



Working with the Mobile Media API – Part 2

Author: Martin de Jode
Status: Version 1.1
Date: 2/7/03

1. Introduction

This paper is the second in a two-part series devoted to programming the Mobile Media API (JSR 135). In Part 1 we introduced the Mobile Media API (MMAPI), discussed the architecture of the API and demonstrated video playback using the MMAPI. In this paper we discuss the further functionality available in the API including audio playback, photo capture and tone generation.

Part one of the [paper](#) is available from the Symbian Developer Network portal and should be read in conjunction with this paper.

2. Playing Audio

In this section we shall demonstrate playing audio files. Playing audio follows the same basic principles as playing video, so we refer the reader to [“Working with the Mobile Media API - Part 1”](#) for an introduction to using players.

We shall illustrate the section with code from a simple Audio Player `MIDlet`. Screen shots from the `MIDlet` are shown below:

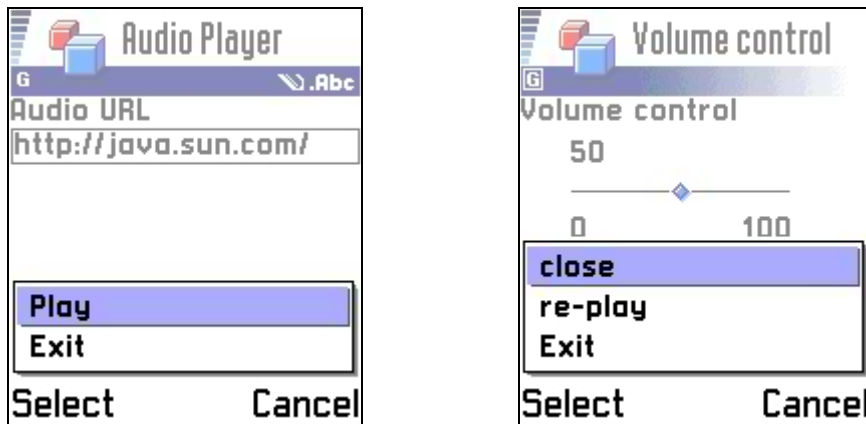


Figure 1 Screen shots of the Audio Player MIDlet

The Audio Player MIDlet consists of four classes: A controller, `MIDletController`; Two views, `InitialView` and `PlayerView`; and the player, `AudioPlayer`. The `AudioPlayer` class performs the actual playing of the audio file and we shall consider it first. The key signatures are shown below:

```
import javax.microedition.media.*;
import javax.microedition.media.control.*;
import java.io.*;

public class AudioPlayer implements Runnable {
    private MIDletController controller;
    private Player player;
    private VolumeControl volumeControl;
    private String url;

    public AudioPlayer(MIDletController controller){...}

    public void initializeAudio(String url){...}

    public void run(){...}

    public void playAudio(){...}

    public void replay(){...}

    public void setvolume(int level){...}

    public void close(){...}
}

```

We shall first look at the constructor.

```
public AudioPlayer(MIDletController controller){
    this.controller = controller;
}

```

This simply takes a reference to the `MIDletController` object, allowing the `AudioPlayer` to make callbacks to the controller.

As in the Video Player example illustrated in the earlier [paper](#) we do the initialization of the `Player` in a separate thread launched from the `initializeAudio(...)` method shown below. The initialization is performed in a separate thread to enable the update of a gauge owned by `InitialView` indicating the progress of the video acquisition. Generally it is good practice to put the, potentially time consuming, initialization in a separate thread, possibly performing it in the background to reduce latency.

```
public void initializeAudio(String url){
    this.url = url;
    Thread initializer = new Thread(this);
    initializer.start();
}
```

The actual initialization takes place in the `run()` method

```
public void run(){
    try {
        player = Manager.createPlayer(url);
        controller.updateProgressGauge();
        player.addPlayerListener(controller);
        player.realize();
        controller.updateProgressGauge();
        player.prefetch();
        controller.updateProgressGauge();
        volumeControl = (VolumeControl)player.getControl("VolumeControl");
        volumeControl.setLevel(50);
        controller.updateProgressGauge();
        player.start();
    } catch (IOException ioe) {
        controller.showAlert("Unable to connect to resource", ioe.getMessage());
    } catch (MediaException me) {
        controller.showAlert("Unable to create media player", me.getMessage());
    }
}
```

A `Player` is created and moved through its states. The `controller` reference serves two purposes: first to facilitate callbacks to the UI indicating the progress of the initialization; and secondly the controller acts as the `PlayerListener` which will be notified of `Player` events. In this example we obtain a `VolumeControl`, although it is not essential for simple audio playback. A `VolumeControl`, naturally enough, provides control over the audio playback volume, including a mute option. The volume range provided by a `VolumeControl` ranges from 0-100. Here we set the volume level midway. We then start the `Player`.

Our `AudioPlayer` class also contains some simple service methods. The `replay()` method is used to immediately re-start an already initialized audio player.

```
public void replay(){
    try{
        player.start();
    } catch (MediaException me) {
```

```

        controller.showAlert("MediaException thrown", me.getMessage());
    }
}

```

The `setVolume()` method is used to change the volume level of the audio playback via the `VolumeControl`

```

public void setVolume(int level){
    volumeControl.setLevel(level);
}

```

The `close()` method is called when the user has finished with the audio player. It calls the `Player close()` method to de-allocate resources held by the player and close it down. Additionally it performs some tidying up so the `AudioPlayer` instance can be garbage collected.

```

public void close(){
    player.close();
    controller = null;
    url = null;
}

```

We shall also look briefly at the `MIDletController` class. The `MIDletController` sets up the instances of the views, `InitialView` and `PlayerView` (see Figure 1). The `InitialView` instance provides a `TextField` for the audio file url, and a `Gauge` to indicate the progress of the audio player initialization, as well as controls to start playback or exit the application. The `PlayerView` instance provides an interactive volume indicator, allowing the user to adjust the playback volume, plus controls to close the player or replay the audio content. The `MIDletController` constructs the `AudioPlayer` instance in response to user commands from the `InitialView`. The `MIDletController` receives callbacks from the `AudioPlayer`, via the `PlayerListener playerUpdate(...)` method, and updates the `InitialView` progress gauge as the player moves through its initialization states. The `MIDletController` also receives callbacks from the `PlayerView` when the user adjusts the volume slider and passes these calls to the `AudioPlayer`. The key signatures of the `MIDletController` class are shown below:

```

import javax.microedition.lcdui.*;
import javax.microedition.midlet.*;
import javax.microedition.media.*;
import javax.microedition.media.control.*;

public class MIDletController extends MIDlet implements CommandListener,
PlayerListener, ItemStateListener {

    public MIDletController() {}

    public void startApp(){}

    public void pauseApp(){}

    public void destroyApp(boolean unconditional){ }

    public void commandAction(Command c, Displayable s){ }
}

```

```

    public void playerUpdate(Player p, String event, Object eventData) { }

    public void itemStateChanged(Item item){ }

    public void updateProgressGauge(){ }

    public void showAlert(String title, String message){ }
}

```

The `playerUpdate(...)` method is shown below.

```

public void playerUpdate(Player p, String event, Object eventData) {

    if (event == PlayerListener.STARTED) {
        if(closeCommand == null){
            closeCommand = new Command("close", Command.SCREEN, 1);
            playerView.addCommand(closeCommand);
        }
        display.setCurrent(playerView);
    }

    else if (event == PlayerListener.VOLUME_CHANGED) {
        VolumeControl volumeControl = (VolumeControl)eventData;
        int currentLevel = volumeControl.getLevel();
        if(playerView.getVolumeIndicator() != currentLevel)
            playerView.setVolumeIndicator(currentLevel);
    }

    else if (event == PlayerListener.END_OF_MEDIA){
        if (replayCommand == null){
            replayCommand = new Command("re-play", Command.SCREEN, 1);
        }
        playerView.addCommand(replayCommand);
    }
}

```

In this example three types of `PlayerListener` events are processed, `STARTED`, `VOLUME_CHANGED` and `END_OF_MEDIA`. `VOLUME_CHANGED` events are processed to ensure the volume indicator gauge of `PlayerView` correctly reflects the volume level of the `AudioPlayer`'s `VolumeControl`.

```

public void itemStateChanged(Item item){
    if (item instanceof Gauge){
        Gauge volumeIndicator = (Gauge)item;
        int level = volumeIndicator.getValue();
        audioPlayer.setVolume(level);
    }
}

```

The `itemStateChanged(...)` method listens for requests by the user to change the volume level via the interactive gauge of `PlayerView`.

The full source code for the `Audio Player MIDlet` is available from the Symbian Developer Network [portal](#).

3. Tone generation

The Mobile Media API also supports tone generation. Generating a single tone is simply achieved using the

```
public static void playTone(int note, int duration, int volume) throws MediaException
```

method of the `Manager` class. The note is passed as an integer value in the range 0 – 127, the duration is specified in milliseconds and the volume is an integer value on the scale 0 – 100.

To play a sequence of tones we need to create a `Player` and use it to obtain a `ToneControl`.

```
byte[] toneSequence = { ToneControl.C4, ToneControl.C4 + 2, ToneControl.C4 + 4, ...};
try{
    Player player = Manager.createPlayer(Manager.TONE_DEVICE_LOCATOR);
    player.realize();
    ToneControl control = (ToneControl)player.getControl("ToneControl");
    control.setSequence(toneSequence);
    player.start();
} catch (IOException ioe) {
} catch (MediaException me) { //handle }
```

A tone sequence is specified as a list of tone-duration pairs and user-defined sequence blocks using Augmented Backus-Naur form (ABNF) syntax (refer to the [MMAPI specification](#) for more detail). The list is packaged as a byte array and passed to the `ToneControl` using the `setSequence(...)` method. To play the sequence we simply invoke the `Player start()` method.

Note the `ToneControl` interface defines some useful static byte constants including `C4`, representing middle C.

4. Capturing media

The Mobile Media API provides support for capturing audio or video from the onboard hardware (microphone or camera) with the “capture” locator syntax. Capture can be used, for instance, to display input from the camera on the screen. Perhaps more importantly, if the implementation supports the appropriate `RecordControl` this can be used to record the media input. You will recall from [Part 1](#) that `RecordControls` are one of the optional controls so there is no requirement to support recording even if the respective media playback is supported.

In this section we will look at code fragments using the `capture` syntax to record audio and take photographs.

```

try {
    // Create a Player that captures live audio.
    Player player = Manager.createPlayer("capture://audio");
    player.realize();
    RecordControl rc = (RecordControl)player.getControl("RecordControl");
    ByteArrayOutputStream output = new ByteArrayOutputStream();
    rc.setRecordStream(output);
    player.prefetch();
    rc.startRecord();
    player.start();
    Thread.currentThread().sleep(5000);
    rc.commit();
    player.close();
} catch (IOException ioe) {
} catch (MediaException me) {
} catch (InterruptedException ie) {
}
}

```

The above code records from the audio device for five seconds. A `Player` is created using the `capture` locator syntax. The player must be realized before a control can be obtained. A `RecordControl` is then obtained from the player and used to record from the audio device. Note the `RecordControl startRecord()` method will start recording immediately if the player has already started. However, if the player has not yet started, the recorder will remain in standby mode until the player starts, whence it will start recording immediately.

We will now look at taking a snapshot from the onboard camera. First we have to set up the camera.

```

Player player;
VideoControl videoControl;

...

try{
    player = Manager.createPlayer("capture://video");
    player.realize();
    videoControl = (VideoControl)(player.getControl("VideoControl"));

    if (videoControl != null) {
        Form form = new Form("video");
        Item guiPrimitive =
            (Item)videoControl.initDisplayMode(videoControl.USE_GUI_PRIMITIVE, null);
        form.append(guiPrimitive);
        Display.getDisplay(midlet).setCurrent(form);
    }
    player.start(); //implicitly calls prefetch() if not in PREFETCHED state.
} catch(MediaException me){
}
}

```

The code paragraph above simply relays the video output from the camera to the display (for more information on using the `VideoControl initDisplayMode(...)` method to display video content see [Part 1](#)). Taking a snapshot is simply achieved using the `getSnapshot (...)` method of `VideoControl`.

```
try {
    byte[] pngImage = videoControl.getSnapshot(null);
    // do something with the image ...
} catch (MediaException me) { }
```

The `getSnapshot(String imageType)` method takes a `String` argument indicating the image format of the returned snapshot. A `null` value indicates the default PNG format which all implementations must support. If the implementation supports other formats these can be specified using the `imageType` argument. To ascertain other formats supported by a particular implementation make a call to

```
System.getProperty(video.snapshot.encodings).
```

For a detailed example of taking snapshots see Nokia's [Camera MIDlet example](#)

5. Other useful features of the Mobile Media API

The design of Mobile Media API allows implementations to provide optional support for different media types and protocols. Not all devices will fully implement the entire MMAPI specification. For instance the implementation on a phone without a camera (or attachment) is unlikely to support the `capture://video` syntax. Similarly not all implementations will support all multimedia types and input protocols. Some implementations may support only a few selected types and protocols.

To allow developers to ascertain which features of the Mobile Media API are supported several useful properties are defined that can be queried with `System.getProperty(...)`. The API supplies the following `Strings` as keys:

```
supports.mixing
```

```
supports.audio.capture
```

```
supports.video.capture
```

```
supports.recording
```

```
audio.encodings
```

```
video.encodings
```

```
video.snapshot.encodings
```

For the format of the returned `String` refer to the [MMAPI specification](#).

The `Manager` class also contains a couple of useful static methods namely

```
String[] getSupportedContentTypes(String protocol)
```

```
String[] getSupportedProtocols(String content_type)
```

The first method takes a protocol such as “http” and returns the MIME type of the supported content types for this protocol. Conversely the second method takes the MIME type of the content type, for example `video/mpeg` and returns the protocols that can be used to deliver this content type.

6. Mobile Media API on the Nokia 3650

In this final section we will briefly consider the implementation of the Mobile Media API as implemented by Nokia on the Nokia 3650. The Nokia 3650 is a Symbian OS v6.1 based Nokia Series 60 camera phone supporting MIDP 1.0, the Mobile Media API and the Wireless Messaging API.



Figure 2 Nokia 3650 camera phone

The Nokia 3650 provides a fairly full implementation of the Mobile Media API specification, including support for video and audio playback, photo capture and tone generation.

In particular the Nokia 3650 supports video playback of the following content types:

`video/3gpp`

video/vnd.nokia.interleaved-multimedia

The 3gpp format is a packet-switched multimedia streaming service standard for 3G mobile phones, based on the MPEG-4 standard. The Nokia interleaved multimedia (nim) content type is a proprietary, highly compressed, multimedia format. Video-playback over HTTP is supported, although streaming, in the literal sense, is not supported by the Nokia 3650; instead the HTTP stream is read to completion before playing commences. The impact of this on player latency and memory requirements should be borne in mind by developers particularly when playing video files over HTTP. In addition to a `VideoControl`, video players support the `StopTimeControl` and the `VolumeControl`.

In terms of audio support, the Nokia 3650 Mobile Media API implementation supports sound play-back of the following formats:

audio/x-wav, audio/amr, audio/midi, audio/sp-midi, audio/x-nokia-rng,
audio/x-tone-seq.

Audio players of the above content types support a `VolumeControl` and `StopTimeControl`, additionally `audio/x-tone-seq` players support a `ToneControl`.

Photo capture using the onboard camera on the Nokia 3650 supports the image formats listed below:

png, bmp, jpeg

The Following are not supported on the Mobile Media implementation on the Nokia 3650:

- Video recording
- Audio recording
- RTP streaming
- Mixing - only one player can be playing at any one time

The Real-time Transport Protocol, RTP locator syntax is defined in the MMAPI specification but not supported by the Nokia 3650 implementation.

7. Summary

This paper is the final paper in a two part series covering the Mobile Media API (JSR 135). In [Part 1](#) the Mobile Media API was introduced and its architecture discussed. The concepts introduced were then illustrated using video playback as an example.

This second paper has gone on to discuss audio playback, tone generation and using the capture locator syntax. We have also considered the specific implementation of the Mobile Media API on the Symbian OS based Nokia 3650 camera phone.

This series has not intended to provide an exhaustive exposition of the Mobile Media API, but instead give enough information to get a developer started with JSR 135. For a more comprehensive review of the API download the full specification from the JCP [Web site](#) and also check out the other references listed below.

8. Resources

Working with the Java Mobile Media API – Part 1

http://www.symbian.com/developer/techlib/papers/Working_with_Mobile_Media_API.pdf

Mobile Media API Specification

<http://jcp.org/aboutJava/communityprocess/final/jsr135/>

Mobile Media API Emulator for the J2ME Wireless Toolkit

<http://java.sun.com/products/mmapi/emulator/>

Series 60 Concept SDK incorporating support for the Mobile Media API

<http://forum.nokia.com>

Forum Nokia Papers

<http://forum.nokia.com>

“A Brief introduction to the Mobile Media API”

“Technical Note: The Nokia 3650 Mobile Media API”

“Camera MIDlet: A Mobile Media API Example v1.0”

AudioPlayer MIDlet example code

<http://www.symbian.com/developer/techlib/staffapps.html>