

---

# **Symbian OS: Multimedia Framework And Other Multimedia APIs**

**Version 1.0**  
May 10, 2005

S  
Y  
M  
B  
I  
A  
N  
O  
S

## Legal Notice

Copyright © 2005 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>Overview .....</b>	<b>5</b>
<b>2.</b>	<b>MMF Architecture .....</b>	<b>6</b>
2.1	Client-Side Plug-in API Layer.....	7
2.1.1	Controller loading API.....	7
2.1.2	Managing sources and sinks API .....	8
2.1.3	Changing controller state API.....	9
2.1.4	Adjusting play-head position API.....	9
2.1.5	Retrieving metadata API.....	9
2.1.6	Handling custom commands API .....	10
2.1.7	Event-monitoring API.....	10
2.2	MMF Controller Framework .....	11
2.3	Controller Plug-in Resolver .....	12
2.4	Controller Proxy .....	13
2.5	Controller API.....	13
2.6	Custom Commands .....	14
2.7	Plug-in Architecture.....	16
2.8	Controller.....	16
2.9	Data Source, Data Sink, Data Path .....	16
2.10	Format Objects.....	18
2.11	Codecs .....	19
2.12	Sound Device .....	19
2.13	Audio Policy .....	20
2.14	Hardware Device API.....	20
<b>3.</b>	<b>Other Multimedia APIs .....</b>	<b>22</b>
3.1	The Font and Bitmap Server.....	22
3.1.1	Graphics devices and graphics context.....	22
3.1.2	Color and display mode.....	23
3.2	Window Server.....	24
3.3	Image Conversion Library .....	25
3.4	Bitmap Transform Library.....	26
<b>4.</b>	<b>Summary .....</b>	<b>28</b>
<b>5.</b>	<b>References .....</b>	<b>29</b>
<b>6.</b>	<b>Evaluate This Document.....</b>	<b>30</b>

## Change History

May 10, 2005	Version 1.0	Initial document release

---

## 1. Overview

The challenge of deploying rich multimedia capabilities on 2.5G and 3G wireless handheld devices is being met by two complementary technologies: dual-core processor architectures, which combine a CPU with a digital signal processor (DSP), and the robust, extensible, and flexible software architecture provided in Symbian OS v7.0s and later versions.

Early Symbian OS releases (Symbian OS v6.0 to v7.0) used a media server for multimedia applications. This original media server had a number of shortcomings when it came to handling the demands of high-performance multimedia applications such as streaming video, CD-quality audio, mobile commerce, or location-based services. Among the reasons why the media server was not an optimal solution: it operated in a single thread with multiple active objects running in the server process; there was a lack of real-time streaming support; there were difficulties in processing multiple asynchronous requests arriving in rapid series; and there were bottlenecks when processing a heavy data load.

Starting with Symbian OS v7.0s, Symbian introduced the multimedia framework (MMF), which has been redesigned from the ground up for mobile media, with powerful enhancements such as multiple threads, format recognition, streaming, and a plug-in media component library. With new and complete base media classes wrapped in the controller framework, licensees and third-party developers can now undertake far more effective multimedia application development for Symbian OS.

The MMF relies heavily on the controller framework, which provides support for multimedia plug-ins. In addition to optimal use of shared system resources, plug-ins allow run-time additions to be made to the built-in functionality, which means greater extensibility by creating support for a wide range of plug-in implementations. This translates into a highly flexible ability to implement specialized proprietary multimedia components, by both licensees and third-party developers.

The MMF does not, however, support still-image processing. Still-image and camera viewfinder functionality are handled by a separate Symbian OS component, the image conversion library (ICL). Video capturing (recording), despite effectively being the rapid capture and processing of multiple still images, is typically implemented using an MMF camcorder plug-in, which drives the necessary codecs and controls transfer of the video stream between sources and the sinks. Such implementation is chosen to extend the ability of the camcorder application to use the pool of available codecs via the MMF.

---

## 2. MMF Architecture

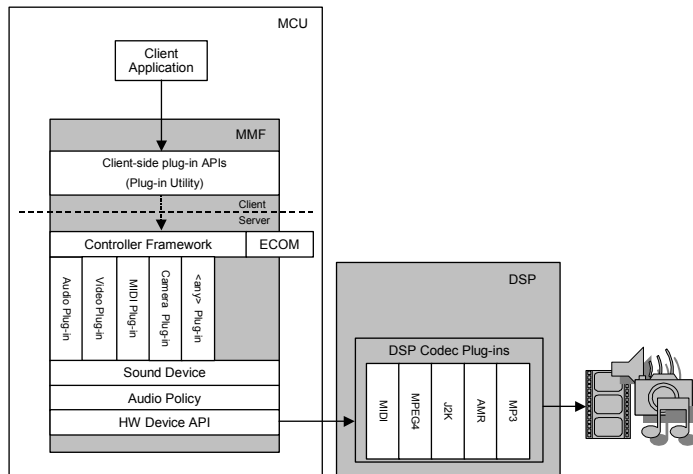
The MMF provides multimedia services to applications, acts as a repository for plug-in multimedia processing units, and serves as the generic interface to a device's hardware. The key element of the MMF is the controller framework, which manages selection, loading, and interaction with the multimedia plug-ins.

The most important change between the media server and the MMF component is the addition of multithreading. While the media server carried all multimedia processing within a single thread of the server process using multiple active objects, the MMF creates one client-side and one server-side thread for each plug-in and attaches them to the client's process. More threads may also be added when various pluggable MMF components such as formats, codecs, and data sources or sinks are used. Having all MMF threads run in the same process reduces context switching between processes while servicing requests, making the MMF more efficient — interthread communication (ITC) is used instead of interprocess communication (IPC).

The MMF relies on client/server architecture as a means of linking client-side plug-in APIs with the server-side controller plug-in interfaces. At any given time, the MMF will consist of numerous proxy server objects because the controller framework creates a proxy server in a new thread for each controller plug-in. Communication between the client and the server threads is supervised by the kernel utilizing ITC passing descriptor-packaged data represented by the `TMMFMessage` class, a secure wrapper class of `RMmfIpcMessage` derived from `RMessage`. Each message sent from a client contains an interface ID and a destination handle, based on which the controller framework can determine the server that should receive the message.

As shown in Figure 1, the MMF consists of the following basic layers:

- Client-side plug-in APIs, which are provided for applications to gain access to plug-ins by the implementation of plug-in utility objects.
- Controller framework that resolves selection and launching of plug-ins as well as facilitates message passing between applications and the plug-ins.
- Controller plug-ins responsible for data processing while moving data from sources to sinks.
- Sound device (DevSound) that provides a common interface to the audio hardware via hardware device interface.
- Audio policy, which resolves access priority of simultaneous client requests to use the sound device.
- Hardware device API that interfaces the low-level hardware devices including optional hardware-accelerated codecs residing on the DSP.



**Figure 1: The MMF architecture**

The following sections will briefly discuss each layer of the MMF architecture.

## 2.1 Client-Side Plug-in API Layer

The client-side plug-in API layer is represented in the MMF by a plug-in utility object. This object provides an application-level interface to the plug-in framework. Each client utility class has a nonstatic member variable, which refers to an instance of the `RMMFController` class. During plug-in loading a logical communication channel is created linking an instance of the `RMMFController` class with the `CMMFController` class on the server-side using Symbian client/server architecture. Service requests from the client to the server are sent using methods provided with the `RMMFControllerProxy` object: `SendSync()` for synchronous and `SendAsync()` for asynchronous requests.

Standard multimedia plug-in APIs can be divided into seven parts in accordance with their functionality:

- Controller loading and unloading
- Managing sources and sinks
- Changing controller state
- Adjusting play-head position
- Retrieving metadata
- Handling custom commands
- Event monitoring

### 2.1.1 Controller loading API

A call to the following method starts the loading of the plug-in controller in a new thread:

```
TInt RMMFController::Open(
    TUid aControllerUid,
    const TMMFPrioritySettings& aPrioritySettings);
```

An application can launch a plug-in by supplying its UID (if known). If not supplied by the application, the correct UID is determined by the controller framework, which is described in detail in the following sections.

The `TMMFPrioritySettings` parameter is used by the audio policy component to determine priority when access to the audio hardware is requested by multiple applications at the same time. Sound priority settings can be modified at any time after the plug-in is loaded by calling the following method:

```
TInt RMMFController::SetPrioritySettings(
    const TMMFPrioritySettings& aPrioritySettings) const;
```

Invoking the `RMMFController::Close()` method unloads the controller plug-in, deallocates all its resources, and terminates associated threads.

### 2.1.2 Managing sources and sinks API

After it is launched, the important role of a controller is to move data between the data source and data sink objects. The following APIs allow applications to add and remove sources and sinks:

```
// Add data source
TInt RMMFController::AddDataSource(
    TUid aSourceUid,
    const TDesC8& aSourceInitData);
TInt RMMFController::AddDataSource(
    TUid aSourceUid,
    const TDesC8& aSourceInitData, TMMFMessageDestination&
    aHandleInfo);

// Add data sink
TInt RMMFController::AddDataSink(
    TUid aSinkUid,
    const TDesC8& aSinkInitData);
TInt RMMFController::AddDataSink(
    TUid aSinkUid,
    const TDesC8& aSinkInitData, TMMFMessageDestination&
    aHandleInfo);

// Remove data source
TInt RMMFController::RemoveDataSource(
    const TMMFMessageDestination& aSourceHandleInfo);

// Remove data sink
TInt RMMFController::RemoveDataSink(
    const TMMFMessageDestination& aSinkHandleInfo);
```

As the code above illustrates, each `AddDataSource()` and `AddDataSink()` method has a basic and an overloaded version. Unlike their basic versions, the overloaded methods return the `TMMFMessageDestination` handle after creating data source and sink objects. The advantage of having a handle returned to the data object is that the data source or sink objects can be destroyed and replaced dynamically by the caller application at any time. This can be useful particularly when dealing with files played out of a play list. In this case, instead of bringing the entire controller plug-in environment down and recreating it between each file's playback, the old file's data source object can be de-referenced by the controller and substituted with the new one. Sources and sinks are discussed in great detail later in this chapter.

### 2.1.3 Changing controller state API

Each multimedia controller plug-in can be in one of three states at any given time: stopped, primed, or playing. The following APIs provide a state transition mechanism:

```
TInt RMMFController::Reset();
TInt RMMFController::Prime();
TInt RMMFController::Play();
TInt RMMFController::Pause();
TInt RMMFController::Stop();
```

The controller plug-in initiates in a stopped state, at which point its “play-head” position is set to zero and no buffers or any resources are allocated. When moving to the primed state, the controller allocates all the necessary resources and prepares the sources and sinks for sending and receiving data. To conserve memory, this should be a short, intermediate state before going to the playing state. Once in the playing state, the controller plug-in is actively transferring data between its sources and sinks.

The `Pause()` method puts the controller in the primed state while `Reset()` is used to wind back the controller plug-in to the initial stopped state after deallocating all of its resources.

### 2.1.4 Adjusting play-head position API

Adjusting the play-head position within a media source and returning the current position during playback is controlled by the following methods:

```
TInt RMMFController::GetPosition(
    TTimeIntervalMicroSeconds& aPosition) const;
TInt RMMFController::SetPosition(
    const TTimeIntervalMicroSeconds& aPosition) const;
```

Similarly, to find the duration of the media source, the `GetDuration()` method is used:

```
TInt RMMFController::GetDuration(
    TTimeIntervalMicroSeconds& aDuration) const;
```

The duration can be queried at any time after the data source and sink objects are instantiated. Adjusting the position can only be done while in the primed or playing state.

### 2.1.5 Retrieving metadata API

The standard API supports two basic methods that allow the retrieval of media metadata properties by the controller plug-in:

```
TInt RMMFController::GetNumberOfMetaDataEntries(
    TInt& aNumberOfEntries) const;
CMMFMetaDataEntry* RMMFController::GetMetaDataEntryL(
    TInt aIndex) const;
```

After retrieving the total number of metadata entries within a media, the calling application can loop through all the available metadata parameters within a media and retrieve each entry’s name and value in the `CMMFMetaDataEntry` container.

### 2.1.6 Handling custom commands API

With custom commands, applications are able to pass specific requests to the plug-ins that are not supported by the standard APIs. Application-specific commands can be passed to plug-ins synchronously and asynchronously and in both cases an overloaded version of the method is provided that allows for returning of the data back from the plug-in via the `aDataFrom` descriptor.

```
TInt RMMFController::CustomCommandSync (
    const TMMFMessageDestinationPckg& aDestination,
    TInt aFunction,
    const TDesC8& aDataTo1,
    const TDesC8& aDataTo2,
    TDes8& aDataFrom);

TInt RMMFController::CustomCommandSync (
    const TMMFMessageDestinationPckg& aDestination,
    TInt aFunction,
    const TDesC8& aDataTo1,
    const TDesC8& aDataTo2);

void RMMFController::CustomCommandAsync (
    const TMMFMessageDestinationPckg& aDestination, TInt
    aFunction,
    const TDesC8& aDataTo1,
    const TDesC8& aDataTo2,
    TDes8& aDataFrom,
    TRequestStatus& aStatus);

void RMMFController::CustomCommandAsync (
    const TMMFMessageDestinationPckg& aDestination, TInt
    aFunction,
    const TDesC8& aDataTo1,
    const TDesC8& aDataTo2,
    TRequestStatus& aStatus);
```

### 2.1.7 Event-monitoring API

A `CMMFControllerEventManager` class is used to listen for events from the controller plug-in. The class is based on an active object and signals the client via the `MMMFControllerEventManagerObserver` interface when an event occurs. Typically, controllers will generate events when detecting an error condition or notifying completion of asynchronous requests. For example, the `KMMFEventCategoryPlaybackComplete` event is generated when the controller stops playing due to reaching the end of a file.

Clients must register to receive events from the controller plug-in. They may also choose not to be notified of any controller-originated events by calling the `CancelReceiveEvents()` method.

```
void RMMFController::ReceiveEvents (
    TMMFEventPckg& aEventPckg,
    TRequestStatus& aStatus);

TInt RMMFController::CancelReceiveEvents ();
```

The events are returned via the `TMMFEvent` container, which consists of the event ID and an error code. It is packaged into the descriptor type `TMMFEventPckg` object because it crosses client/server boundary.

## 2.2 MMF Controller Framework

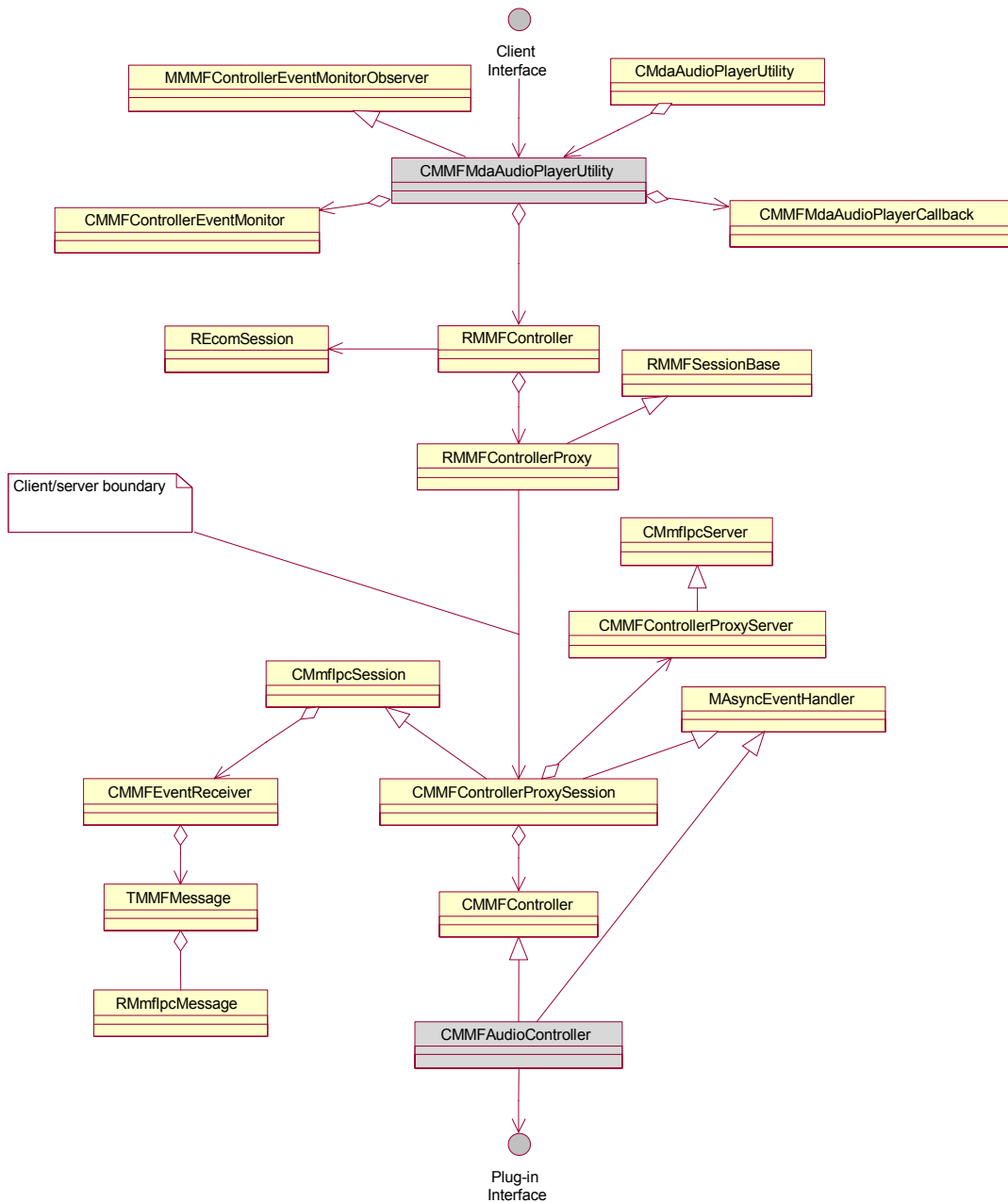
The controller framework provides support for specialized multimedia plug-ins called controller plug-ins. The main role of a controller plug-in is to coordinate the flow of data between the data sources and sinks. For example, an audio controller plug-in might take data from a file, the source, and output it to a speaker, the sink, for playback. Similarly, it could take data from a microphone source and save it into a file sink. A typical controller plug-in supports multiple media formats. For example, a single audio controller plug-in may support WAV, AMR, and MP3 audio formats. This is possible since each controller may employ multiple format and codec objects.

The controller framework consist of the following components:

- Controller plug-in resolver, which allows the system to match the most suitable plug-in for a particular data source.
- Controller proxy that handles plug-in thread allocation and interthread communication (ITC) using the Symbian OS client/server mechanism.
- Controller APIs, a set of concrete APIs used to communicate with plug-ins.
- Custom commands that allow the standard set of controller APIs to be extended.
- Miscellaneous helper classes, such as `CMMFBuffer`, that help with data buffering, metadata parsing, or formatting.

Figure 2 illustrates class associations of a complete MMF plug-in architecture for an audio plug-in. The `CMMFMdaAudioPlayerUtility` object is a concrete implementation of basic audio functionality found in `CMdaAudioPlayerUtility`, which owns `RMMFController` class APIs. (Although the examples in this section focus on the audio plug-in, all MMF plug-ins follow the same implementation mechanism.)

Calling `RMMFController::Close()` results in canceling of any outstanding requests, terminating proxy session with the server, and deallocating all of the controller plug-in resources.



**Figure 2: MMF controller framework with audio plug-in API**

### 2.3 Controller Plug-in Resolver

The controller plug-in resolver is an ECOM plug-in (the ECOM interface maintains a registry of all the supported plug-ins on Symbian OS). It is responsible for finding the appropriate plug-in for a particular data source format and returning its unique ID. Plug-in resolution is accomplished by extracting plug-in information from the ECOM registry. More specifically, the description data is read from the ECOM resource files associated with all of the available plug-ins on the system and then compared against the properties of the data source to find a positive match. The examined plug-in properties may include

plug-in supplier, media MIME type, file extension, etc. In addition, more exhaustive resolution would typically involve parsing of header data segments to match the first few bytes of multimedia data type to ensure the supported plug-in formats are capable of handling the data source.

Because there might be more than one plug-in available for the same data format on a device, an application can specify its preferred supplier of the plug-in. The resolver will then return an array of supported plug-ins that complement the application's criteria. Then it is up to the application to select the plug-in it wishes to use.

Some instructions about the use of plug-in resolver classes, along with example code, appear in the document *Series 60 Platform: Techniques To Deal With Lead- And Device-Specific Features in C++*.

## 2.4 Controller Proxy

When the correct plug-in for a specific data source is matched by the resolver, the next component of the controller framework, namely the controller proxy, handles allocation of the controller framework resources to set up the environment within which the plug-in will launch and live.

In the heart of the controller proxy is a thin Symbian OS client/server layer, which provides for thread creation, server startup, and a message-passing mechanism between the controller proxy client and the proxy server. As shown in Figure 2, the client side is represented by the `RMMFControllerProxy` object, derived from `RMMFSessionBase`, and the server side by the `CMMFControllerProxyServer` object. Both client and server objects run in separate threads owned by the application process. The server thread is responsible for creating and destroying just a single session (`CMMFControllerProxySession`) that is used to transmit messages from the application to the plug-in.

The `CMMFControllerProxySession` object is the receiver of messages sent from the client side. Upon receiving a message, the session object passes it to the `CMMFController` base class. Since each MMF message has an assigned interface ID and destination handle, it is possible for the `CMMFController` base object to determine whether the request can be handled at its level or should be passed on to the controller plug-in. This is determined in the `CMMFController::HandleRequestL()` method.

For example, if a message's interface ID equals `KUIdInterfaceMMFController`, the request is handled by the base controller object. This is due to the fact that the `CMMFController` object has a built-in capability for handling all standard API requests. On the other hand, all requests sent as custom commands would not have a matching interface ID for this controller, and therefore would be passed to the controller plug-in for handling — in the case of an audio plug-in, it would be the `CMMFAudioController` object.

Because each controller proxy server maintains only a single session, this means that only one application can be a client of a particular instance of a controller. All other clients wanting to use the same controller plug-in, as well as any client that wants to use the same controller more than once, will be given a second or subsequent copy of the plug-in in a new thread.

## 2.5 Controller API

The controller plug-in API is represented by the `CMMFController` interface. It is responsible for receiving client requests passed via the application's side APIs identified in the `RMMFController` class and then dispatching them further to the lower layers. When instantiated, `CMMFController` allocates all of the internal plug-in factory objects

(described in detail in sections to follow), such as data sources and sinks, format classes, and codecs. In addition, it establishes a session with the sound device. Every MMF plug-in is implemented as a derivation of `CMMFController` and must provide implementations to all of its virtual functions.

The declaration of the `CMMFController` class is found in `MMFController.h`, which is part of the SDK documentation. The APIs look very similar to those defined in the `RMMFController` class at the application interface level. In addition to many similarities, it's easy to spot a few differences. First, there is a lack of methods equivalent to `RMMFController::Open()` and `RMMFController::Close()`. This is due to the fact that `RMMFController::Open()` results in a call to `CMMFController::NewL()` with the resolved plug-in UID, and `RMMFController::Close()` causes execution of the `CMMFController` object's destructor.

Another important difference marks the existence of synchronous and asynchronous versions of the state-changing methods: `PlayL()`, `PauseL()`, and `StopL()`. The significance of this is such that these requests should be processed asynchronously in the controller while maintaining a synchronous client API. For example, the Play request must return once playing has commenced and not wait until playing is complete.

```
virtual void CMMFController::PlayL() = 0;
virtual void CMMFController::void PlayL(
    TMMFMessage& aMessage); //async

virtual void CMMFController::PauseL() = 0;
virtual void CMMFController::PauseL(
    TMMFMessage& aMessage); //async

virtual void CMMFController::StopL() = 0;
virtual void CMMFController::StopL(
    TMMFMessage& aMessage); //async
```

Custom command support at the controller API level is represented by the `CustomCommand()` handler. A method is also provided that allows adding of custom command parsers. Symbian provides default implementation for the custom command handler to ensure completion of all client/server messages even in the case of an unrecognized request. (Note: Symbian client/server requests must always complete. If the developer forgets to do that, synchronous requests will lock up the device, waiting indefinitely for the request to complete, and asynchronous requests will deplete resources by taking from the limited pool of message slots.)

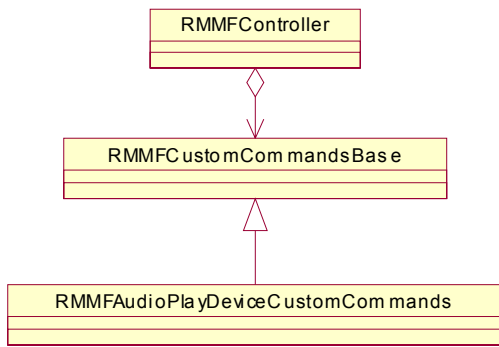
```
virtual void CMMFController::CustomCommand(
    TMMFMessage& aMessage)
    {aMessage.Complete(KErrNotSupported);}; //default implementation

void CMMFController::AddCustomCommandParserL(
    CMMFCustomCommandParserBase& aParser);
```

## 2.6 Custom Commands

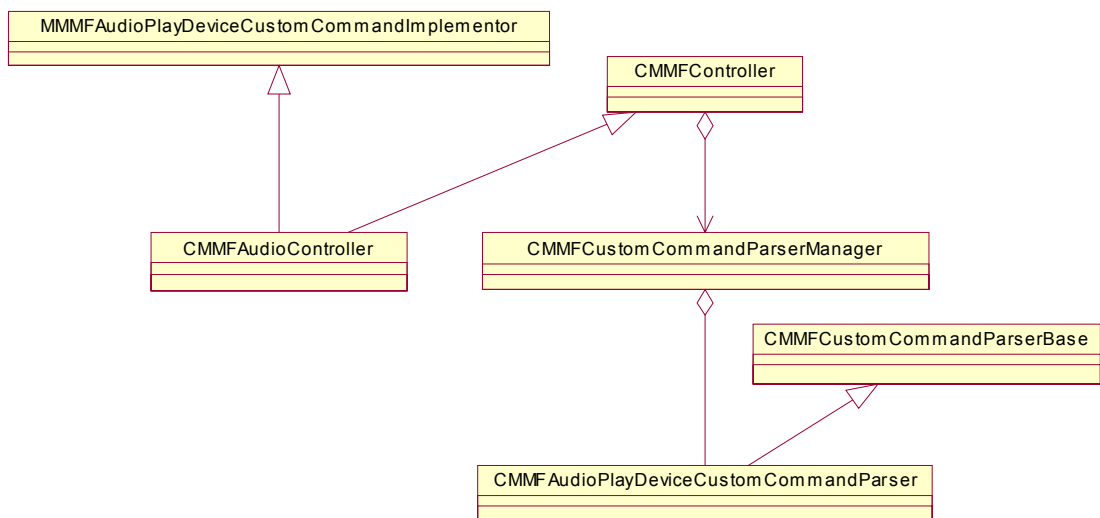
Because standard plug-in APIs provide limited functionality, the MMF comes with a set of custom command handlers to allow plug-in implementers the flexibility of adding controller-specific APIs without changing of the base classes. Figure 3 and Figure 4 show custom command handler implementations on both client and server sides.

Messages sent via custom commands can be either synchronous or asynchronous and are treated differently from standard APIs. As mentioned earlier, the controller framework has a built-in knowledge of how to interpret and manage messages sent from the applications via standard APIs. In case of messages sent via custom commands, the responsibility of deciphering and acting upon them is passed on to the controller plug-in, which would typically use the help of a custom command parser.



**Figure 3: Custom command handlers on the client side**

On the server side, the `CMMFCustomCommandParserManager` object is owned by the controller framework. It knows about all of the custom command parsers derived from `CMMFCustomCommandParserBase` for the particular controller plug-in. When a custom command is sent to the controller plug-in, the `CMMFController` class checks with its `CMMFCustomCommandParserManager` object to see whether it has any parsers available to handle the command. If it does not have one, the message is dispatched directly to the controller plug-in for processing — to the `CMMFAudioController` class in Figure 4.



**Figure 4: Custom command handler for the audio controller plug-in with the sample parser on the server side**

As shown in Figure 4, the `CMMFAudioController` plug-in uses a custom command parser — `CMMFAudioPlayDeviceCustomCommandParser` — to handle the playback of common audio formats. By deriving from the `CMMFAudioPlayDeviceCustomCommandParser` mixin class, `CMMFAudioController` extends its ability to handle special audio requests, such as setting or getting current audio volume, balance, and maximum allowed volume, and then pass them on to the sound device for execution.

Any special request not supported by the standard APIs or the existing custom command parsers would require either adding a new parser or overriding the `CMMFController::CustomCommand()` handler inside the controller plug-in implementation.

## 2.7 Plug-in Architecture

Software plug-ins within the MMF are referred to as controller plug-ins (as opposed to the hardware-accelerated plug-ins running on the DSP). Designing an MMF plug-in can range from a very simple to a fairly complex task depending on the scope of the desired plug-in framework capabilities. As mentioned earlier, the MMF has been designed to provide flexibility to the developer. The MMF plug-in architecture itself is a configurable framework that consists of several “pluggable” components. Consequently, licensees and third-party developers can enhance the multimedia framework by providing their own controller classes, formats, codecs, sources, and sinks.

The minimum required component for an MMF plug-in is the controller. The controller provides the client with an interface to the device’s hardware environment and to other (optional) plug-ins. Having said that, depending on the implementation any plug-in may consist of all or some of the following components:

- Plug-in controller (mandatory)
- Data path, data source, and data sink
- Format decoder and encoder
- Codec

The framework takes care of automatic instantiation of all the necessary plug-in components to process data from the data sources upon creation of the plug-in controller.

The class association of all the plug-in components is shown in Figure 2.

## 2.8 Controller

The plug-in controller is a concrete implementation of the `CMMFController` interface discussed earlier in Section 2.5, “Controller API.” It is responsible for controlling the flow of data between one or more data sources, transforming data into a different format, and then passing it to one or more data sinks. It is contained within a controller framework proxy session associated with a proxy server, each running in a separate thread of the application process.

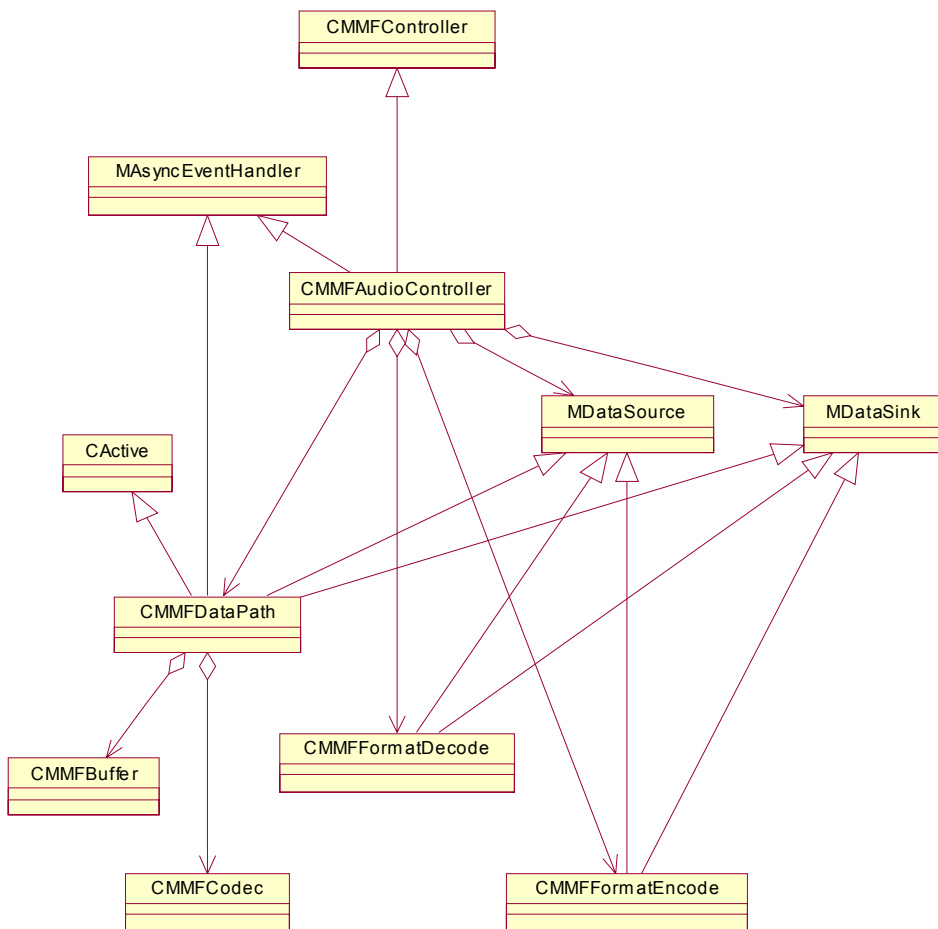
A standard audio plug-in controller is represented by the `CMMFAudioController` class (see Figure 3). The factory functions in this class hide the ECOM interface that resolves the data that is passed in. The controller can be implemented as a single complete plug-in component managing all aspects of data conversion and transport to the lower layers. A much better design, however, involves employing highly specialized MMF components, such as data sources, data sinks, data path, formats, and codecs, which are also ECOM plug-ins. The ultimate goal when developing a controller plug-in should be to aim for flexible design with a single controller supporting multiple types of data formats including automatic future expansions with no or very minimal additional coding.

## 2.9 Data Source, Data Sink, Data Path

Data sources and sinks are software manifestations of input and output devices such as files, data streams, communication ports, speakers, microphones, cameras, and displays. They are ECOM plug-ins intended to support MMF controller plug-ins. They are created and owned by the controller framework and their references are passed into the controller plug-in by using the `AddDataSource()` and `AddDataSink()` APIs. Because they are not owned by the controller plug-in, they should never be deleted by the controller plug-in. Instead, a client application should invoke `RemoveDataSourceL()` or

`RemoveDataSinkL()` to remove the reference and allow the controller framework to delete a particular source or sink object.

Data source and data sink are represented by the `MDataSource` and `MDataSink` mixin classes, respectively. Both `MDataSource` and `MDataSink` abstract the concept of a supplier and consumer of data. Each class contains methods that can validate whether data comes from or is being sent to the correct type of source or sink by examining its respective FourCC code. (FourCC, or the Four Character Code standard, is a four-byte code stored in the header of files containing multimedia data including images, sound, or video.)



**Figure 5: Class diagram for a common MMF audio plug-in**

The following are the minimum required data sources and sinks: `CMMFFile`, `CMMFDescriptor`, `CMMFUrlSource`, `CMMFUrlSink`, `CMMFAudioInput`, and `CMMFAudioOutput`. The `CMMFFile` and `CMMFDescriptor` objects can be used as either sources (when playing) or sinks (when recording).

To better control the transport of data between plug-in components, the MMF provides a helper class, `CMMFDataPath`, which derives from both `MDataSource` and `MDataSink`. `CMMFDataPath` is a base class for objects coordinating the flow of data between the source and destination buffers, using a codec when necessary. Including data path in the plug-in design removes much of the data-processing complexity from the controller. Since each data path object can run in a separate thread, the performance of the controller servicing multiple sources and sinks (for instance in the case of a video

controller plug-in, which deals with audio and video tracks) benefits a great deal from having defined several data path objects.

When the received data in the source buffer requires formatting, before sending it to the sink, `CMMFDataPath` will use one of the two available format classes: `CMMFFormatDecode` or `CMMFFormatEncode`. If the sink expects a different data type from the one supplied by the data path after formatting, it can use a software codec, derived from `CMMFCodec` (described later in Section 2.11, “Codecs”). For instance, when the source is a compressed audio file, such as MP3, and the hardware sink is a PCM audio DAC module (D/A converter), which expects an uncompressed 16-bit PCM signal, a specialized MP3 decoder is needed. On the other hand, when the source and the sink data types are the same, no codec is required and the data path will use a so-called “null codec,” which will simply copy data straight from the source to the sink data buffers.

When using a hardware-accelerated codec implemented on the DSP, the data from the source is moved by `CMMFDataPath` directly to the hardware device layer, which ships it to the DSP for decoding. (DSP codecs are beyond the scope of this document.)

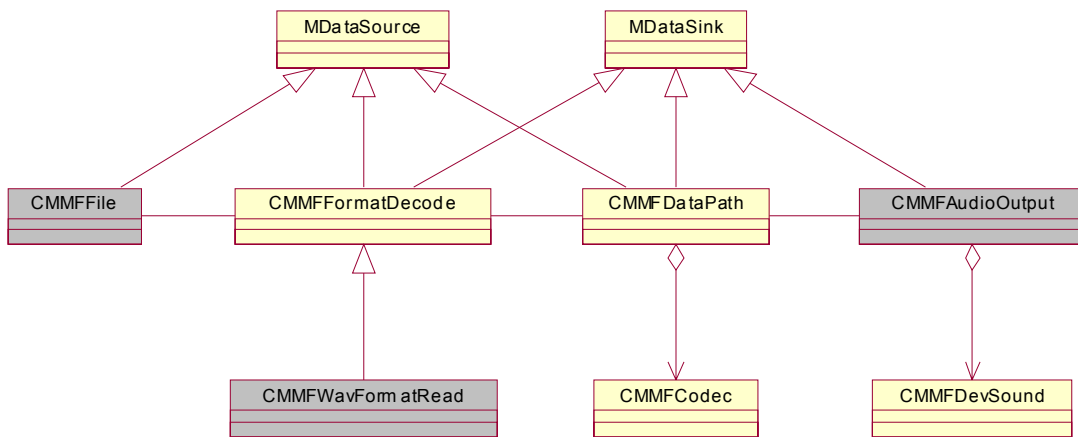
## 2.10 Format Objects

Format objects are ECOM plug-ins capable of resolving data formats of the media sources and sinks. They can effectively abstract code specific to format recognition out of the controller and allow a single controller plug-in to support multiple data formats. By using format plug-ins, a controller plug-in can dynamically extended its ability to support different data formats.

Format plug-ins work as interpreters of source and sink data formats for the data path. It is important to understand that the formats are different from codecs or data types. A format may contain one or more data types — for example, the WAV format is a container for a number of different data types, such as PCM, ADPCM, GSM6.10, Dolby AC-2, MPEG-1, etc. Unlike codecs that perform algorithm-based data compression and decompression of particular data types, the objective of format objects is unpacking and packing of the data, parsing to strip headers, locating actual payload within media source containers, performing demultiplexing of audio and video streams for playback (or multiplexing when video recording), and retrieving media-specific metadata properties, such as number of channels, clip duration, bit rate, sample rate, and frame size.

There are two types of format objects: format decoder and format encoder. The format decoder embodied by the `CMMFFormatDecode` class is responsible for decoding the format of the data from the source before it can be sent to the sink by the data path (for example, for playback). The format encoder object is based on the `CMMFFormatDecode` class and performs data encoding (for example, while recording).

Figure 6 illustrates a sample scenario of a WAV file played. `CMMFFile` represents WAV file data source. The `CMMFAudioOutput` object abstracts an interface to the sound device sink. `CMMFDataPath` coordinates the flow of data between the file source and the lower-level audio output sink with the help of the `CMMFWavFormatRead` format object. If the WAV file contained any encapsulated compressed data types, such as MPEG-1, the data path would load a specific `CMMFCodec`-based class to convert it to the type that can be handled by the sound hardware of the device (for example, PCM16).



**Figure 6: Relationship of format objects to data path, sources, sinks, and codecs**

## 2.11 Codecs

Codecs perform conversions between different compressed data types supported by sources and sinks of multimedia data. There are two types of codecs: MMF software codecs and hardware-accelerated (DSP-based) codecs. Software codecs are part of the MMF controller framework and implemented as standard ECOM plug-ins. Use of DSP codecs has the advantage of increased processing speed while lowering the power consumption of a device. DSP-based codecs are commonly used for A/V encoding and decoding on OMAP processors.

Software codecs are derived from the `CMMFCodec` class and operate in their own threads. They convert source data in a certain FourCC coding type to a destination buffer of another FourCC coding type usually via `CMMFDataPath` or `CMMFDataPathProxy`. For example, if the source is an H.263 video stream, the codec will uncompress and convert it to YUV/RGB data type. After the conversion, the data path will control the synchronization of the audio and video signals and ship them to the appropriate sinks — video to be rendered on the display and audio played through the speaker.

Support of hardware-accelerated codecs in the MMF relies upon the `CMMFHwDevice` interface. Loading and managing of hardware codecs is beyond the controller's competency, and therefore it is left out to the lower layers, such as `CMMFDevSound` and `CMMFHwDevice`, to control them. Unlike software codecs, hardware-accelerated codecs provide an asynchronous interface for data transformation. Typically, they would be equipped with direct hooks to the hardware offering better performance and more flexibility to the third-party developers.

## 2.12 Sound Device

Sound device, often referred to as a DevSound, provides the link between the MMF and the audio hardware via a common hardware device API. Some of the main responsibilities of the DevSound include providing applications with access to available audio resources, preparing hardware for audio activity, selecting and loading of hardware-accelerated codecs, adjusting volume, and assisting in audio playback and recording.

Each application that is using audio resources on the device through the MMF has an instance of the `CMMFDevSound` class associated with it and is subjected to the audio

policy rules managed by the `RMMFAudioPolicyProxy` object. The audio policy component determines which instance of `CMMFDevSound` should be allowed access to the audio hardware when more than one application requests simultaneous access to the audio hardware. In addition to satisfying audio policy rules, before allowing an application to play any audio, the `DevSound` must also query the Symbian device management server (DMS) to acquire permission to use hardware resources. Once the audio path is established, the `DevSound` passes audio data for playback from the data path through the hardware device layer (`CMMFHwDevice`), or, as in the case of DTMF tones, requests the tone player to generate and play simple tones.

It is worth mentioning that the `DevSound` can also be accessed directly by the applications instead of via the MMF. This use is mainly for games or similar applications where direct access to the device's sound features is required.

### 2.13 Audio Policy

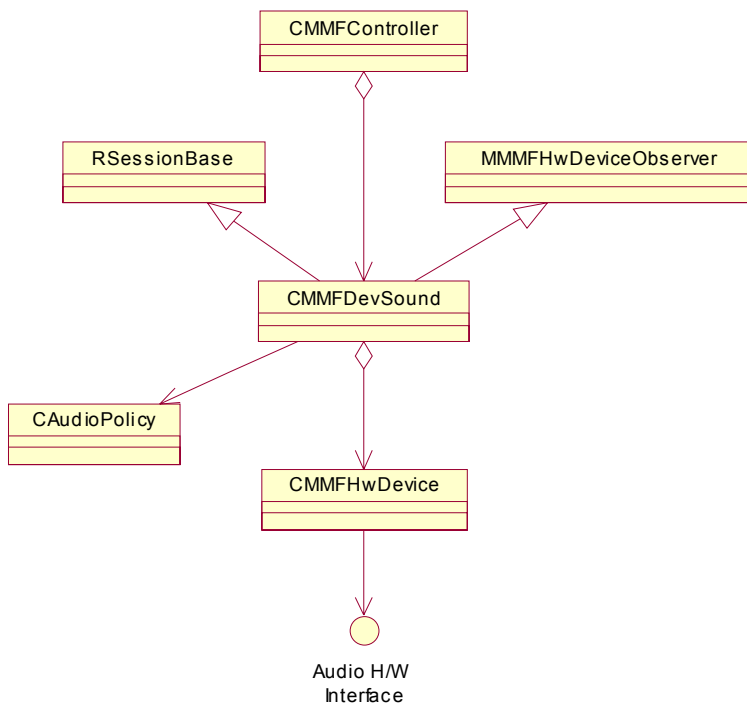
The audio policy component manages access priority to the audio hardware on the device by resolving simultaneous requests from multiple instances of the `CMMFDevSound` object. When multiple applications make simultaneous audio playback requests, audio policy decides which should be permitted to play, which is denied the access to the audio resources, or whether it is possible to mix two or more sounds.

Because the audio policy rules may differ considerably for various products, the implementers of the audio policy components must provide custom versions for each product. There are a few design aspects that need to be considered when designing the custom audio policy component. Different requirements for the product behavior, capability of the hardware, and availability of system resources will determine the adequate audio policy implementation. In any case, the minimum requirement is having audio priority of the telephony application always set to a higher priority than any other multimedia application. This imposes the necessity for the audio policy component to constantly monitor the call status and allow a ring tone to be played during an incoming call, even if it means preempting an ongoing audio playback.

The audio policy should also monitor changes of the profile settings and update attributes accordingly in the `CAudioProfileSettings` class. Audio attributes from this class are retrieved from `CMMFDevSound` and used for audio processing. It is important for the audio policy to be aware of special audio preferences assigned to certain applications, such as allowing a clock alarm ring even when the phone is in a "silent" mode.

### 2.14 Hardware Device API

The MMF interface to the hardware components is provided via the outermost layer — the hardware device API. This interface is implemented as an ECOM plug-in and is responsible for handling the flow of data between the MMF and the hardware components on the device. It effectively abstracts the device's hardware components, such as speaker, microphone, display, or a secondary processor used on the device. Figure 7 shows an overview of all the low-level MMF components in relation to each other and to the controller plug-in.



**Figure 7: High-level overview of the low-level MMF layers**

Devices that run on dual-core processor architectures, such as OMAP, use this layer to interface the pool of DSP-based algorithms and codecs. Each DSP-based codec or algorithm that is controlled from the MMF must have an associated object derived from the `CMMFhwDevice` base class to provide a gateway to the DSP resources. For instance, a third-party MPEG-4 decoder plug-in running on the DSP requires a `CMMFhwDevice`-based object registered with the ECOM interface so it can be accessible via the MMF. This object will then handle initialization of tasks on the DSP and perform routing of data transcoded by the hardware codec to other sinks if necessary.

## 3. Other Multimedia APIs

In addition to the Multimedia Framework API described in the preceding sections, there are also related servers and libraries that are commonly used in multimedia applications. These servers include the font and bitmap server and the window server. The font and bitmap server handles fonts and bitmaps in memory while the window server handles application input and output resources, such as screen, pointer, and keypad. In addition, there is also the image conversion library (ICL), which converts any image format to Symbian OS bitmap format and vice versa. Finally, there is the bitmap transform library, which provides additional handling for bitmap objects such as image rotation or image scaling. This section provides a brief overview of these servers and libraries from the perspective of multimedia application development.

### 3.1 The Font and Bitmap Server

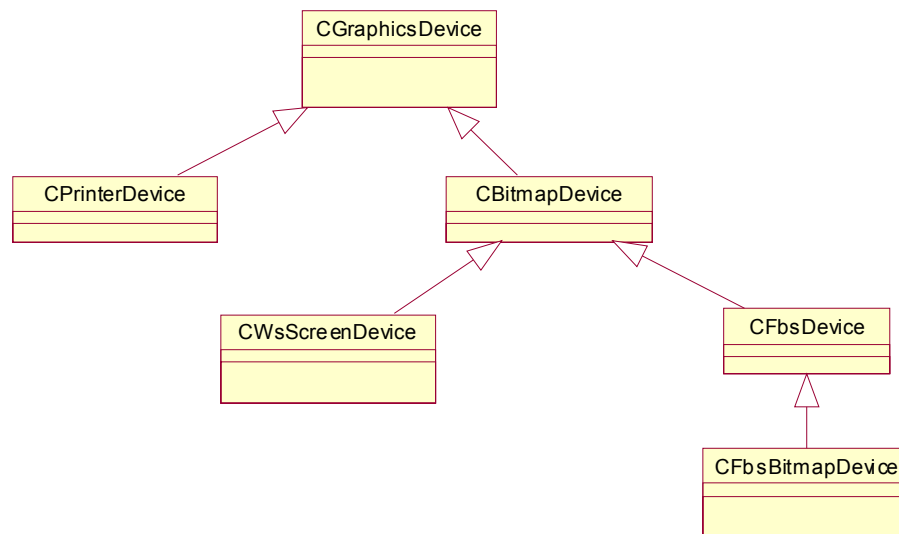
The font and bitmap server is used to handle displaying text and bitmaps on screen efficiently. The font and bitmap server also provides a mechanism to create and manage off-screen bitmaps. Off-screen bitmaps, also known as double buffering, are very useful in applications that implement animation where a flicker-free redrawing is a requirement. Off-screen bitmaps are created using `CFbsBitmapDevice`, `CFbsBitGc`, and `CFbsBitmap` together.

In order to display any image object, the object must first be converted into a bitmap object. The most common bitmap classes are `CFbsBitmap` and `WsBitmap`. Both classes provide the necessary APIs to handle displaying and managing bitmaps on screen. The `CFbsBitmap` class is managed by the font and bitmap server while the `CWsBitmap` class is handled by the window server.

A full discussion of services provided by the font and bitmap server is beyond the scope of this chapter, which only covers the basic concepts used when dealing with graphics and bitmaps.

#### 3.1.1 Graphics devices and graphics context

A graphics device represents a drawing surface. This surface can be a screen, a printer, an off-screen bitmap, or any similar item. The graphics device is defined by two basic attributes, its size and color depth. Figure 8 shows the graphics devices classes.



**Figure 8: The graphics devices classes**

A graphics context provides the means of drawing to a certain graphics device. Graphics context is created using the `CreateGraphicsContext()` method.

The base class for graphics context is `CGraphicsContext`. This is an abstract class. Concrete devices provide the actual implementation by deriving their own classes from this base class. A graphics context provides an extensive set of drawing operations for drawing various types of shapes (rectangles, lines, polygons, ellipses, rounded rectangles, ovals, etc.), text, and bitmaps. These drawing operations are accompanied with drawing tools such as:

- A pen for drawing shapes outlines and lines,
- A brush for area filling with a specified color and texture,
- A font for drawing text with specified size, color, and style.

### 3.1.2 Color and display mode

Color is represented in 32 bits in Symbian OS. Eight bits are used to represent each primary color (red, green, and blue) and eight bits are unused. This is the maximum number of bits used to represent a certain color; other color representations with fewer than eight bits also exist. Those different representations are referred to by the display mode. Each display mode specifies the current color depth used for drawing to active display.

The most common system-defined display modes are listed in Table 1.

Display Mode	Description
Egray2	Monochrome display mode (1 bit per pixel [bpp])
Egray4	4 grayscales display mode (2 bpp)
Egray16	16 grayscales display mode (4 bpp)
Egray256	256 grayscales display mode (8 bpp)

Display Mode	Description
Ecolor16	Low-color EGA 16-color display mode (4 bpp)
Ecolor256	256-color display mode (8 bpp)
Ecolor64K	64,000-color display mode (16 bpp)
Ecolor16M	True-color display mode (24 bpp)
Ecolor4K	4,096-color display (12 bpp)

**Table 1: Display mode types**

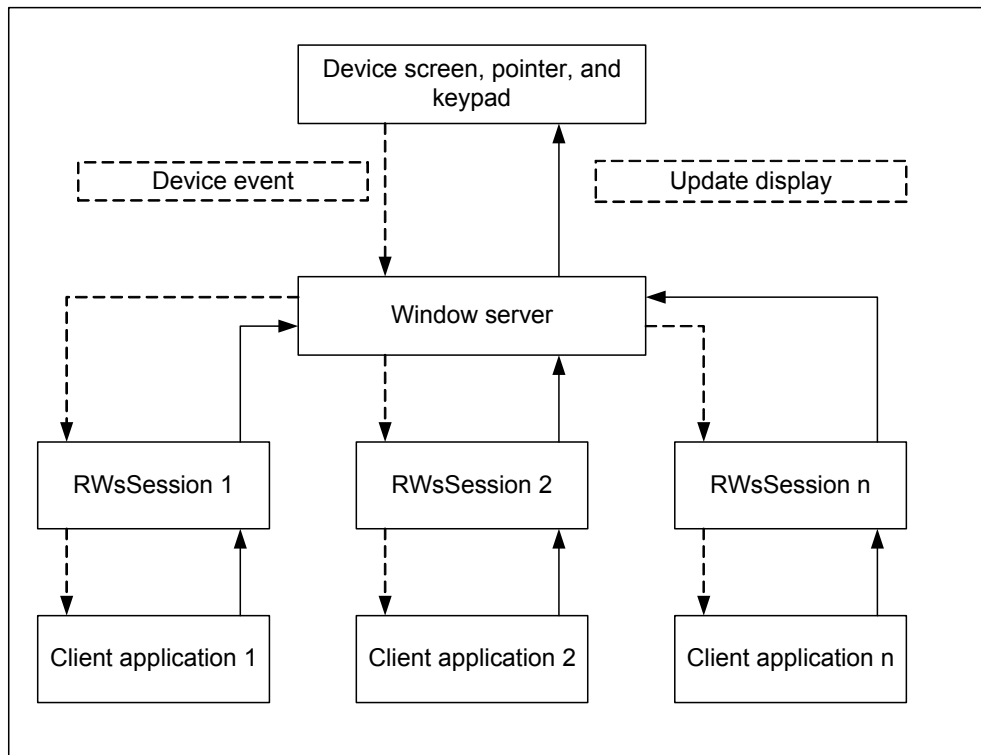
### 3.2 Window Server

The window server enables applications to interact with the device's screen and its input mechanisms, such as a keypad or a pointer. As a result, any application with a user interface should use the window server. The window server decides which application should receive user input and have access to the device screen, based on client application requests to use the screen.

A window object represents a screen area that an application can draw to. Applications redraw using the `RWindow` class, which represents a handle to the standard window, derived from the abstract base class `RdrawableWindow`.

As shown in Figure 9, each application connects to the window server using `RWsSession`. Through this connection, the application can access the display and receive display-related events. There is always one active session at a time, which belongs to the application in active focus. From a development perspective an application does not deal with `RWsSession` directly. Instead, the application receives a drawing request to its higher-level method, the `Draw()` method.

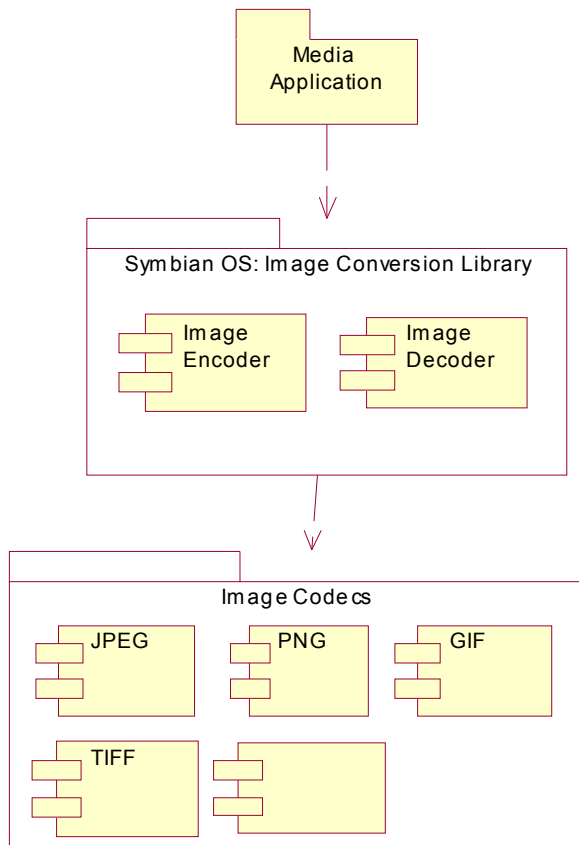
Although the window server and the font and bitmap server are two different entities, both provide a Bitmap support API. As discussed in the previous section, the window server provides the `CWsBitmap` class, which is used as an alternative to `CFbsBitmap` to manage bitmap objects. Functions with `CWsBitmap` objects are faster to display than those with `CFbsBitmap` objects. For off-screen bitmaps, which are prepared by the font and bitmap server, the window server classes are needed to display those off-screen bitmaps to the screen. In addition, the construction of `RFbsSession` takes place when `RWsSession` is created, which means that all window server clients are also font and bitmap server clients.



**Figure 9: Flow control between applications and the window server**

### 3.3 Image Conversion Library

The ICL is used to convert standard image format files to Symbian bitmap format as well as convert bitmap images into these standard formats. Examples of standard image formats include JPEG, GIF, and TIFF. The use of different formats in storing and displaying images is due to the conflicting performance requirements of image display and image storage. The process of converting any nonbitmap formats to bitmap format is called image decoding. Conversely, image encoding is used to convert a bitmap object into another format.

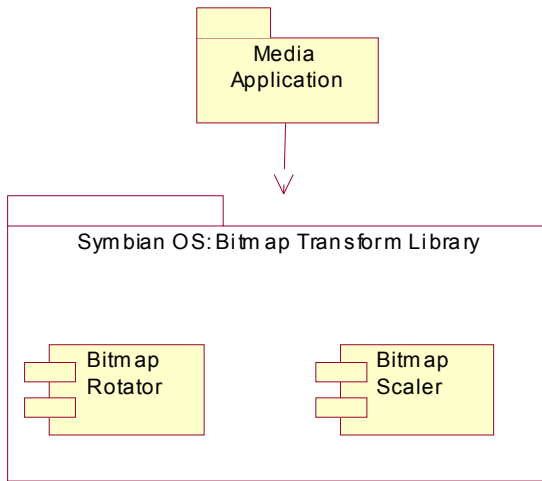


**Figure 10: Image conversion library**

The Image Conversion Library API contains the interfaces necessary to encode and decode image objects. In addition, many image files include embedded information, which may define features such as image resolution, image format, or number of image frames in the file, all of which could be useful to the client application. This information can be read using the additional methods in the Image Conversion API. As shown in Figure 10, the ICL also abstracts client applications for interacting directly with image codecs that exist in the platform. Those image codecs could be changed from one platform or implementation to another. This abstraction allows new image codecs to be added without any code changes in the client application.

### 3.4 Bitmap Transform Library

The bitmap transform library provides additional features for handling bitmaps. As shown in Figure 11, the main supported features include bitmap scaling and bitmap rotation. By combining features that exist in both the image conversion library and bitmap transform library, a client application can open an image from a file, modify it, and save the modified image into a file again.



**Figure 11: Symbian bitmap transform library subsystems**

---

## 4. Summary

As demonstrated in this document, the MMF offers a very robust and versatile architecture. It fully meets the requirements necessitated by the demands of high-performance media-centric Symbian OS applications performing audio and video playback and recording or processing of concurrent multiple multimedia data streams. It also supports advanced games on handheld devices.

The most valuable advantage of the MMF is that it offers developers the possibility to complement an existing design with custom components, ensuring rapid and cost-effective provisioning of advanced services across different hardware platforms. This is due mainly to the presence of a fully customizable, multithreaded multimedia plug-in framework. The framework allows for the dynamic loading and unloading of many specialized pluggable components representing actual resources and abstract components, and lessens the burden of performing complex tasks by delegating them for processing to the specialized plug-ins.

---

## 5. References

Documentation available on the Forum Nokia Web site at  
<http://www.forum.nokia.com/documents>:

- *Symbian OS: Creating Audio Applications In C++*
- *Symbian OS: Creating Video Applications In C++*
- *Series 60 Platform: Techniques To Deal With Lead- And Device-Specific Features In C++*

---

## **6. Evaluate This Document**

In order to improve the quality of our documentation, please fill in the [document survey](#).