

MIDP: Mobile Media API Developer's Guide

Version 1.0; April 7, 2006

Java™

NOKIA

Copyright © 2006 Nokia Corporation. All rights reserved.

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

Disclaimer

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

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

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

The phone UI images shown in this document are for illustrative purposes and do not represent any real device.

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

1	Introduction	6
2	The Mobile Media API	7
2.1	Architecture	7
2.2	Player State Model	8
3	Using the Mobile Media API.....	9
3.1	Player Content Types	9
3.2	Playing Tones and Tone Sequences.....	10
3.2.1	Playing tone sequences.....	10
3.2.2	Playing tones from stream.....	11
3.3	Playing Sampled Sound and MIDI.....	11
3.4	Playing Video.....	12
3.4.1	Saving resources when playing video multiple times.....	13
3.5	Recording Sound and Video	14
3.6	Time Bases	14
3.7	Taking Photos	14
3.8	StopTimeControl.....	15
3.9	VolumeControl	15
3.10	Security Issues	16
3.11	SDK Versus Devices	16
4	Recommended Practices	18
4.1	Content Type Selection.....	18
4.2	Application Resource Management.....	18
4.3	Player Pooling	18
4.4	Remote Resources	19
4.5	Minimizing Media Playback Delay	21
4.6	Restarting the Player After Being Unavailable	21
4.7	Stopping Playback When a MIDlet Is Switched to the Background.....	21
4.8	Playing DRM-Protected Media.....	22
5	Media Sampler MIDlet Design	23
5.1	Media Sampler MIDlet.....	23
5.1.1	AudioCanvas.....	23
5.1.2	Video source selector	24
5.1.3	VideoCanvas	25
5.1.4	DRM MIDI	25
5.1.5	MMAPI support	25
5.2	MIDlet Design.....	26

6	References	28
	Appendix A: System Properties	29
	Evaluate This Resource.....	30

Change History

April 7, 2006	Version 1.0	Replaces the document <i>Brief Introduction to the Mobile Media API</i> .

1 Introduction

The Mobile Media API (JSR-135) (MMAPI, see [6]) is implemented in Nokia devices that support Mobile Information Device Profile (MIDP) 2.0 — in the S60 platform, the MMAPI is supported already from Nokia 3650 onwards. The API allows MIDlets to use various media types more powerfully. These MIDlets are able to play several kinds of audio and video, capture images using the camera (see document [Camera MIDlet: A Mobile Media API Example](#) [1]), and adjust many of these operations using special kinds of controls.

This guide briefly describes and illustrates how to use the Mobile Media API for the above-mentioned actions using some short source code snippets. The document refers to the [MIDP: Mobile Media API Example - Media Sampler](#) [4]. In addition, this guide offers some recommended practices for the MMAPI usage. By following these practices, you can utilize the API more efficiently and powerfully.

Furthermore, the differences between mobile devices and device groups are noted under the relevant topics. The technical reference document, [MIDP: Mobile Media API Support In Nokia Devices](#) [5], gives more detailed information about the capabilities of different devices. The guide assumes that you are familiar with Java™ programming and that you understand the basics of MIDP programming. For more information about MIDP programming, refer to the document [MIDP 1.0: Introduction to MIDlet Programming](#) [2].

2 The Mobile Media API

2.1 Architecture

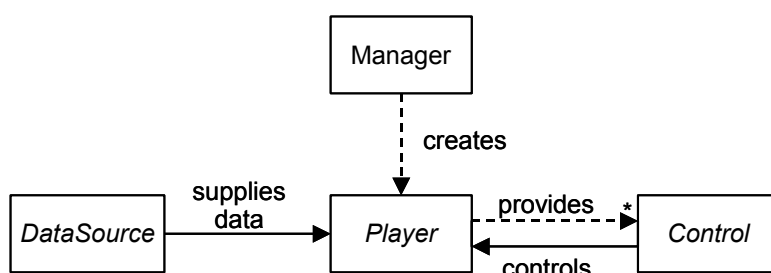


Figure 1: MMAPI architecture

`Manager` is a class with only static methods; applications cannot create an instance of it. `Manager` provides static methods for creating players and querying supported protocols and content types. It also provides a convenient method, `playTone`, for playing a single tone.

`Manager`'s `createPlayer` methods create a `Player` with an associated `DataSource` to supply it with data. This `DataSource` can be built from an `InputStream` or from a URI-style locator. The `DataSource` instance is not visible to the application programmer. Nokia's MMAPI implementation supports, for example, the following locator forms:

- `http://something.com/somefile.wav`
- `capture://video` (displays video from the device's built-in camera)

This document will discuss more about locators in the forthcoming chapters. Audio and video recording is discussed in Section 3.5, "Recording Sound and Video." In addition, the technical note [MIDP: Mobile Media API Support In Nokia Devices](#) [5] contains more information about locator support on device level.

`Manager` creates the correct kind of `Player` implementation class by checking the `DataSource`'s content type (for example, from an HTTP response's Content-Type header) or file extension. If it cannot determine the `DataSource`'s content type, it will throw a `MediaException`.

Once the player has been created, you can ask it for various kinds of `Control`, for instance:

- `VolumeControl` — to control a player's audio volume
- `StopTimeControl` — to make a player stop after playing for a given length of time
- `VideoControl` — to control how a video player's image is shown

There are many other types of `Control`, but not all will be supported by a given device's Mobile Media API implementation, or for a given content type. You can find out what controls are supported in a device by checking the system properties (for example, `System.getProperty("supports.video.capture")`). A full list of these properties is provided in the *Mobile Media API specification* [6].

2.2 Player State Model

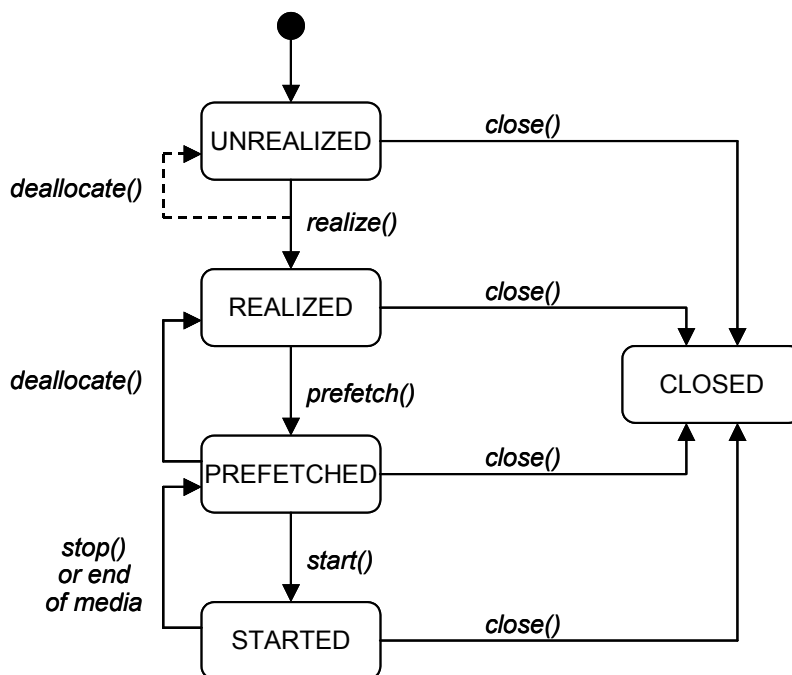


Figure 2: Player state model

Figure 2 shows the state model for `PlayerS`. `PlayerS` are created in the UNREALIZED state, and are typically realized, then prefetched, and then started. When they reach the end of their media file, or `stop()` is called, they return to the PREFETCHED state.

You can check a player's state by calling its `getState` method. But be aware that players can change state dynamically or other threads can change their state, so that by the time the `getState` method returns, its result may no longer reflect the current state of the player.

Similarly, you can get notifications of a player's state changes by registering a `PlayerListener` with it and by using the listener's `playerUpdate` method. But note that the *Mobile Media API specification* gives no guarantees about how promptly the state change events will be delivered, so when you receive the "started" event, the media may already have ended (and there may be an "end of media" event already next in the queue for delivery to you).

Another complication is that the notifications are asynchronous, that is, your event callback may be called from a different thread to the usual user interface thread, and it may be called while your class is already handling a user interface event in a different thread. To get back to the purely sequential event model, the player state change callbacks in the example MIDlet presented in this document use the `callSerially` method of the `MIDP Display` class and simply queue new user interface events. These new events are then handled sequentially with the other user interface events, rather than possibly overlapping.

3 Using the Mobile Media API

Starting from the Mobile Media API version 1.1, API availability can be requested from the system property `microedition.media.version`. It contains the API version in a string format (for example, "1.1"). On devices supporting Mobile Media API version 1.0, other MMAPI system properties, such as `audio.encodings`, can be used instead.

3.1 Player Content Types

The media types listed in Table 1 are supported in the `Manager.createPlayer(InputStream stream, String type)` method. Note that because of platform- and device-specific variance, different subsets of these media types may be supported in devices. For a detailed list of the supported MIME types per device, refer to [MIDP: Mobile Media API Support In Nokia Devices](#) [5].

MIME type	Notes
audio/basic	General audio format. The <code>audio/basic</code> MIME type can be specified in the above method to play back sampled audio streams, for instance, WAV, AU, AMR, MIDI, and SP-MIDI.
audio/wav or audio/x-wav	WAV sound files end with a <code>.wav</code> extension and can be played by nearly all Windows applications that support sound. WAV audio is widely supported by most audio tools. WAV audio is suitable for short sound effects in games.
audio/au or audio/x-au	Short for audio, a common format for sound files on UNIX machines. It is also the standard audio file format for the Java programming language. AU files generally end with an <code>.au</code> extension.
audio/amr	AMR audio is an audio format intended for encoding speech content. The AMR codec is a variable bit rate codec that operates at bit rates in the range of 4.75 Kbit/s to 12.2 Kbit/s for narrowband. AMR audio can be generated with Nokia Multimedia Converter 2.0.
audio/amr-wb	The AMR-WB codec operates at bit rates in the range of 6.6 Kbit/s to 23.85 Kbit/s for wideband.
audio/midi	MIDI audio is a multi-channel synthesized music format. A number of software programs are available for composing and editing music that conforms to the MIDI standard.
audio/sp-midi	Scalable Polyphonic MIDI (SP-MIDI) is a variation of MIDI that supports prioritization of channels, so that a player supporting fewer channels than the MIDI file contains knows which channels to choose to play. SP-MIDI files can be generated from MIDI files using, for example, the Nokia Sound Converter application included in Nokia PC Suite.
video/mpeg4 or video/mp4	Short for Moving Picture Experts Group. MPEG achieves a high compression rate by storing only the changes from one frame to another, instead of each entire frame. MPEG uses a type of lossy compression, since some data is removed. But the diminishment of data is generally imperceptible to the human eye.

MIME type	Notes
video/3gpp	3GPP is the new worldwide standard for the creation, delivery, and playback of multimedia over third-generation, high-speed wireless networks. It is defined by the 3rd Generation Partnership Project. The standard seeks to provide uniform delivery of rich multimedia over newly evolved, broadband mobile networks (third-generation networks) to the latest multimedia-enabled wireless devices. Tailored to the unique requirements of mobile devices, 3GPP takes advantage of MPEG-4.
video/vnd.nokia.interleaved-multimedia	Nokia Interleaved Multimedia (NIM) is a compressed video format supported by the Nokia 9200 Communicator series. NIM video can be produced from other formats using Nokia Multimedia Converter 2.0. In 1st Edition and 2nd Edition compatible devices, 3GPP video should be used rather than NIM video.
application/vnd.rn-realmedia	RealMedia video format. RealMedia files generally end with an <code>.rm</code> extension.

Table 1: Supported media types

A player produced by `Manager.createPlayer(Manager.TONE_DEVICE_LOCATOR)` returns `audio/x-tone-seq` as its content type.

3.2 Playing Tones and Tone Sequences

To play a simple tone once, use:

```
Manager.playTone(note, duration, volume);
```

3.2.1 Playing tone sequences

To play a tone sequence, you must use `ToneControl`. Create a `Player` using a special locator, get the `ToneControl`, set its command sequence, and then start the player:

```
Player player = Manager.createPlayer(Manager.TONE_DEVICE_LOCATOR);
player.realize();
ToneControl tc = (ToneControl) (player.getControl("ToneControl"));
tc.setSequence(new byte[] { ToneControl.VERSION, 1,
                           ToneControl.C4, 8
                           ToneControl.C4 + 2, 8 }); // D4

player.start();
```

Note that you must call `player.realize()` before getting the tone control. You cannot get any type of control from a player before you have realized it.

`ToneControl` has byte commands for playing notes, creating and playing repeatable blocks of notes, and changing tempo, volume, and so on (see the *Mobile Media API specification* [6] for details).

None of the example code fragments presented in this section includes the necessary try/catch blocks for the various exceptions which can be thrown by the method calls.

3.2.2 Playing tones from stream

Tones can be played from stream in the same way as sampled audio and MIDIs.

First create `InputStream` to tone data and pass that stream to the `Player`:

```
InputStream is = getClass().getResourceAsStream("/ukkonooa.jts");
Player player = Manager.createPlayer(is, "audio/x-tone-seq");
```

Once the player is created, you can add a listener, and then realize and prefetch the `Player`:

```
player.addPlayerListener(this);
player.realize();
player.prefetch();
```

Start the player when the tone should be played:

```
player.start();
```

3.3 Playing Sampled Sound and MIDI

Sampled sound means sound formats like WAV, where the data is a stream of sound samples that represent the sound every fraction of a second. MIDI, on the other hand, is a sequence of commands for a multi-instrument “virtual synthesizer.”

To play a sound file accessible over HTTP, use:

```
Player player =
Manager.createPlayer("http://something.com/somefile.wav");
player.start(); // implicitly calls realize() and prefetch()
```

To play a sound file that you have included in the MIDlet’s JAR file, you need to know its MIME type (for example, `audio/x-wav`) in advance. Then use:

```
InputStream is = getClass().getResourceAsStream("/somefile.wav");
Player player = Manager.createPlayer(is, "audio/x-wav");
player.start();
```

Note: In the S60 and Series 80 platform, when using a `null` MIME type parameter in player creation, the implementation may try to search a suitable codec automatically. However, it is recommended to specify the MIME type.

In the Series 40 platform, content type is not auto-detected.

To play a sound file stored in an RMS “record store,” use:

```
RecordStore rs = RecordStore.open("name");
byte[] data = rs.getRecord(id);
ByteArrayInputStream is = new ByteArrayInputStream(data);
Player player = Manager.createPlayer(is, "audio/x-wav");
player.start();
```

Note: When the device is set to the silent mode, it depends on the device whether the audio is still muted or audible when using the MMAPI. For example, when the device is set to silent mode, the S60 test device (Nokia 3660) is completely silent and some Series 40 devices (Nokia 6230, Nokia 6111) play sounds with the MMAPI audibly. Some Series 40 devices (for example, Nokia 6230) have a separate setting for sound in the Games menu. Future Series 40 devices are expected to support the silent mode.

Note: Mixing support is introduced in S60 3rd Edition. In other Nokia devices only one player can be playing at a time. With the MMAPi this is expressed by a notification that mixing is not supported. The support can be checked from the system property `supports.mixing`, which returns a boolean value `true` if the mixing is supported. When mixing is not supported and two players are started sequentially, the last call for the start method will cause the first player to stop and the second to start (not in the Series 40 platform).

Current Series 40 devices do not support true mixing. From Series 40 3rd Edition onwards, high-end devices, such as Nokia 6280, have a “Swap and Play” support that allows a MIDlet to prefetch more than one player, and a player must pause whilst resuming another. This way, several active players can exist at the same time.

3.4 Playing Video

Playing video is similar to playing audio. However, the video player needs to be told where to display the video. Therefore, you get a “video control” from the video player and display it either as a `Form` item or in a `Canvas`.

To display video as a `Form` item:

```
InputStream is = getClass().getResourceAsStream("/somefile.3gp");
Player player = Manager.createPlayer(is, "video/3gpp");
player.realize();
VideoControl vc =
    (VideoControl)player.getControl("VideoControl");
if (vc != null)
{
    Item it =
        (Item)vc.initDisplayMode(VideoControl.USE_GUI_PRIMITIVE,
                                null);
    myForm.append(it);
    p.start();
}
```

To display video in a `Canvas` (for a full example, see the example MIDlet’s source code):

```
InputStream is = getClass().getResourceAsStream("/somefile.3gp");
Player player = Manager.createPlayer(is, "video/3gpp");
player.realize();
VideoControl vc =
    (VideoControl)player.getControl("VideoControl");
if (vc != null)
{
    vc.initDisplayMode(VideoControl.USE_DIRECT_VIDEO, myCanvas);
    vc.setVisible(true);
    p.start();
}
```

Note that by default a video control displayed in a `Canvas` is not visible.

If the videocontrol’s display mode is set to `USE_DIRECT_VIDEO`, the second parameter of the `initDisplayMode` method can be either a `Canvas` or the class extended from `Canvas`.

In the videocontrol’s `setDisplaySize` method, the image will be scaled to fit if the requested display size is different from the dimensions of the video clip. In `Canvas`, videocontrol’s `setDisplayFullScreen(true)` method sets the display to the whole `Canvas`, not the whole

screen. The values returned by the `getDisplayWidth` and `getDisplayHeight` methods are unaffected by this call (that is, they do not afterwards return the width and height of the full screen).

Note: Video scaling is not supported in current Series 40 devices.

3.4.1 Saving resources when playing video multiple times

The video content is often large and playing it multiple times in a row consumes a significant amount of memory if a new `Player` instance is created for each play action. A player can be kept usable, so that playing can be restarted without loading the resources again. If you can assume that the memory of the mobile device does not run out, you can store created and initialized players to a player pool. That makes it faster to restart the video clip, because the content does not need to be reloaded to the memory. In addition, prefetched and realized players are somewhat faster to start.

The following `doPlay()` method from the Media Sampler example MIDlet initializes a player only when needed, and starts playing the video clip:

```
void doPlay()
{
    ...
    if (!initDone || player==null){
        initPlayer();
    }
    int state = player.getState();
    if (state == Player.CLOSED) {
        player.prefetch();
    }
    else if (state == Player.UNREALIZED) {
        player.realize();
    }
    else if (state == Player.REALIZED) {
        player.prefetch();
    }

    player.start();
    ...
}
```

The code snippet above first creates the player by calling the `initPlayer()` method in the MIDlet. After that the player will be initialized, so that it can be started by calling the player's `start()` method. In if-blocks, the player first tries to prefetch if the state is `CLOSED`. Secondly, the unrealized player will be realized and the realized player will be prefetched. Note that if `prefetch()` is called when the player is in the `UNREALIZED` state, it will implicitly also be realized.

Note: In the Series 40 platform and from S60 3rd Edition onwards, when playing some data from a file locator, media will be played directly from the file without loading it to the memory first. This can be useful when playing large media files that already exist in the file system.

3.5 Recording Sound and Video

Sound and video recording can be done by using `RecordControl`. On the S60 platform, sound and video recording is supported from 2nd Edition onwards. Currently the Series 40 platform does not support sound recording, and Nokia 6280 is the first Series 40 device that supports video recording. You can check the support by using the following system properties:

- `supports.audio.capture` — for audio recording
- `supports.video.capture` — for video recording

These properties return `true` if recording is supported.

When creating a `Player` for audio or video capture, or specific capture, the locator URL must be used as a parameter to the `createPlayer` method of the `Manager` class. The following code line creates a `Player` instance for audio capture:

```
Player p = Manager.createPlayer("capture://audio");
```

Camera snapshot for pictures or video recording use the following locators:

- Series 40 platform (Nokia 6280):
 - `capture://image` (for photo capture)
 - `capture://video` (for video recording)
- S60 platform:
 - `capture://video`
 - `capture://devcam0` (first camera, not supported by all devices)
 - `capture://devcam1` (second camera, if the device has two cameras)
- Audio capture locator:
 - `capture://audio`

You can also define the exact recorded media type to the locator URL. The supported parameters vary between platforms and devices. Note that SDKs might not support capture features at all, so testing should be done with the actual mobile devices. For further details about recording, check the *Mobile Media API specification* [6].

3.6 Time Bases

All players support the `setTimeBase` method, provided that the new `TimeBase` was obtained from another player. They do not support the `setTimeBase` method with the MIDlet's own implementations of the `TimeBase` interface.

3.7 Taking Photos

Taking photo snapshots requires that you first create a `Player` for the capture action. Now you can realize the `Player` and get a `VideoControl` from your `Player`'s instance. In your application, you need to specify a locator that is supported by your target device or platform. Supported `Player` locators were discussed in Section 3.5, "Recording Sound and Video."

`VideoControl` has a method, `getSnapshot`, which allows you to get a snapshot of the video it is displaying. This is particularly useful when it is displaying live video from the device's camera, since it allows you to take a photo. The following is a code sample of how to take a photo using default settings:

```
byte[] pngImageData = videoControl.getSnapshot(null);
```

You can also specify other formats if the device supports them, and give a width and height:

```
videoControl.getSnapshot("encoding=bmp"); // BMP, default size
videoControl.getSnapshot("width=80&height=60"); // default encoding, size 80x60
videoControl.getSnapshot("encoding=bmp&width=80&height=60"); // BMP, 80x60
```

If values are specified for width and height, both of them must be specified, not just the width or height. In this case, the image is scaled to the requested width and height. If the requested aspect ratio (that is, 4:3) does not match that of the default size (in most devices 160 x 120), the resulting image will be distorted, or in Series 40 devices, if the width and height do not match a supported size, the closest match is used instead. The maximum size that can be captured depends on the free heap memory available. The system property `video.snapshot.encodings` contains a list of the valid video snapshot encodings for your device. Supported image types can be, for example, `encoding=png`, `encoding=bmp`, `encoding=jpeg`, and `encoding=gif`. JPEG format is the most compact format. The use of compact format reduces memory usage, and is therefore recommended in many cases.

For a complete example of a MIDlet that takes photos using this method, see document [Camera MIDlet: A Mobile Media Example](#) [1].

3.8 StopTimeControl

`StopTimeControl` allows a user to set the media time at which `Player` should stop. `StopTimeControl` usage is similar to other `Control` usage. Get `StopTimeControl` of the player and set the desired stop time in milliseconds with the `setStopTime()` method of `StopTimeControl`.

```
// Get the player whose stop time should be set...
Player player = ...;

// Get the StopTimeControl of the Player
StopTimeControl control = (StopTimeControl)
player.getControl("StopTimeControl");
if (control!=null)
{
    // Set stop time to one second
    control.setStopTime(1000000);
}
```

3.9 VolumeControl

Volume of the player can be set via the `VolumeControl` interface. Once you have created and realized the player, get the `VolumeControl` interface of the player by calling the `getControl()` method and set the desired volume level in a range from 0 to 100 with the `setLevel()` method of `VolumeControl`.

```
// Get the player whose volume level should be changed...
Player player = ...;

// Get the VolumeControl of Player
```

```

VolumeControl control = (VolumeControl)
player.getControl("VolumeControl");
if (control!=null)
{
    // Set new volume level to 50% of maximum
    control.setLevel(50);
}

```

3.10 Security Issues

Many people may consider media capture as a potential security risk. Unsigned MIDP 2.0 MIDlets exist in an untrusted security domain, in which the MIDlet has restricted security access. By signing the MIDlet with a valid code signing certificate, the MIDlet can obtain a trusted third-party security domain access level.

The following methods are MMAPI access points in which a platform checks whether a security prompt is needed. Those prompts are shown to users and they may accept or deny the action. Table 2 shows the API access points in which access checks are performed.

Class	Method
javax.microedition.media.control.RecordControl	setRecordLocation(java.lang.String locator)
javax.microedition.media.control.VideoControl	getSnapshot(java.lang.String imageType)

Table 2: Security access point in the Mobile Media API

Note: Other methods in the Mobile Media API may be used to play protected content, such as network-stored content. In that case security permissions may also apply.

A signed MIDlet may request higher access rights for specific actions by asking for them in the `MIDlet-Permissions` attribute of the MIDlet's application descriptor.

The permission name for audio / video recording is:

```
javax.microedition.media.control.RecordControl
```

The permission name for camera snapshot is:

```
javax.microedition.media.control.VideoControl.getSnapshot
```

For more information about MIDlet signing, see [MIDP 2.0: Tutorial On Signed MIDlets](#) [3].

3.11 SDK Versus Devices

A MIDlet using the MMAPI may not work in exactly the same way in development environments (SDKs) and real devices. The biggest differences between development environments and real devices are the following:

- SDKs and devices may support different sets of media formats. In addition, there are differences between the media format support levels among devices from the same platform.
- Players may have different latencies in devices and SDKs. For example, player realization, prefetching, and starting may take more time on the actual devices because of the underlying hardware differences.
- Input/output quality of the video playback and camera resolutions may differ.

- Memory management (available space, tolerance, garbage collector behavior, and so on) may differ.
- Audio and video capture capabilities may not be implemented on the SDKs.

4 Recommended Practices

4.1 Content Type Selection

It is important to choose a media content type that can be used in many devices or most of the target devices of your MIDP application. With the proper media content type you do not need to constantly update your MIDP application or create multiple versions of it for different device groups. Refer to [MIDP: Mobile Media API Support In Nokia Devices](#) [5] for more information regarding which media contents function on your target mobile devices. It is also possible to query the supported protocols and content types by protocol from the MMAPI. Protocols can be queried with the `getSupportedProtocols` method of the `Manager` class. If you enter a `null` parameter, it returns an array of protocols that the platform's MMAPI implementation should support. The following is an example of the query:

```
String[] protocols = Manager.getSupportedProtocols(null);
```

Secondly, by using the `getSupportedContentTypes` method of the `Manager` class it is possible to obtain an array of content types that a specific protocol supports. With a `null` parameter, all supported content types will be returned. The returned list with the `null` parameter contains supported content types for all the protocols and content types supported in capture. For example, the list might contain a content type that works with RTSP but not with HTTP. To get a protocol-specific content type list, you may first query the list of supported protocols by using the `getSupportedProtocols` method and use one of those as a parameter on the `getSupportedContentTypes` method. The following is an example of how to obtain an array containing supported content types:

```
String protocol = ...
String[] contents = Manager.getSupportedContentTypes(protocol);
```

4.2 Application Resource Management

The place where the players are created is important. Proper application design can reduce the amount of bugs that are hard to find. The following tips may reduce MMAPI-related bugs in MIDlets:

- Do not try to create players before the execution of the `startApp()` method of the MIDlet has begun. For example, do not try to create player objects in the MIDlet constructor or static sections. These are executed before the `startApp()` method of the MIDlet.
- Problems will occur if you create players in the constructors of classes that have members of their object type in the MIDlet or which are further instantiated because of the MIDlet construction.
- Pool players to reduce unnecessary player object re-creation. However, pool only the needed players, because S60 devices have limits on the amount of devices that can be prepared. The amount depends on the device platform (for example, in S60 3rd Edition devices, the limit is around 40).

4.3 Player Pooling

Player pooling is an efficient way to store REALIZED and PREFETCHED state `Player` objects. In general, the start of pooled `Players` should be a bit faster than the unprepared players, because the realization and prefetching of `Players` takes time. A pooled player can be accessed several times just by calling the `Player's` `start()` method. This should minimize the delay in starting the media. Currently, `Player` pooling can be implemented in S60 devices. At least in Series 40 2nd Edition and earlier devices, multiple realized and prefetched players cannot be pooled because only one `Player`

object can be set up at a time. Although some Series 40 devices might allow to realize multiple players, multiple-player prefetching is not supported.

In the following example a `Vector` is used as a `Player` pool. However, any other suitable storage could also be used (for example, array).

```
Vector vect = new Vector(); // Vector to store Players
String file = ... // Give the path of the media file here
String type = ... // Give the media type of the file here
```

Now, let's create a player. First, an `InputStream` needs to be obtained for the media file:

```
// Create first player...
InputStream is = getClass().getResourceAsStream(file);
```

Second, create the `Player` with the `Manager` using the `InputStream` and the media type (MIME; for example, `audio/midi` for MIDI audio) of the file:

```
Player player1 = Manager.createPlayer(is, type);
```

Third, you may optionally make your class listen for the player events:

```
player1.addPlayerListener(this);
```

When doing this, instruct your class to implement the `PlayerListener` interface and add the required `playerUpdate` method. Now the player can be prefetched and realized by calling the `realize()` and `prefetch()` methods:

```
player1.realize();
player1.prefetch();
```

When the players are realized and prefetched, they are ready to play. This can be seen from Figure 2, which describes the states of a player. When creating other players, just repeat the previous steps. Finally, after you have created all the needed players, store them in the pooling place, which is in this case a `Vector` object:

```
vect.addElement(player1);
vect.addElement(player2);
```

When you need to access the player, just obtain the reference to the player object from the pool. The following code lines show how to get a player from the vector pool and start playing the sound:

```
// Play the second player
Player player2 = vect.elementAt(1);
player2.start();
```

Note: Do not close the player if you are going to use it later. If the player has been closed, it needs to be reinitialized before reuse. However, if the player is not needed any longer, close it and remove it from the pool to let garbage collector clean the memory.

4.4 Remote Resources

The JAR archive size is limited in Series 40 devices. That is why all the audio or video resources the MIDlet needs may not fit into the JAR archive. To overcome this limit, the MIDlet may download content for the MMAPi use from an HTTP server. The downloaded media can be stored to the persistent storage the device supports, for example, `RecordStore` or a file. In addition, using an external HTTP server as a media content source allows the media to be updated more easily in a centralized way.

This document defines two optional ways for downloading video content from an HTTP server. With the first alternative you can give media files a URL to the `createPlayer()` method as a parameter. The following is an example of how to do this:

```
String url = "http://www.myhost.com/videos/video.3gp";
Player player = Manager.createPlayer(url);
```

In the Series 40 platform, the above method is blocking, which means that it does not return until the media is loaded. In the S60 platform, this will just open the connection, not load the media. Media will be loaded in prefetch. In S60 3rd Edition, there can also happen recognition from data (so it might also load a small piece of data). Recognition in the Series 40 platform and in S60 3rd Edition is done in the following way:

1. Trying to recognize the type from the URL. In this case with `.3gp` (this is the most likely to be recognized).
2. If there is no URL, recognition is done with the provided content type (InputStream and MIME).

If there is no recognition with the above cases, or in the second case `null` was given as content type, recognition will be tried with header data.

The second alternative uses buffer technique. It gives the MIDlet the possibility to handle the reading process in a customized way. When a separate thread is used in reading, the media can be loaded in a non-blocking way. To simplify the reading example, thread usage is skipped. The following is an example of how to use the buffering technique:

```
private InputStream urlToStream(String url) throws IOException
{
    // Open connection to the http url...
    HttpURLConnection connection = (HttpURLConnection) Connector.open(url);
    DataInputStream dataIn = connection.openDataInputStream();
    byte[] buffer = new byte[1000];
    int read = -1;
    // Read the content from url.
    ByteArrayOutputStream byteout = new ByteArrayOutputStream();
    while ((read=dataIn.read(buffer))>=0)
    {
        byteout.write(buffer, 0, read);
    }
    dataIn.close();
    connection.close();
    // Fill InputStream to return with content read from the URL.
    ByteArrayInputStream byteIn =
        new ByteArrayInputStream(byteout.toByteArray());
    return byteIn;
}}
```

In the example, `HttpURLConnection` is used to open a networking connection. In the code snippet, byte data is read from the server to the 1000 bytes length buffer by using `DataInputStream (dataIn)`. The read operation continues in a while loop until there is no more data left to read from the server. Inside the loop, the buffered data is written to `ByteArrayOutputStream (byteout)`. Finally, after exiting the while loop, the byte data from `ByteOutputStream` is converted to `ByteArrayInputStream (byteIn)`. Now, `InputStream` returned by the example's `urlToStream(String url)` method above can be used to create a `Player` instance with the `createPlayer` method of the `Manager` class:

```
createPlayer(java.io.InputStream stream, java.lang.String type)
```

Note: HTTP streaming is not supported on any current platform. RTSP streaming is supported from S60 2nd Edition, Feature Pack 3 onwards. The list of supported protocols can be queried using the `Manager.getSupportedProtocols (null)` method. When RTSP streaming is used, the whole file is read to the memory before playback.

4.5 Minimizing Media Playback Delay

Especially in games it is important to minimize the media playback delay to get the best possible gameplay experience. To make your MIDlet use the MMAPI as smoothly as possible, consider the following in your MIDlet's design:

- Avoid using `System.gc ()` to garbage-collect your objects while using players (for example, realizing or playing sound). If garbage collection is performed during playback, it may affect the audio start time.
- Prepare only the needed audio players and pool them. See also Section 4.2, "Application Resource Management."
- Avoid blocking UI thread, for example, by using blocking methods. An example of this is the `createPlayer` method of the MMAPI `Manager`. Section 4.4, "Remote Resources," discusses how streams and threading can be used to prevent blocking.
- Use synchronized blocks in your code only when really needed, for example, to prevent two threads accessing the same method concurrently. Synchronized block code execution may be somewhat slower than regular code blocks.

4.6 Restarting the Player After Being Unavailable

MIDlet device availability events are sent through the `javax.microedition.media.PlayerListener` interface. The device sends the `DEVICE_UNAVAILABLE` state event to the registered `PlayerListeners` when a player becomes unavailable, and `DEVICE_AVAILABLE` state event when the device is again available. The player will be in the realized state when it becomes available, so the MIDlet will not need to call the `realize` method before restarting the sound.

Note: When a voice call interrupts the MMAPI MIDlet use, Series 40 devices send the `DEVICE_UNAVAILABLE` event. S60 devices can handle set `PlayerS`, so players should not usually receive unavailability events during voice calls.

4.7 Stopping Playback When a MIDlet Is Switched to the Background

S60 devices are Symbian OS devices and support multitasks. When switching a MIDlet to the background, audio stream playback does not stop. Developers should implement stopping playback at the moment the MIDlet loses foreground. With a low-level UI, such as `Canvas` and `GameCanvas`, stopping playback can be implemented in the `Canvas.hideNotify ()` method, which is called just after the MIDlet loses the foreground. Restarting playback can be implemented in the `Canvas.showNotify ()` method, which is called just before the MIDlet returns to the foreground. With a high-level UI, such as `List` and `Form`, `Displayable.isShown ()` can be used to check whether the calling `Displayable` is actually visible on the display. Developers can implement checking `isShown ()` in a proper frequency to stop or restart playback.

4.8 Playing DRM-Protected Media

Digital Right Management (DRM) protected files are supported at the moment only in S60 3rd Edition devices. These files can be played by using the MMAPI's `Manager.createPlayer(String locator)` method. As an example, a player using DRM-protected audio content could be created in the following way:

```
Player player =  
Manager.createPlayer("file:///C:/Data/Sounds/Digital/Song.dcf");
```

Note: The `Manager's createPlayer(InputStream stream, String type)` method does not work with DRM-protected content. There is no need to indicate in any way that the file to be played back is DRM-protected. In addition, it is not possible to play the DRM-protected files packaged into the JAR archive.

5 Media Sampler MIDlet Design

5.1 Media Sampler MIDlet

The main screen of the [MIDP: Mobile Media API Example - Media Sampler](#) [4] is a list (also called *MediaList*) that shows the available MMAPI examples (see Figure 3).



Figure 3: MediaList on the Media Sampler MIDlet

MediaList contains four options:

- Play audio: Guides how to play audio with cached players on canvas.
- Play video: Guides how to play video on canvas.
- DRM MIDI: Guides how to load a DRM-protected data MIDI file.
- Check MMAPI support: Guides how to enquire MMAPI-specific properties.

5.1.1 AudioCanvas

When the “Play audio” selection has been made, the *AudioCanvas* is set to visible (see Figure 4).



Figure 4: Audio Canvas

The *AudioCanvas* canvas allows a user to play sounds that are mapped to the numeric keys of the device keyboard. All sounds that are supported by the device are displayed on the canvas below the “Sound key mapping:” text. The file path in the JAR file and the MIME type of the media are displayed

after the numeric key to which the sound is mapped. You can play the sound by pressing the numeric key to which the sound is mapped from the device keyboard.

AudioCanvas has two commands: **Info** and **Back**. You can return to the *MediaList* list by selecting the **Back** command on the *AudioCanvas* main view. A device or SDK may set up the **Info** command to **Options** menu as in Figure 4. When you select the **Info** command, you will see extra information (Info view, see Figure 5) regarding sounds that are not supported by the device. You can also see information about whether the MIDlet was able to create multiple media player instances and cache them. From the *Info view* you can return to the main view of the *AudioCanvas* by selecting the **Back** command.



Figure 5: Info view on Audio Canvas

5.1.2 Video source selector

Video source selector presents a list from which you can select a video source.

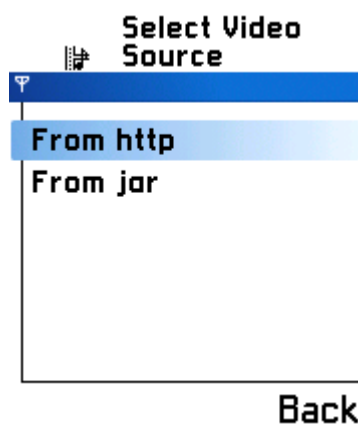


Figure 6: Video source selector

There are two selections: *From http* and *From jar*. If you select *From http*, the HTTP URL input form is shown. You can type there the HTTP URL of the video to be rendered on the *VideoCanvas*. If you select *From jar*, the video rendered on the *VideoCanvas* is loaded from the JAR file. Once the **OK** command is selected, the video source selection is passed to the *VideoCanvas* and it becomes visible.

5.1.3 VideoCanvas

VideoCanvas is a LCDUI Canvas component, which displays the video selected on the *Video source selector* view.



Figure 7: Video Canvas

The video will start automatically when the *VideoCanvas* is shown and the device is ready to render the video. The Video can be stopped with the Stop command by pressing the **FIRE** (5) key while the video is playing. A stopped video can be restarted and a paused video can be resumed by pressing the **FIRE** key, or by selecting the Replay command.

5.1.4 DRM MIDI

DRM MIDI guides the loading of DRM-protected media by using the Mobile Media API. Note that DRM-protected content parsing is supported only on S60 3rd Edition and onwards.

5.1.5 MMAPI support

The *MMAPI support view* displays MMAPI-specific information gathered from the device.



Figure 8: MMAPI support view

5.2 MIDlet Design

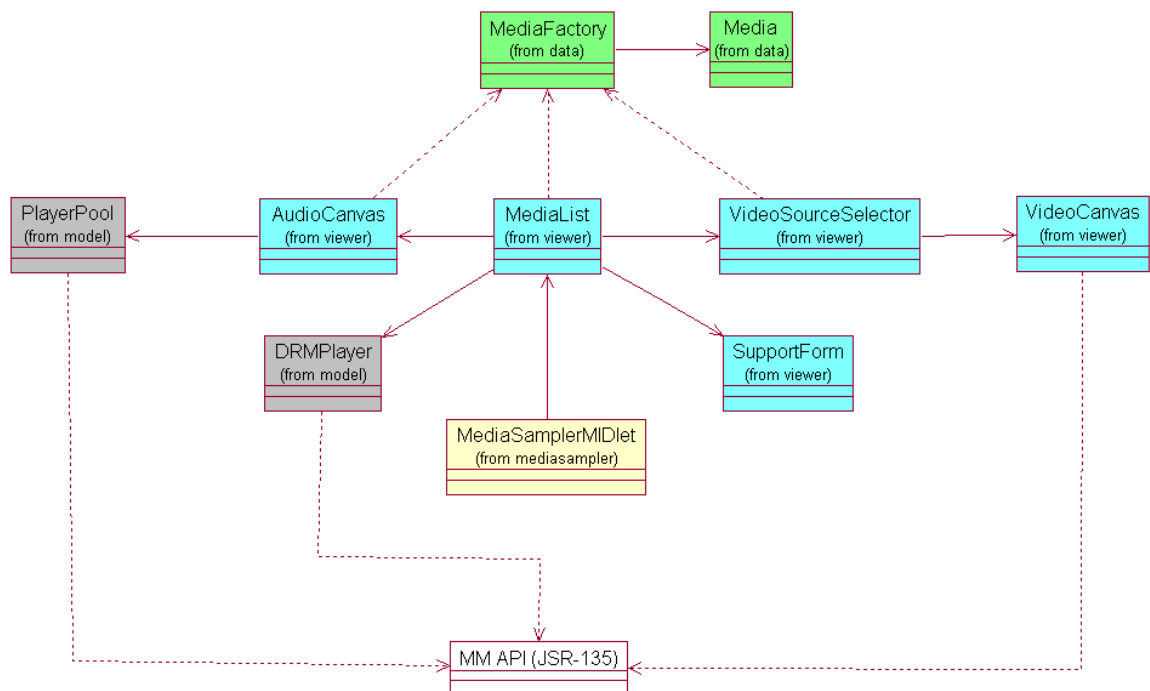


Figure 9: Media Sampler MIDlet's design

The Media Sampler MIDlet consists of the following classes:

- MIDlet class:
 - `com.nokia.example.mmapi.mediasampler.MediaSamplerMIDlet`
- UI classes (in viewer package):
 - `com.nokia.example.mmapi.mediasampler.viewer.AudioCanvas` (Canvas)
 - `com.nokia.example.mmapi.mediasampler.viewer.MediaList` (List)
 - `com.nokia.example.mmapi.mediasampler.viewer.SupportForm` (Form)
 - `com.nokia.example.mmapi.mediasampler.viewer.VideoCanvas` (Canvas)
 - `com.nokia.example.mmapi.mediasampler.viewer.VideoSourceSelector` (List)
- Model classes:
 - `com.nokia.example.mmapi.mediasampler.model.DRMPlayer`
 - `com.nokia.example.mmapi.mediasampler.model.PlayerPool`
- Data type classes:
 - `com.nokia.example.mmapi.mediasampler.data.Media`
 - `com.nokia.example.mmapi.mediasampler.data.MediaFactory`

The *MediaList* main view is created and displayed by the *MediaSamplerMIDlet* class. A reference of *MediaSamplerMIDlet* is passed to all LCDUI components of the application (*MediaList*, *AudioCanvas*, *VideoSourceSelector*, and so on). The reference is used for getting the display and for displaying an alert when an error occurs. *MediaList* creates and displays the classes *AudioCanvas*, *VideoSourceSelector*, *SupportForm*, and *DRMPlayer*.

`PlayerPool` is a class that handles audio player pooling and playing. `DRMPlayer`'s task is to play DRM-protected audio files.

The `MediaFactory` class provides media-specific information such as available sound and video medias, their MIME types and file paths in an application package (JAR). `MediaFactory` reads resource-specific file path properties from the application descriptor (JAD). Data information returned from `MediaFactory` is encapsulated in the `Media` class.

UI classes in the viewer package display the UI or route to another display.

6 References

- [1] [Camera MIDlet: A Mobile Media API Example](http://www.forum.nokia.com), available at www.forum.nokia.com
- [2] [MIDP 1.0: Introduction to MIDlet Programming](http://www.forum.nokia.com), available at www.forum.nokia.com
- [3] [MIDP 2.0: Tutorial On Signed MIDlets](http://www.forum.nokia.com), available at www.forum.nokia.com
- [4] [MIDP: Mobile Media API Example - Media Sampler](http://www.forum.nokia.com), available at www.forum.nokia.com
- [5] [MIDP: Mobile Media API Support In Nokia Devices](http://www.forum.nokia.com), available at www.forum.nokia.com
- [6] Mobile Media API (JSR-135) Specification, Java Community Process, <http://www.jcp.org>

Appendix A: System Properties

System property	Description
supports.mixing	<p>A query for whether audio mixing is supported. The string returned is either true or false. If mixing is supported, the following conditions are true:</p> <ul style="list-style-type: none"> • At least two tones can be played with <code>Manager.playTone</code> simultaneously. • <code>Manager.playTone</code> can be used at the same time when at least one player is playing back audio. • At least two players can be used to play back audio simultaneously.
supports.audio.capture	A query for whether audio capture is supported. The string returned is either true or false.
supports.video.capture	A query for whether video capture is supported. The string returned is either true or false.
supports.recording	A query for whether recording is supported. The string returned is either true or false.
audio.encodings	The string returned specifies the supported capture audio formats.
video.encodings	The string returned specifies the supported capture video formats.
video.snapshot.encodings	Supported video snapshot (i.e., still image) formats for the <code>getSnapshot</code> method in <code>VideoControl</code> .
microedition.media.version	Returns the implementation's version, for example "1.1" for an implementation that is compliant with MMAPI 1.1.
streamable.contents	Returns formats that can be streamed.

Evaluate This Resource

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