Helsinki University of Technology
Telecommunications Software and Multimedia Laboratory
3.12.2004
T-111.550 Multimedia Seminar
Autumn 2004

MIDP 2.0 Sound API and JSR-135/234

Teemu Harju (55730D)
Antti Nummiaho (48004M)

MIDP 2.0 Sound API and JSR-135/234

Teemu Harju (tsharju@cc.hut.fi)
Antti Nummiaho (anummiah@cc.hut.fi)

Abstract

Mobile media devices and their multimedia features have been developing rapidly during the last few years. In this paper we inspect audio features of Mobile Media API and Advanced Multimedia Supplements developed for Java 2 Platform Micro Edition. We go through the essential features and architectural components of these two APIs. Some development related issues are also dealt in this document through simple code examples. There are currently several different mobile devices on market. We take a brief look on their support for these audio features. Possible future audio supplements to MMAPI in addition to AMMS are also considered mainly by comparing MMAPI and AMMS to Java Media Framework for J2SE. Main findings of this paper are that audio support for current mobile devices is extremely comprehensive and that AMMS is a very extensive supplement to MMAPI.

1 Introduction

Java-enabled mobile devices have recently become more and more common among regular customers. Latest mobile phones have microphones and cameras and are hence equipped with quite good multimedia capabilities. To be able to take full advantage of this hardware some kind of software platform is also required. The Micro Edition of the Java 2 Platform (J2ME) provides an application platform that fits to this purpose. J2ME has application programming interfaces (APIs) such as Mobile Media API (MMAPI, JSR-135) and Mobile Information Device Profile 2.0 (MIDP 2.0, JSR-118) that provide some basic multimedia features. Also, in the near future a new supplement to MMAPI called Advanced Multimedia Supplements (AMMS, JSR-234) will be finalized. It will bring multimedia features to mobile devices that are quite close to the ones in desktop devices.

In this paper we will deal only with the audio features of these three APIs. We will explain the basic architecture behind audio in mobile devices and show some examples how some features can be implemented. Some attention has also been given to the support of these APIs in the current mobile multimedia devices as well as the comparison to Java Media Framework (JMF) that has been developed for desktop devices.

2 Mobile Media API (JSR-135)

2.1 Overview

Mobile Media API is a standardized way to bring multimedia into mobile devices. The MMAPI specification is defined in Java Specification Request 135 (JSR-135). MMAPI is created based on the concepts of Java Media Framework for Java 2 Standard Edition but targeted for mobile devices with limited audio and video capabilities, processing power and memory size. MMAPI's key feature is that it is not designed to be used with certain protocols or media formats. Rantalahti *et al.* , 2003

The MMAPI specification is an optional package that can be adopted to any J2ME-enabled device. The main target for the API is a CLDC based device, but other environments like CDC are not excluded. Latest version of the MMAPI specification is version 1.1, which was finalized in June 2003. It does not differ much from version 1.0. Basically only some documentation updates have taken place and the MIDP 2.0 security framework has been taken into account. Rantalahti *et al.* , 2003

Almost all modern mobile multimedia devices on market support MMAPI.

2.2 Features

Tone generation, audio playback and audio recording are MMAPI's essential audio features. MMAPI has also some interesting general features. All main audio and general features of the MMAPI are listed and explained shortly in the following. Mahmoud, 2003; Rantalahti & Huopaniemi, 2003

- Support for tone generation. The API includes an ability to play single tones or to create and play tone sequences. Tempo and volume changes in tone sequences are also supported.
- Support for audio playback and recording with some basic controls like volume control.
- Support for interactive MIDI. An application can trigger MIDI events directly to create music or sound effects on the fly.
- Support for audio seeking that is the API enables the ability to move ahead or behind in time while playing the audio.
- Support for audio synchronization that is the API enables the ability to use timing information in an audio stream to decide when to play out sample from buffer.
- Protocol and format independency. An implementation of the API can support any formats and protocols that it needs. Formats are handled as MIME types.

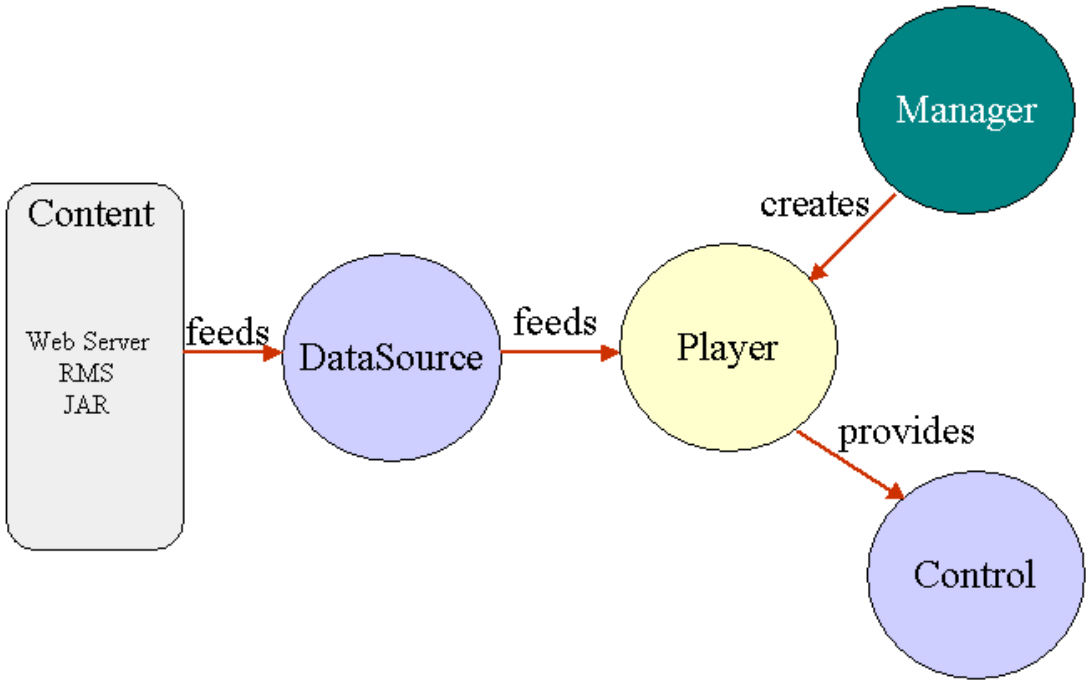

Figure 1: MMAPI Architecture.

- Extensibility. New features can be added easily without breaking the architecture. Advanced MultiMedia Supplements is an example of a rather large extension to the MMAPI.
- Subsettability. Developers can limit support to particular types of content, for example basic audio as in the MIDP 2.0 Sound API which is described later in this document.
- Low footprint. The API works within the strict memory limits of mobile devices.
- Options for implementations. The API is designed to allow some features to be left unimplemented if they for example cannot be supported by the target mobile device.

2.3 Architecture

The architecture of MMAPI is very similar to the architecture of Java Media Framework for J2SE. Four main components are `DataSource`, `Player`, `Manager` and `Control`. `DataSource`'s main purpose is to hide the details of reading and writing the actual data. This way the `Player` can concentrate on providing different media-specific `Controls`, like `VolumeControl`, for handling the media content. `Manager`'s main task is to create `Players` from different `DataSources` and also

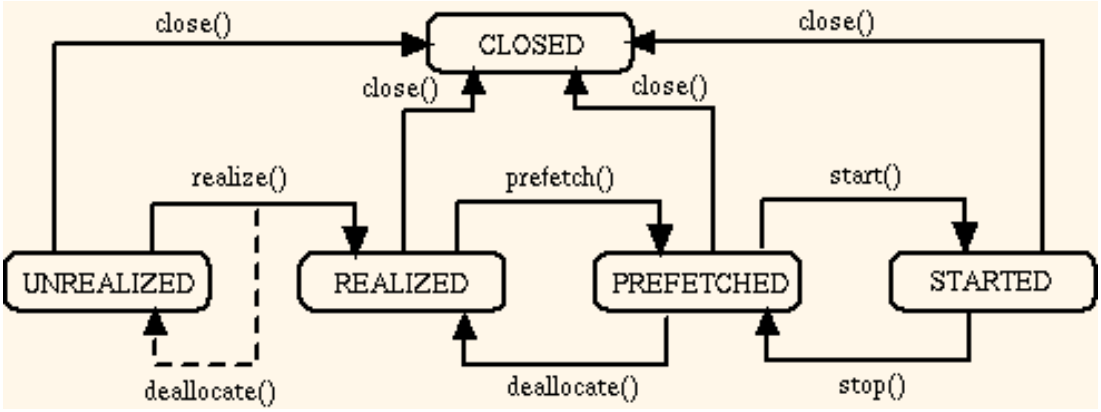

Figure 2: MMAPI Player States.

from `InputStream`s. The architecture is visualized in figure 1 on page 3. Mahmoud, 2003

The `Player`'s life-cycle consists of five states, which are `UNREALIZED`, `REALIZED`, `PREFETCHED`, `STARTED`, and `CLOSED`. The purpose of these states is to enable programmatic control over potentially time-consuming operations. When a `Player` is created, it is in the `UNREALIZED` state. A call to the `Player`'s `realize()` method transfers the `Player` to the `REALIZED` state. During this transition the `Player` initializes the resources it needs to function. The `realize()` method allows the execution of this potentially time-consuming process at an appropriate time. After the `REALIZED` state the `Player` typically moves to the `PREFETCHED` state. This transition takes care of the rest of the possible time-consuming operations reducing the startup time of the `Player` to the minimum. Finally the `Player` normally moves to the `STARTED` state. When the `Player` reaches the end of media or when it is stopped, it moves back to the `PREFETCHED` state and is ready to repeat the cycle. Calling `close()` transfers the `Player` to the `CLOSED` state. The `Player` is moved through its states using its state transition methods `realize()`, `prefetch()`, `start()`, `stop()`, `deallocate()` and `close()`. The `Player`'s life-cycle is illustrated in figure 2. Rantalahti *et al.* , 2003

2.4 MIDP 2.0 Sound API

The MIDP 1.0 specification had only minimal support for sound. The MIDP 2.0 specification extends this support by including a subset of MMAPI. The purpose of this subset is to enable more sophisticated audio features on mobile devices that do not have the resources to support the full MMAPI. The MIDP 2.0 Sound API or ABB (Audio Building Block) as it is also called is upwardly compatible with the full MMAPI. As the name states, it only supports audio related features of the MMAPI. Video capture or playback is not supported. Audio capture is not supported either. Features that are

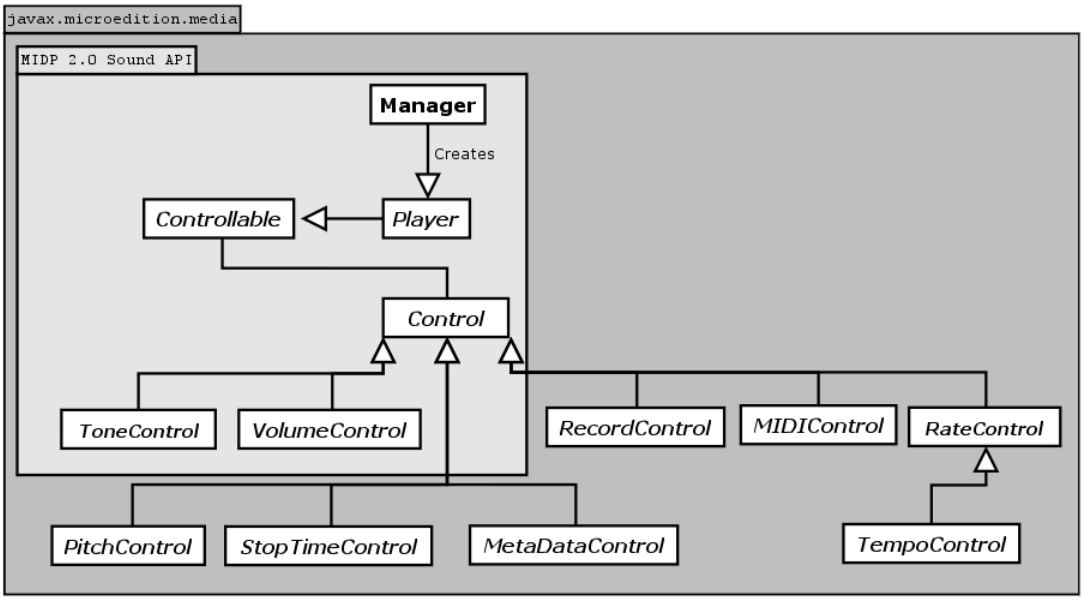

Figure 3: MMAPI and MIDP 2.0 Sound Classes.

supported are listed in the following. Riggs *et al.* , 2003; Hemphill, 2004

- Media format and protocol independency (some mandatory protocols and formats have however been defined to ensure interoperability)
- Tone generation (including single tones and tone sequences)
- Audio playback
- General media flow controls (start, stop, etc.)
- Media-specific controls (volume control, tone control)
- Ability to query for supported media features

Other main characteristics of the MIDP 2.0 Sound API are that multiple `Player`s cannot be synchronized and that the package `javax.microedition.media.protocol` including `DataSource` class is excluded. This means that custom protocols are not supported. Knudsen, 2002

In the figure 3 the relationship between MMAPI and MIDP 2.0 sound API is presented in the form of main sound classes.

2.5 Development

Basically the main tool to use for MMAPI related development as well as almost any other J2ME development is Sun Microsystems' J2ME wireless toolkit. J2ME wireless toolkit 2.2, which is currently the latest version of the toolkit, supports MMAPI 1.1. It contains several emulators that can be used to emulate MIDP applications. Some mobile device manufacturers, like Nokia, also offer their own J2ME development kits. Those development kits usually have emulators that are more alike with a certain real mobile device of the manufacturer that provides the kit. Of course, it is always recommendable to test the applications also with the targeted mobile device, since there are almost always some differences between the emulator and the actual device.

As stated earlier, the essential features of MMAPI are audio playback and audio recording. The implementation of these features is now illustrated with some simple code examples.

The following example illustrates audio playback from a network file. At first the `Manager` creates a `Player` from a given media locator. The created `Player` is then set to play the audio file three times. After that the `Player` is realized so that its `VolumeControl` can be obtained. `VolumeControl` can be used to set the volume of the audio that is played. Its method `setLevel(int level)` takes values between 0 and 100 where 0 represents silence and 100 the highest volume possible and the scaling is linear. If the `VolumeControl` can be obtained, the volume is set to 80 in this example. Finally the `Player` is started and the audio playback begins.

```
public class PlaySoundFromNetworkMIDlet extends MIDlet {
    protected void pauseApp() {}
    protected void destroyApp(boolean unconditional) {}
    protected void startApp() {
        try {
            Player player = Manager.createPlayer
                ("http://localhost/example.wav");
            player.setLoopCount(3);
            player.realize();
            VolumeControl vc;
            if ((vc = (VolumeControl) player.getControl
                    ("VolumeControl")) != null) {
                vc.setLevel(80);
            }
            player.start();
        } catch (MediaException e) {
        } catch (IOException e) {}
    }
}
```

The next example illustrates audio playback from a resource in the classpath. The

audio file can for example be included in the JAR file of the application. In this case the `Manager` creates a `Player` from a given `InputStream` and media type. This time the volume is left at the default value and there is no need to realize the `Player` separately before starting it.

```
public class PlaySoundFromSystemResourceMIDlet
        extends MIDlet {
    protected void pauseApp() {}
    protected void destroyApp(boolean unconditional) {}
    protected void startApp() {
        try {
            String type = "audio/x-wav";
            InputStream is = getClass().getResourceAsStream
                ("/res/audio/example.wav");
            Player player = Manager.createPlayer(is, type);
            player.start();
        } catch (MediaException e) {
        } catch (IOException e) {}
    }
}
```

The following example demonstrates a simple audio tone generation. The tone is generated using the method `Manager.playTone(int note, int duration, int volume)`.

```
public class PlaySingleToneMIDlet extends MIDlet {
    protected void pauseApp() {}
    protected void destroyApp(boolean unconditional) {}
    protected void startApp() {
        try {
            Manager.playTone(50, 500, 100);
        } catch (MediaException e) {}
    }
}
```

If more control over the tone generation is desired or if one wants to play a sequence of tones, one should use a `ToneControl` obtained from a `Player` as in the following example. In the example the tone sequence is first specified as a list of tone-duration pairs and user-defined sequence blocks. A tone `Player` is then created using the `Manager` after which the `Player` is realized so that its `ToneControl` can be obtained. The specified tone sequence is then set to the `ToneControl` and played by the `Player`.

```
public class PlayToneSequenceMIDlet extends MIDlet {
    protected void pauseApp() {}
    protected void destroyApp(boolean unconditional) {}
    protected void startApp() {

        byte tempo = 30;
        byte volume = 100;
        byte d = 8; // eighth note
        byte C = ToneControl.C4;
        byte D = (byte) (C + 2);
        byte E = (byte) (C + 4);

        byte[] sequence = {
            ToneControl.VERSION, 1,
            ToneControl.TEMPO, tempo,
            ToneControl.SET_VOLUME, volume,
            // define block 0 and repeatable block
            // of 3 eigth notes
            ToneControl.BLOCK_START, 0,
                C, d, D, d, E, d,
            // end block 0
            ToneControl.BLOCK_END, 0,
            // play block 0
            ToneControl.PLAY_BLOCK, 0,
            // play some other notes
            ToneControl.SILENCE, d, E, d, D, d, C, d,
            // play block 0 again
            ToneControl.PLAY_BLOCK, 0
        };

        try {
            Player player = Manager.createPlayer
                (Manager.TONE_DEVICE_LOCATOR);
            player.realize();
            ToneControl tc = (ToneControl)
                player.getControl("ToneControl");
            tc.setSequence(sequence);
            player.start();
        } catch (IOException e) {
        } catch (MediaException e) {}
    }
}
```

The next example demonstrates media synchronization. Two `Player`s are created. The first `Player` plays a midi sound file and the second `Player` an mpeg movie file. Both `Player`s are first realized so that their timebases can be set to same. After that both `Player`s are prefetched so that they can be started with minimum delay possible. Finally the `Player`s are started.

```
public class RecordAudioMIDlet extends MIDlet {
    protected void pauseApp() {}
    protected void destroyApp(boolean unconditional) {}
    protected void startApp() {
        try {
            Player player1 = Manager.createPlayer
                ("http://localhost/example.mid");
            player1.realize();
            Player player2 = Manager.createPlayer
                ("http://localhost/example.mpg");
            player2.realize();
            player2.setTimeBase(player1.getTimeBase());
            player1.prefetch();
            player2.prefetch();
            player1.start();
            player2.start();
        } catch (IOException e) {
        } catch (MediaException e) {
    }
}
```

The final example of MMAPI illustrates audio recording. At first a `Player` for audio capturing is created using the `Manager`. Then the `Player` is realized so that its `RecordControl` can be obtained. A stream to which to record is then set to the `RecordControl`. After that the recording can be started with the `RecordControl`'s method `startRecord()`. The Player has to be started as well, because the `RecordControl` records what is being played by the `Player`. In this example the method `Thread.sleep(int duration)` is used to set the duration of the recording. The recording is stopped using the `RecordControl`'s method `commit()`. The `Player` is also closed and the recorded audio is available in the stream that was set as the `RecordControl`'s output stream.

```
public class RecordAudioMIDlet extends MIDlet {
    protected void pauseApp() {}
    protected void destroyApp(boolean unconditional) {}
    protected void startApp() {
        try {
```

```
            Player player = Manager.createPlayer
                ("capture://audio");
            player.realize();
            RecordControl rc = (RecordControl)
                player.getControl("RecordControl");
            ByteArrayOutputStream recordStream =
                new ByteArrayOutputStream();
            rc.setRecordStream(recordStream);
            rc.startRecord();
            player.start();
            // record for 5 seconds
            Thread.currentThread().sleep(5000);
            rc.commit();
            player.close();
        } catch (IOException e) {
        } catch (MediaException e) {
        } catch (InterruptedException e) {}
    }
}
```

3 Advanced MultiMedia Supplements (JSR-234)

3.1 Overview

JSR-234 specifies an advanced multimedia supplements to Mobile Media API (JSR-135). It will enhance the multimedia support on mobile devices by adding features like camera control, radio tuner and even 3D audio. Of course, all this can be done on resource-constrained devices such as mobile phones and personal digital assistants (PDA). JSR-234 is still at the time of writing this document under development, but the final version of the specification should be published by January 2005. Rantalahti, 2004; Haley, 2004

In this section we will concentrate only on the sound features of JSR-234 such as different kind of additional audio effects it provides. We will also take a look in to the way that the effects are processed, since that differs quite drastically from the earlier specifications. Another very interesting feature of JSR-234 is 3D audio and virtual acoustics, so we are also going to take a brief look at that. In addition to these, a quite detailed architectural view to the API will also be presented as well as some simple code examples.

3.2 Features

3.2.1 Advanced Audio

More than a half of the new interfaces defined by JSR-234 are concentrating on advanced audio. It will define many new sound effects such as reverb and chorus. Also sound equalizer is added to the specification. This can be used for example to make music sound more natural to the listener in some certain situations.

The most interesting new feature in JSR-234 is probably the possibility to create three-dimensional sound effects. This makes it possible to create virtual acoustic environments to the listener that is using a device equipped with either earphones or loudspeakers. In practice this means that it is possible to create sounds that from the listener's perspective seem to be coming from different directions and different distances. Human ear can perceive direction from where the sound is coming very accurately using certain physical properties of the sound arriving to the ears. These physical properties can be simulated quite efficiently using computers, thus creating a feeling to the listener that the sound is coming from some certain direction. A more detailed explanation on how this is implemented in JSR-234 can be found from section 3.3.4 of this document.

3.2.2 Radio

JSR-234 also defines controls for AM/FM radio tuner. Of course presuming that the device in question has hardware implemented to support this also. It is possible to seek channels and save them as presets as in normal radio devices. JSR-234 supports also Radio Data System (RDS) and this gives lots of additional features to regular radio. With RDS it is possible for the radio broadcasters to send signals to radio tuner and it can for example automatically tune to a channel that is broadcasting important traffic announcements.

3.3 Architecture

JSR-234 is built on Mobile Media API, so it automatically inherits its structure. Mobile Media API is described in more detail in section 2.3 of this document. The concept of `Player`, `Manager` and `Control` is hereafter used in JSR-234 also. In addition to more sophisticated audio effect controls, which we will deal later in this chapter, JSR-234 adds `GlobalManager`, `Spectator`, `Module`, `EffectModule`, `SoundSource3D` and `MediaProcessor` classes to the Mobile Media API specification. In the figure 4 the relationship between AMMS, MMAPI and MIDP 2.0 sound API is presented in the form of main sound classes.

`GlobalManager` class is used to create `EffectModule`s, `SoundSource3D`s and `MediaProcessor`s. This is pretty much the same concept as in the `Manager` class that is used to create `Player`s. Furthermore a `Spectator` class can be fetched from `GlobalManager`.

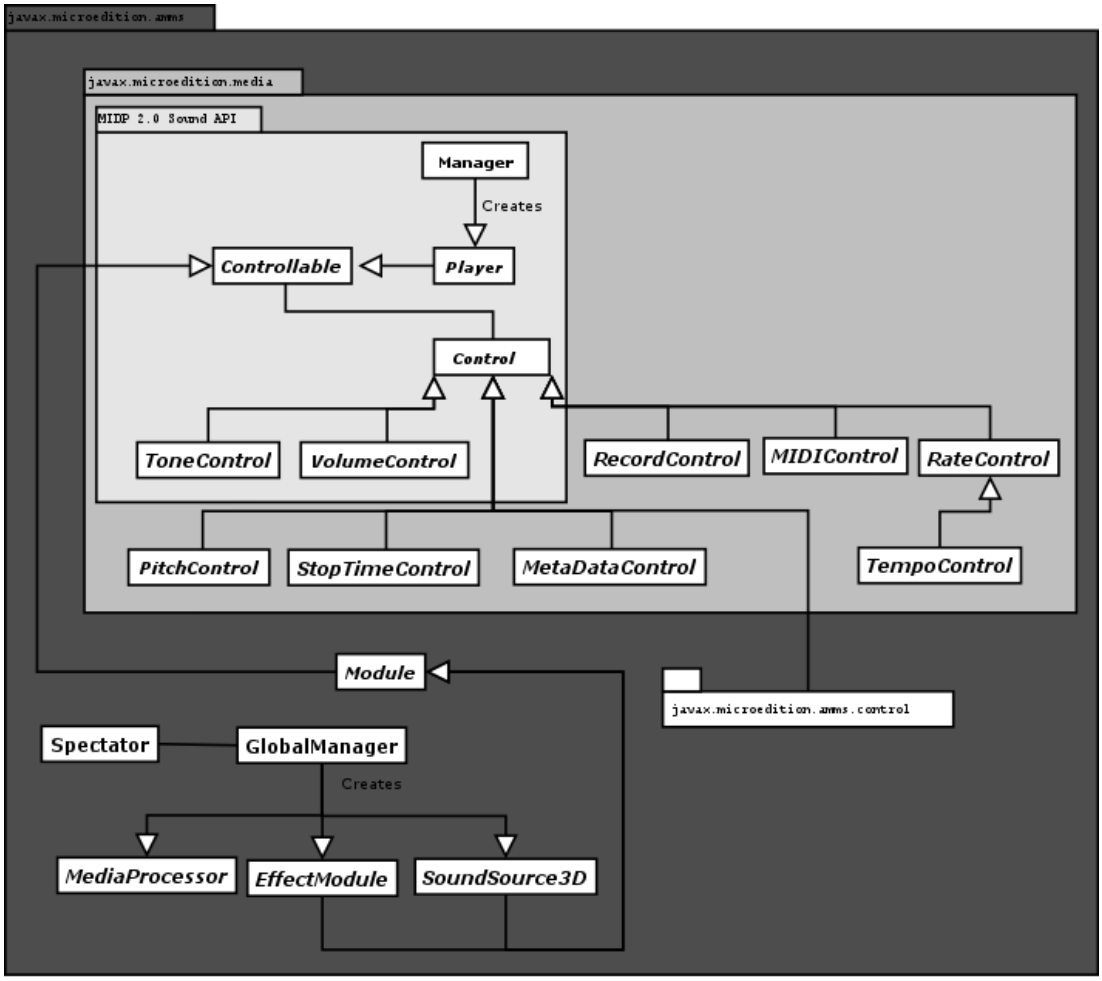

Figure 4: AMMS, MMAPI and MIDP 2.0 Sound Classes.

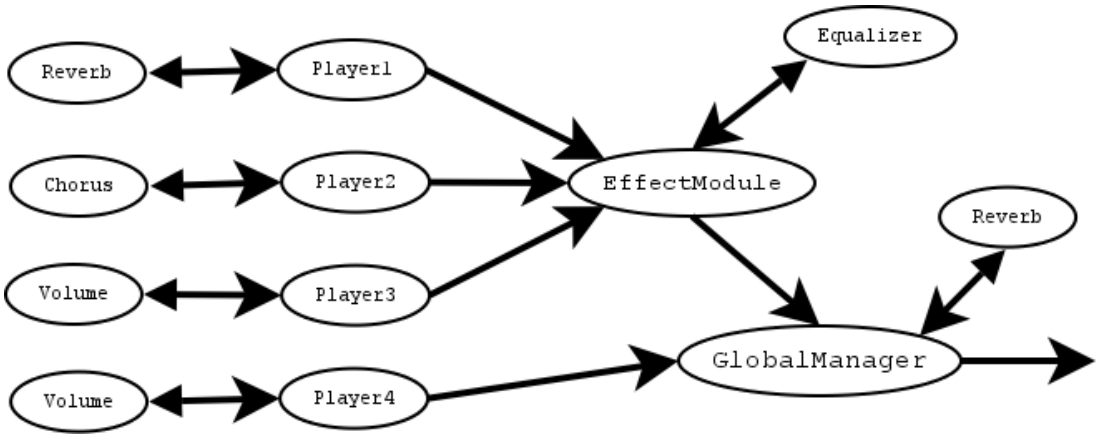

Figure 5: JSR-234 Effects Network.

The idea behind `GlobalManager` is that it serves as a manager to all `Player` objects in the application. `Player`s are usually grouped under some certain `Module`. `Module`s itself cannot be fetched from anywhere but its subclasses `EffectModule` and `SoundSource3D` are used instead. These two classes operate almost exactly the same way. The important difference between them is that `EffectModule` is meant for effects in general whereas `SoundSource3D` is meant only for three-dimensional audio effects. This is done only to keep the implementation as simple as possible.

3.3.1 Effects Network

JSR-234 provides means for specifying effects and their order. This can be done by grouping `Player`s, adding effects after a group of `Player`s or after each `Player` individually and optionally, even the order of the effects applied can be specified. This can be called as the effects network. Essential parts of this network are the `EffectModule` and `SoundSource3D` classes introduced in the previous chapter. Visualization of how the effects network works can be seen in figure 5. The order in which the individual effects are applied can be defined using `EffectOrderControl` interface. It simply assigns different priorities to different effects.

3.3.2 Media Post-processing

`MediaProcessor` interface is used to handle post-processing of media. It also can be retrieved using `Globalmanager`. Using a `MediaProcessor` it is possible to specify source media and several post-processing effects to be applied on it. In other words, it gives a possibility to reuse the effects network. `MediaProcessor` has three different states: `SETUP`, `STARTED` and `STOPPED`. When `MediaProcessor` is in `SETUP` state it is possible to define the `Control`s applied to the media that will be processed. Of course, also the input and output of the processor need to be set in

SETUP state. Input to the MediaProcessor is given in InputStream and output is written to OutputStream. Before processing can start MediaProcessor needs to read some data from the start of the stream to setup the different Controls. For example, the header of the file may contain information about the format that is used. Processing can be started using MediaProcessor's start() method after which the processor moves to STARTED state. After processing has started the changes that take place in defined Controls don't affect the MediaProcessor. This is the same also for the STOPPED state where the processor goes after the method stop() is called. The changes will take effect only after the processor has returned to SETUP stage. MediaProcessor will return to SETUP state after the processing has completed or some error interrupts it.

Using MediaProcessor it is also possible to convert media to different formats. This can be achieved using the FormatControl and in the case of audio, its subclass AudioFormatControl.

3.3.3 Media Player Priorities

In JSR-234 specification it is possible to assign different priorities to Player objects as well as for the effects. Player prioritization is done to assure the proper functioning of the application in the case where numerous Players are used.

When a Player is making a transition from REALIZED to PREFETCHED or from PREFETCHED to STARTED and there are no resources left to perform the transition, prioritization is used to determine which Player can use the resources of the device. If a Player with lower priority than the one that is making the upward state transition is using the resources needed, the lower priority Player will send DEVICE_UNAVAILABLE event to its PlayerListener and return back to previous state, this way giving the resources to the higher priority Player. If a Player is trying to do a transition when there are no resources available and the Player has equal or lower priority than the Players using the resources, the transition simply cannot be done.

3.3.4 Spatialization and Virtual Acoustics

JSR-234 specifies a LocationControl interface that can be used to place different auditory objects, usually Spectator or SoundSource3D, around the virtual acoustic space. Spectator, that can be again fetched from the GlobalManager, represents the human listener. Location in 3D space is expressed using XYZ coordinates. Since sounds coming from different distances sound different because of varying attenuation the location information is not adequate. We need also the distance, so the location is noted using a location vector.

In addition to defining the location of the sound source, it is also possible to define the orientation of the sound source in three-dimensional space. This can be accomplished via OrientationControl interface. OrientationControl can again

be used with the `Spectator` or `SoundSource3D`. In the latter case the subclasses `DirectivityControl` and `MacroscopicControl` are used.

3.3.5 Commit

One main feature of JSR-234 that also has to be mentioned is the `CommitControl`. It is a mechanism that enables one to commit many audio parameters at the same time. If it is supported, it can be obtained from `GlobalManager`. `CommitControl` has two different modes: *immediate* and *deferred*. The first one basically means that the `CommitControl` is disabled and the other one means that all audio effects that are applied will be buffered and activated only when the `commit()` method of the `CommitControl` is called. This is done in order to make synchronization easier and also to improve the performance of the applications.

3.4 Development

In this section we will concentrate on code examples which are based on the proposed final draft of the JSR-234 API. Since a final version of the API has not been published yet, we are not able to test or actually implement the examples we present here, so they might not be totally accurate. The idea that is behind this API should however become clearer through these examples.

3.4.1 Tuner and RDS Example

In this first very simple example we show how it is possible to control the radio tuner using classes defined in `javax.microedition.amms.control.tuner` package of JSR-234 API.

It all starts simply by creating a `Player` that uses the device's radio tuner hardware as its input. This is done with the following code.

```
Player player = Manager.createPlayer("capture://radio");
```

Then we need to get the `Control` of the tuner so we can actually control the radio. Here we use the `TunerControl` class that naturally has to be supported by the device.

```
TunerControl tuner = (TunerControl) player.getControl
    ("javax.microedition.media.control.tuner.TunerControl");
```

Now using the `TunerControl` we can for example search for some radio station and save it as a preset.

```
int freqFound = tuner.seek
    (970000, TunerControl.MODULATION_FM, true);
tuner.setStereoMode(TunerControl.STEREO);
tuner.setPreset(1);
tuner.setPresetName(1, "Station 1");
```

In the previous code sample we started to search an FM radio channel from 97 kHz and up. Then we stored it to the preset slot number one.

If `RDSControl` is supported by the device, we can use RDS to get some channel specific data. Next we will use RDS to get the current date, set the traffic announcements on and get the current program type.

```
if ((RDSControl rds = (RDSControl) radio.getControl
    ("javax.microedition.media.control.tuner.RDSControl")
    )!= null) {
    Date date = rds.getCT();
    if (rds.getTA()) {
        rds.setAutomaticTA(false);
    }
    String pty = rds.getPTYString(true);
}
```

3.4.2 3D Audio Example

This example shows how to create three-dimensional sound sources using JSR-234 API. This example is very illustrative because it also shows how the effects network is used via `EffectModule` and `SoundSource3D`. In the code below we first add a `Player` to the `SoundSource3D`. This then enables us to set the `Player`'s position in virtual acoustic environment using `LocationControl`. We are going to set the location of the sound source to be 10 meters in front, meaning that it resides on the negative Z-axis.

```
SoundSource3D src = GlobalManager.createSoundSource3D();
src.addPlayer(p1);
LocationControl ls = (LocationControl) src.getControl
    ("javax.microedition.media.control.audio3d.LocationControl");
ls.setCartesian(0, 0, -10000);
```

Next we set the way the sound attenuates when it travels from some location to the position where the listener is. This is done using `DistanceAttenuationControl`. The sound attenuates exponentially when moving away from the listener. We set the sound to be totally silent after 50 meters.

```
DistanceAttenuationControl distSrc =
    (DistanceAttenuationControl) src.getControl
    ("javax.microedition.media.control.audio3d." +
     "DistanceAttenuationControl");
distSrc.setParameters(10, 50000, true, 1000);
```

Now we are going to use `EffectModule` to set some 2D effect. We are going to set a flanger effect to the sound using `ChorusControl`.

```
EffectModule effect = GlobalManager.createEffectModule();
effect.addPlayer(p2);
effect.addPlayer(p3);
if ((ChorusControl chorusEffect =
    (ChorusControl) effect.getControl
    ("javax.microedition.media.control.audioeffect.ChorusControl")
    ) != null) {
    chorusEffect.setPreset("flanger");
    chorusEffect.setModulationDepth(4000);
    chorusEffect.setModulationRate(260);
}
```

It is also possible to set the location of the spectator using `LocationControl` and the orientation using the `OrientationControl`. In the following example we will set the `Spectator` to the origin, which actually is not needed since it is the default value. Then we will slightly turn the spectator so the rotation is ten degrees on the Y-axis, zero degrees on the X-axis and five degrees on the Z-axis.

```
Spectator s = GlobalManager.getSpectator();
LocationControl lc = (LocationControl) s.getControl
    ("javax.microedition.media.control.audio3d.LocationControl");
lc.setCartesian(0, 0, 0);
OrientationControl oc = (OrientationControl) s.getControl
    ("javax.microedition.media.control.audio3d.OrientationControl");
oc.setOrientation(10, 0, 5);
```

4 Support on Current Mobile Devices

Today the biggest mobile phone manufacturers such as Nokia, Siemens and Sony-Ericsson have quite good multimedia support in their latest phone models. MMAPI is included in almost all new mobile devices that have some kind of multimedia features. The level of `Control`s that these devices support varies based on different content types. For example, a whole lot of `Control`s are supported for content types

Manufacturer	Model	Content type	Supported Controls
Nokia	Series 40 DP 1.0, Series 40 DP 2.0	audio/x-tone-seq	VolumeControl, StopTimeControl, ToneControl
		audio/midi, audio/sp-midi	VolumeControl, StopTimeControl, MIDIControl, TempoControl, PitchControl
	Series 60 DP 2.0	audio/x-tone-seq	VolumeControl, StopTimeControl, ToneControl
		audio/wav, audio/x-wav, audio/au, audio/x-au, audio/amr, audio/amr-wb	VolumeControl, StopTimeControl
		audio/midi, audio/sp-midi	VolumeControl, StopTimeControl
Siemens	Siemens SX1	audio/x-tone-seq	VolumeControl, StopTimeControl, ToneControl
		audio/x-wav, audio/amr, audio/midi, audio/sp-midi	VolumeControl, StopTimeControl
Sony-Ericsson	All models with MMAPI support	audio/x-tone-seq	VolumeControl, StopTimeControl, ToneControl
		audio/x-wav, audio/amr, audio/midi, audio/sp-midi	VolumeControl, StopTimeControl

Table 1: Supported Controls on Current Mobile Phones.

`audio/midi` and `audio/sp-midi` whereas basically only `VolumeControl` and `StopTimeControl` are supported for `audio/wav` content type. Naturally this is because of the difficulty in implementing some more complicated `Control`s for sampled audio and of course the devices' resources are still somewhat limited as well. Table 1 on page 18 shows the controls that are implemented in mobile phones of some today's biggest mobile phone manufacturers. Hui, 2004; Polish, 2004; Nokia, 2004; Siemens, 2003; Sony-Ericsson, 2004

5 Possible Future Supplements to MMAPI

The fact that the MMAPI has a very similar structure with the Java Media Framework for J2SE suggests that one could find out possible future supplements to MMAPI by comparing these two technologies. However, JMF does not seem to contain any major elements that are not in the MMAPI/AMMS. Of course, there are differences that arouse from the fact that JMF has been designed for desktop devices and MMAPI for mobile devices. For example, RTP is a large part of JMF, but is not included at all in the MMAPI. This decision has been made according to the protocol independency principle and it is vital in allowing MMAPI to be implemented on all kinds of different mobile devices. Therefore, it is not likely that the RTP support feature will be a future supplement to MMAPI. On the other hand, MMAPI also contains some elements that are not so well handled in the JMF. For example, the support for MIDI is a key part of the MMAPI but is not especially supported in the JMF. As a summary, we do not see any major supplements to the MMAPI in the near future after the AMMS supplement.

6 Conclusions

In this paper the support for sound on current mobile devices was discussed by going through the MIDP 2.0 sound API and JSR-135/234 standards. We found out that the MIDP 2.0 sound API contains the basic audio features including audio playback and tone generation while the JSR-135 also known as MMAPI adds some more sophisticated features, most importantly audio recording, to it. We took a glance on the support of these technologies on current mobile devices and found out that most of the new devices on market already support MMAPI. We also got acquainted with the new standard JSR-234 also known as AMMS that adds a bunch of even more sophisticated audio features to MMAPI. In our opinion, AMMS is such a large extension to MMAPI that most likely it will take some time before the real mobile devices on market will have the resources to be able to fully support all of its possibilities.

REFERENCES

Haley Dan. 2004 (August). *Get ready for advanced multimedia on your Java mobile platform - A tour of the features in the upcoming Advanced Multimedia*

Supplements for J2ME API. http://www.javaworld.com/javaworld/jw-08-2004/jw-0823-multimedia.html.

Hemphill David. 2004 (May). *Exploring the J2ME Mobile Media APIs*. http://www.devx.com/wireless/Article/20911.

Hui Ben. 2004 (October). *MIDP 2.0 Phones and PDAs*. http://www.benhui.net/modules.php?name=Midp2Phones.

Knudsen Jonathan. 2002 (June). *Mobile Media API Overview*. http://developers.sun.com/techtopics/mobility/apis/articles/mmapi_overview/.

Mahmoud Qusay H. 2003 (June). *The J2ME Mobile Media API*. http://developers.sun.com/techtopics/mobility/midp/articles/mmapioverview/.

Nokia. 2004 (May). *Mobile Media API Implementation In Nokia Developer Platforms*. http://www.forum.nokia.com/main/1,6566,21,00.html?fsrParam=1-3-/main/0,,21_20,00.html&fileID=4992.

Polish J2ME. 2004 (November). *MIDP 2.0 Devices*. http://www.j2mepolish.org/devices/midp2.html.

Rantalahti Antti. 2004 (October). *JSR 234: Advanced Multimedia Supplements*. http://www.jcp.org/en/jsr/detail?id=234.

Rantalahti Antti & Huopaniemi Jyri. 2003 (May). *Birth of Mobile Java Multimedia*. http://www.nokia.com/nokia/0,,53719,00.html.

Rantalahti Antti, Yli-Nokari Jyrki & Huopaniemi Jyri. 2003 (June). *JSR 135: Mobile Media API*. http://jcp.org/en/jsr/detail?id=135.

Riggs Roger, Taivalsaari Antero, Peursem Jim Van, Huopaniemi Jyri, Patel Mark, Uotila Aleksi & Holliday Jim. 2003. *Programming Wireless Devices with the Java 2 Platform, Micro Edition, Second Edition*. Addison Wesley.

Siemens. 2003 (Nov). *Siemens SX1, Techical Note for Application Developers*. https://communication-market.siemens.de/portal/main.aspx?pid=1&langid=0.

Sony-Ericsson. 2004 (July). *Java J2ME for Sony Ericsson Mobile Phones*. http://developer.sonyericsson.com/getDocument.do?docId=65067.