder of a camera on your application.

Solution: Symbian OS provides the Onboard Camera API, which is an open and extensible generic API for controlling digital camera devices. It can be used to capture still images and record videos.

The CCamera class provides the base class for camera devices. It provides virtual methods for controlling a camera, such as acquiring images and videos. Phone manufacturers derive CCamera and provide their implementations.

There are two observer classes for CCamera: MCameraObserver and MCameraObserver2. MCameraObserver2 is the recommended API. Unfortunately, at the time of writing this book, S60 devices do not support MCameraObserver2 yet. In order to maintain compatibility with S60 devices, this book uses MCameraObserver in the examples.

In our recipe, the viewfinder is handled by the CCameraEngine class.

Discussion: A number of Symbian smartphones have two camera devices, which is why the NewL() factory method of the CCamera class allows you to choose a camera index between 0 and (CCamera::Cameras-Available() - 1). The camera at index 0 is usually the one with the higher resolution, pointing away from the handset user. A bad camera index will cause a panic.

Getting camera information

The TCameraInfo class specifies camera information, such as supported image format, flash support, zoom support and many others. You can get camera information by calling CCamera::CameraInfo().

In the recipe, we use TCameraInfo::iOptionsSupported, which has the type of TOptions. It is a bit flag that stores the camera's supported options. Here is the list of possible values of the bit flag:

- EViewFinderDirectSupported
- EViewFinderBitmapsSupported
- EImageCaptureSupported
- EVideoCaptureSupported
- EViewFinderMirrorSupported
- EContrastSupported
- EBrightnessSupported
- EViewFinderClippingSupported
- EImageClippingSupported
- EVideoClippingSupported.

There are several values that are related to the viewfinder, but for this example we will be using the first two values only. There are basically two viewfinder types:

- Direct viewfinder (`EViewFinderDirectSupported`). The viewfinder can be drawn using direct screen access. It is easier to use and has a better performance. Unfortunately, not all devices support this mode; for example, at the time of writing this book, S60 devices do not support direct viewfinder mode.
- Bitmap viewfinder (`EViewFinderBitmapsSupported`). The viewfinder has to be drawn manually via a bitmap to a device context.

Initializing camera

There are two steps involved in initializing the camera:

- Reserve the camera for exclusive use by calling `CCamera::Reserve()`. This is an asynchronous method. When complete, it will call `MCameraObserver::ReserveComplete()`.
  The camera reservation is based on priority. That means, if there are two applications attempting to reserve a camera, the application with the higher priority will preempt the lower one. As with audio, you need `MultimediaDD` capability to change the priority.
- Switch on the camera power by calling `CCamera::PowerOn()`. This has to be called after `Reserve()` has been called. It is also an asynchronous method. When complete, it will call `MCameraObserver::PowerOnComplete()`.

After you have finished using the camera, you have to turn it off by calling `CCamera::PowerOff()`. Otherwise, your application will drain the battery very fast.

What may go wrong when you do this: The camera consumes a lot of battery power. If you don't use the camera, you have to turn it off. You can react to your application being sent to the background by overriding `CAppUi::HandleForegroundEventL()`.

Displaying the viewfinder

There are two methods to display the viewfinder. These are `CCamera::StartViewFinderDirectL()` and `CCamera::StartViewFinderBitmapsL()`. Each method has some more variants; we will not discuss all of them.

`CCamera::StartViewFinderDirectL()` is used to start the direct viewfinder:

```
virtual void StartViewFinderDirectL(RWsSession& aWs,
    CWsScreenDevice& aScreenDevice, RWindowBase& aWindow,
                                   TRect& aScreenRect)=0;
```

You would usually use `CCoeEnv::WsSession()`, `CCoeEnv::ScreenDevice()` and `CCoeControl::DrawableWindow()` to assign the first three parameters.

The last parameter is the area where the viewfinder will be displayed. The position is relative to the origin of the screen.

`CCamera::StartViewFinderBitmapsL()` is used to start the bitmap viewfinder:

```
virtual void StartViewFinderBitmapsL(TSize& aSize);
```

`StartViewFinderBitmapsL()` requires one parameter with type `TSize`, which is the size of the viewfinder to be used. When calling `StartViewFinderBitmapsL()`, you have to initialize `aSize` with the area where you want to display the viewfinder. On return, `aSize` will be assigned to the actual size of the viewfinder. They may be different because the control size and the viewfinder size may have different aspect ratio.

You can only display the viewfinder after the camera's power has been switched on. This means you have to do it after `MCameraObserver::PowerOnComplete()` has been called.

In the case of a direct viewfinder, you don't need to do anything else after calling `StartViewFinderDirectL()`. The viewfinder drawing is done direct to the screen by the API.

When using a bitmap viewfinder, you have to draw the viewfinder manually. `MCameraObserver::ViewFinderFrameReady()` is called periodically when the new viewfinder frame is ready. This callback has one parameter with type `CFbsBitmap&`, which is the viewfinder to be displayed.

There are applications that may require the viewfinder to process the image from the camera without actually capturing it. For example, it can be used in a motion detector application. When the application detects a moving object, it performs a specific action (this is discussed further in the Symbian Press book *Games on Symbian OS* – see ***developer.symbian.com/gamesbook*** for more details). A game may also use a viewfinder as its controller. When the user moves the camera in one direction, a character in the game also moves in the same direction.

4.7.3.4 Capture Still Images from a Camera

Amount of time required: 20 minutes
Location of example code: `\Multimedia\CameraImage`
Required libraries: `ecam.lib`
Required header file(s): `ecam.h`
Required platform security capability(s): `UserEnvironment`

Problem: You want to capture still images from the device's camera. The images are then saved to a file in JPEG format.

Solution: In the recipe, the `CCameraEngine::CaptureImageL()` method is used to capture still images from the camera. This is an asynchronous method. When complete, it calls `CCameraEngine::DoSaveImageL()` to save the image to a file.

Discussion:

Preparing image capture

Capturing a still image using a camera is done by calling `CCamera::CaptureImage()`. This is an asynchronous method. When it completes, it calls `MCameraObserver::ImageReady()`.

Before capturing any image, the `CCamera::PrepareImageCaptureL()` method has to be called to keep the latency of `CaptureImage()` to a minimum. It needs to be called only once for multiple `CaptureImage()` calls.

The `PrepareImageCaptureL()` method requires two parameters. The first parameter is the format of the image. Here is the list of formats for still images:

- `EFormatMonochrome`
- `EFormat16bitRGB444`
- `EFormat16BitRGB565`
- `EFormat32BitRGB888`
- `EFormatJpeg`
- `EFormatExif`
- `EFormatFbsBitmapColor4K`
- `EFormatFbsBitmapColor64K`
- `EFormatFbsBitmapColor16M`
- `EFormatFbsBitmapColor16MU`.

One device may not support all the formats above. You can check which formats are supported by a device from `TCameraInfo::iImageFormatsSupported`. It is a bit flag with type `CCamera::TFormat`.

The second parameter of `PrepareImageCaptureL()` is the index of the image size. It must be in the range of 0 to (`TCameraInfo::iNumImageSizesSupported -1`). You can get all supported image sizes by calling `CCamera::EnumerateCaptureSizes()`.

For the sake of simplicity, the discussion in this book focuses only on JPEG (Joint Photographic Expert Group) and EXIF (Exchangeable Image File Format) formats. The EXIF format is a specification of an image file format used by digital cameras that contains additional metadata tags, such as date and time information, aperture, shutter speed, ISO speed and many more. It is supported by JPEG and some other formats. It is not supported by JPEG 2000, PNG and GIF though.

> **What may go wrong when you do this:** Before using a format, you have to make sure that it is supported by the device. For example, most S60 devices do not support `EFormatJpeg`. They support `EFormatExif`.

In order to keep the recipe simple, it uses the synchronous overload of `RFile::Write()` to save the captured image. In production code, we recommend that you use the asynchronous method, called using an active object, to keep the GUI responsive.

4.7.3.5 Record Video

Amount of time required: 30 minutes
Location of example code: `\Multimedia\VideoRecording`
Required libraries: `mediaclientvideo.lib`
Required header file(s): `videorecorder.h`
Required platform security capability(s): `UserEnvironment`

Problem: You want to record a video clip from a camera and save it to a file.

Solution: Before you continue reading this recipe, make sure that you already know how to use the `CCamera` class (see Recipe 4.7.3.2).

The class from MMF that is used to record video is `CVideoRecorderUtility`. As for many other MMF APIs, there is an observer class: `MVideoRecorderUtilityObserver`.

The `CVideoRecorderUtility` class requires `UserEnvironment` capability. There are some methods that require `MultimediaDD`, but they will not be discussed in this book.

> **What may go wrong when you do this:** There is a known issue on some devices, which require that applications that perform video recording possess the `MultimediaDD` capability. A good example of this is the Nokia N93 and on early firmware of the Nokia N93i. Please check the Forum Nokia Technical Library at ***www.forum.nokia.com*** for more information.

Using `CVideoRecorderUtility` is similar to `CMdaAudio-RecorderUtility`, although it is a little more complex. Here are the steps required to record a video:

- Display the viewfinder using `CCamera` class (see Recipe 4.7.3.2).
- Create an observer that implements `MVideoRecorderUtility-Observer`.
- Create a new instance of `CVideoRecorderUtility`.
- Open the file by calling `CVideoRecorderUtility::OpenFile-L()`.
- Wait until the observer method, `MVideoRecorderUtility-Observer::MvruoOpenComplete()`, is called.
- Set some configurations, such as audio type.
- Call `CVideoRecorderUtility::Prepare()` to prepare for video recording.
- Wait until the observer method, `MVideoRecorderUtility-Observer::MvruoPrepareComplete()`, is called.
- Call `CVideoRecorderUtility::Record()` to start recording.
- Stop the recording by calling `CVideoRecorderUtility::Stop()`.

In our recipe, the `CSimpleVideoRecorder` class implements `MVideoRecorderUtilityObserver`.

Discussion: First, let's take a look at the usage of `CVideoRecorder-Utility::OpenFileL()`. This method requires several parameters, as shown below:

```
IMPORT_C void OpenFileL(const TDesC& aFileName, TInt aCameraHandle,
                            TUid aControllerUid, TUid aVideoFormat,
                               const TDesC8& aVideoType=KNullDesC8,
                          TFourCC aAudioType = KMMFFourCCCodeNULL);
```

The first parameter, `aFileName`, is the filename to which the video clip is saved.

The `aCameraHandle` parameter is the handle to the camera to use for recording. You can get the handle of the camera from the `CCamera::Handle()` method.

The `aControllerUid` parameter is the UID of the controller to use for recording. Phone manufacturers provide a controller plug-in to the

MMF, and you must specify the UID of the video controllers in this parameter. This is different from audio recording, where you don't need to specify which controller because the audio recorder utility class selects the controller automatically based on the file extension.

The `aVideoFormat` parameter is the UID of the video format to record to, and it depends on the controller plug-in. If you specify a video format that is not supported by the controller, you will receive an error.

The `aVideoType` parameter is the descriptor containing the video MIME type. There is a difference in how you use this parameter on UIQ and S60 devices. It is required on UIQ devices, but not required on S60 devices. If you forget to specify it on UIQ devices, you will get an error code 103.

The `aAudioType` parameter is the FourCC representing the audio format for recording. The same as for the video type, this parameter is mandatory on UIQ devices but it is not on S60 devices. If you don't specify this parameter on UIQ devices, the audio will not be recorded, and you will record video without audio.

What may go wrong when you do this: Not all methods in `CVideoRecorderUtility` are supported by all Symbian OS devices. For example, `CVideoRecorderUtility::OpenDesL()`, `CVideoRecorderUtility::OpenFileL()` and `CVideoRecorderUtility::OpenUrlL()` are not supported in S60 devices as at the time of writing.

How to select a video controller plug-in

The `CMMFControllerPluginSelectionParameters::ListImplementationsL()` method can be used to retrieve all controller plug-ins that match our search criteria. There are several possible criteria, but we will discuss two of them only: format and media IDs.

The first criterion is the required format support. You can set this by calling `CMMFControllerPluginSelectionParameters::SetRequiredRecordFormatSupportL()`, which takes a `CMMFFormatSelectionParameters` parameter.

Note that you can use a full filename as a parameter for `CMMFFormatSelectionParameters::SetMatchToFileNameL()` as it uses `TParse` (see Recipe 4.1.2.1) for an example of how to use this class.

The next criterion is the media IDs that must be supported by the plug-in, for example audio or video. It is set by calling `CMMFControllerPluginSelectionParameters::SetMediaIdsL()`.

The first parameter is an array of media IDs that the selected plug-ins must support. The second parameter, `aMatchType`, is the type of match to be made. There are three possible values: