APPENDIX A

Mobile Media API (MMAPI) Reference

This appendix is a reference to the MMAPI. All classes, interfaces, and exceptions are listed here so that you can find them in one easy-to-use place. The Javadocs, however, remain the best place for getting detailed explanation about each method and class and is used in this appendix to take the one-line explanation of each class, interface, and exception.

This API is listed alphabetically, grouped by package.

Package `javax.microedition.media`

Interface `Control`

A `Control` object is used to control some media processing functions.

```
public interface Control {
  // no methods
}
```

Interface `Controllable`

`Controllable` provides an interface for obtaining the controls from an object such as a `Player`.

```
public interface Controllable {
  // methods
  public Control getControl(java.lang.String controlType);
  public Control[] getControls();
}
```

Class Manager

Manager is the access point for obtaining system-dependent resources such as Player instances for multimedia processing.

```
public final class Manager {
  // Constructors
  public static final java.lang.String MIDI_DEVICE_LOCATOR;
  public static final java.lang.String TONE_DEVICE_LOCATOR;

  // Methods
  public static Player createPlayer(java.lang.String locator)
    throws java.io.IOException, MediaException;
  public static Player createPlayer(
      java.io.InputStream stream, java.lang.String type)
        throws java.io.IOException, MediaException;
  public static Player createPlayer(DataSource source)
    throws java.io.IOException, MediaException;
  public static java.lang.String[] getSupportedContentTypes(
    java.lang.String protocol);
  public static java.lang.String[] getSupportedProtocols(
    java.lang.String content_type);
  public static TimeBase getSystemTimeBase();
  public static void playTone(int note, int duration, int volume)
    throws MediaException;
}
```

Class MediaException

A MediaException indicates an unexpected error condition in a method.

```
public class MediaException extends java.lang.Exception {
  // Constructors
  public MediaException();
  public MediaException(java.lang.String reason);
}
```

Interface Player

Player controls the rendering of time-based media data.

```
public interface Player extends Controllable {
  // Constants
  public static final int CLOSED;
  public static final int PREFETCHED;
  public static final int REALIZED;
  public static final int STARTED;
  public static final long TIME_UNKNOWN;
  public static final int UNREALIZED;
```

```
  // Methods
  public void addPlayerListener(PlayerListener playerListener);
  public void close();
  public void deallocate();
  public java.lang.String getContentType();
  public long getDuration();
  public long getMediaTime();
  public int getState();
  public TimeBase getTimeBase();
  public void prefetch() throws MediaException;
  public void realize() throws MediaException;
  public void removePlayerListener(PlayerListener playerListener);
  public void setLoopCount(int count);
  public long setMediaTime(long now) throws MediaException;
  public void setTimeBase(TimeBase master) throws MediaException;
  public void start() throws MediaException;
  public void stop() throws MediaException;
}
```

Interface `PlayerListener`

`PlayerListener` is the interface for receiving asynchronous events generated by `Player` instances.

```
public interface PlayerListener {
  // Constants
  public static final java.lang.String BUFFERING_STARTED;
  public static final java.lang.String BUFFERING_STOPPED;
  public static final java.lang.String CLOSED;
  public static final java.lang.String DEVICE_AVAILABLE;
  public static final java.lang.String DEVICE_UNAVAILABLE;
  public static final java.lang.String DURATION_UPDATED;
  public static final java.lang.String END_OF_MEDIA;
  public static final java.lang.String ERROR;
  public static final java.lang.String RECORD_ERROR;
  public static final java.lang.String RECORD_STARTED;
  public static final java.lang.String RECORD_STOPPED;
  public static final java.lang.String SIZE_CHANGED;
  public static final java.lang.String STARTED;
  public static final java.lang.String STOPPED;
  public static final java.lang.String STOPPED_AT_TIME;
  public static final java.lang.String VOLUME_CHANGED;

  // Methods
  public void playerUpdate(Player player, java.lang.String event,
                                          java.lang.Object eventData);
}
```

Interface TimeBase

A TimeBase is a constantly ticking source of time.

```
public interface TimeBase {
  // Methods
  public long getTime();
}
```

Package javax.microedition.media.control

Interface FramePositioningControl

The FramePositioningControl is the interface to control precise positioning of a video frame for Player instances.

```
public interface FramePositioningControl extends Control {
  // Methods
  public long mapFrameToTime(int frameNumber);
  public int mapTimeToFrame(long mediaTime);
  public int seek(int frameNumber);
  public int skip(int framesToSkip);
}
```

Interface GUIControl

GUIControl extends Control and is defined for controls that provide GUI functionalities.

```
public interface GUIControl extends Control {
  // Constants
  public static final int USE_GUI_PRIMITIVE;

  // Methods
  public java.lang.Object initDisplayMode(int mode, java.lang.Object arg);
}
```

Interface MetaDataControl

MetaDataControl is used to retrieve metadata information included within the media streams.

```
public interface MetaDataControl extends Control {
  // Constants
  public static final java.lang.String AUTHOR_KEY;
  public static final java.lang.String COPYRIGHT_KEY;
  public static final java.lang.String DATE_KEY;
  public static final java.lang.String TITLE_KEY;
```

```
  // Methods
  public java.lang.String[] getKeys();
  public java.lang.String getKeyValue(java.lang.String key);
}
```

Interface MIDIControl

MIDIControl provides access to MIDI rendering and transmitting devices.

```
public interface MIDIControl extends Control {
  // Constants
  public static final int CONTROL_CHANGE;
  public static final int NOTE_ON;

  // Methods
  public int[] getBankList(boolean custom) throws MediaException;
  public int getChannelVolume(int channel);
  public java.lang.String getKeyName(int bank, int prog, int key)
    throws MediaException;
  public int[] getProgram(int channel) throws MediaException;
  public int[] getProgramList(int bank) throws MediaException;
  public java.lang.String getProgramName(int bank, int prog) throws MediaException;
  public boolean isBankQuerySupported();
  public int longMidiEvent(byte[] data, int offset, int length);
  public void setChannelVolume(int channel, int volume);
  public void setProgram(int channel, int bank, int program);
  public void shortMidiEvent(int type, int data1, int data2);
}
```

Interface PitchControl

PitchControl raises or lowers the playback pitch of audio without changing the playback speed.

```
public interface PitchControl extends Control {
  // Methods
  public int getMaxPitch();
  public int getMinPitch();
  public int getPitch();
  public int setPitch(int millisemitones);
}
```

Interface RateControl

RateControl controls the playback rate of a Player instance.

```
public interface RateControl extends Control {
  // Methods
  public int getMaxRate();
  public int getMinRate();
```

```
  public int getRate();
  public int setRate(int millirate);
}
```

Interface RecordControl

RecordControl controls the recording of media from a Player instance.

```
public interface RecordControl extends Control {
  // Methods
  public void commit() throws java.io.IOException;
  public java.lang.String getContentType();
  public void reset() throws java.io.IOException;
  public void setRecordLocation(java.lang.String locator)
    throws java.io.IOException, MediaException;
  public int setRecordSizeLimit(int size) throws MediaException;
  public void setRecordStream(java.io.OutputStream stream);
  public void startRecord();
  public void stopRecord();
}
```

Interface StopTimeControl

StopTimeControl allows one to specify a preset stop time for a Player instance.

```
public interface StopTimeControl extends Control {
  // Constants
  public static final long RESET;

  // Methods
  public long getStopTime();
  public void setStopTime(long stopTime);
}
```

Interface TempoControl

TempoControl controls the tempo, in musical terms, of a song.

```
public interface TempoControl extends RateControl {
  // Methods
  public int getTempo();
  public int setTempo(int millitempo);
}
```

Interface ToneControl

ToneControl is the interface to enable playback of a user-defined monotonic tone sequence.

```
public interface ToneControl extends Control {
  // Constants
  public static final byte BLOCK_END;
  public static final byte BLOCK_START;
  public static final byte C4;
  public static final byte PLAY_BLOCK;
  public static final byte REPEAT;
  public static final byte RESOLUTION;
  public static final byte SET_VOLUME;
  public static final byte SILENCE;
  public static final byte TEMPO;
  public static final byte VERSION;

  // Methods
  public void setSequence(byte[] sequence);
}
```

Interface VideoControl

VideoControl controls the display of video.

```
public interface VideoControl extends GUIControl {
  // Constants
  public static final int USE_DIRECT_VIDEO;

  // Methods
  public int getDisplayHeight();
  public int getDisplayWidth();
  public int getDisplayX();
  public int getDisplayY();
  public byte[] getSnapshot(java.lang.String imageType) throws MediaException;
  public int getSourceHeight();
  public int getSourceWidth();
  public java.lang.Object initDisplayMode(int mode, java.lang.Object arg);
  public void setDisplayFullScreen(boolean fullScreenMode) throws MediaException;
  public void setDisplayLocation(int x, int y);
  public void setDisplaySize(int width, int height) throws MediaException;
  public void setVisible(boolean visible);
}
```

Interface VolumeControl

VolumeControl is an interface for manipulating the audio volume of a Player instance.

```
public interface VolumeControl extends Control {
  // Methods
  public int getLevel();
  public boolean isMuted();
  public int setLevel(int level);
  public void setMute(boolean mute);
}
```

Package javax.microedition.media.protocol

Class ContentDescriptor

A ContentDescriptor identifies media data containers.

```
public class ContentDescriptor {
  // Constructors
  public ContentDescriptor(java.lang.String contentType);

  // Methods
  public java.lang.String getContentType();
}
```

Class DataSource

A DataSource is an abstraction for media protocol handlers.

```
public abstract class DataSource implements Controllable {
  // Constructors
  public DataSource(java.lang.String locator);

  // Methods
  public abstract void connect() throws java.io.IOException;
  public abstract void disconnect();
  public abstract java.lang.String getContentType();
  public java.lang.String getLocator();
  public abstract SourceStream[] getStreams();
  public abstract void start() throws java.io.IOException;
  public abstract void stop() throws java.io.IOException;
}
```

Interface SourceStream

Abstracts a single stream of media data.

```
public interface SourceStream extends Controllable {
  // Constants
  public static final int NOT_SEEKABLE;
  public static final int RANDOM_ACCESSIBLE;
  public static final int SEEKABLE_TO_START;

  // Methods
  public ContentDescriptor getContentDescriptor();
  public long getContentLength();
  public int getSeekType();
  public int getTransferSize();
  public int read(byte[] b, int off, int len) throws java.io.IOException;
  public long seek(long where) throws java.io.IOException;
  public long tell();
}
```

APPENDIX B

URI Syntax for Media Locators

The Uniform Resource Identifier (URI) syntax is described using a Request For Comment (RFC) paper as a universal means of describing the location of different types and kinds of resources. The RFC in question is 2396 and its full text can be accessed at *http://www.ietf.org/rfc/rfc2396.txt*.

Essentially, this RFC defines a way for providing a uniform and consistent way of describing the location of different types of resources, wherever they may be located. This is important to us in MMAPI, because MMAPI, being protocol and format agnostic, requires consistent means to access resources.

URI defines two parts to each identifier string: the scheme part and the scheme specific part separated by a `':'` as shown here:

```
<scheme>:<scheme-specific-part>
```

As it applies to MMAPI, the scheme part is the protocol over which the resource is being referenced, whereas the scheme-specific part is a hierarchical, location-specific string, which tells the system the exact physical location of the resource.

Not surprisingly, the most common example of this is when resources are accessed over the Internet using HTTP. As an example, *http://www.ietf.org/rfc/rfc2396.txt* consists of the scheme "http" and the scheme-specific part of "//www.ietf.org/rfc/rfc2396.txt". The hierarchies in the scheme-specific parts are separated by the "/" character. Thus, `rfc2396.txt` is a physical document located in the `rfc` folder on the `www.ietf.org` server.

Of course, with MMAPI, the scheme part can be an entire range of options. As you have seen in this book, the scheme part can be any one of `capture`, `rtp`, `file`, `resource`, and, of course, `http` strings.

With each scheme part, different options for the scheme-specific part can be set. For capture scheme, you have used the `audio` and `video` scheme-specific parts. These can be further qualified to specify attributes, for example: `capture://audio?encoding=pcm` would give you a specific URI for capturing WAV audio files.

APPENDIX C

Advanced Multimedia Supplements—JSR 234

The Java Community Process (JCP) at *http://www.jcp.org* defines new technology for the Java technology using a community process, whereby new specifications are released for public overview and debated on. Once approved, the specs are introduced in the Java language. One of the most prolific areas of technology development is Java ME, and new APIs are being constantly created and released to target this area of Java development.

Specifications are released with a Java Specification Request (JSR) number. MMAPI was released as JSR 135 (*http://www.jcp.org/en/jsr/detail?id=135*). However, this specification is now more than two years old, and newer advances in technology have necessitated an update. JSR 234 (*http://www.jcp.org/en/jsr/detail?id=234*), titled Advanced Multimedia Supplements, is a step in updating the MMAPI for new technology devices. The final release of this specification was in June 2005.

However, note that JSR 234 is a supplement to JSR 135. Thus, it doesn't replace MMAPI nor is it used on its own, but provides supplementary implementations for newer technology devices. This appendix gives you an overview of the new JSR.

Introduction to JSR 234

JSR 234 was initiated by a Nokia request to create an API that could provide richer experiences to users of mobile devices. Mobile devices are increasing in their processing power capacity, so this enables sophisticated media processing functionalities available to MIDlet developers and distributors. Specifically, better controls for camera, radio and audio, and sophisticated MIDlets created based on them is the target aim of this specification.

The specification creates a new package, `javax.microedition.amms`, where `amms` stands for Advanced Multimedia supplements. The specification mostly defines new controls that are derived from the `javax.microedition.media.control.Control` interface, thus making these controls compatible with MMAPI (as you would expect, as JSR 234 is a supplement to JSR 135). It also defines four new concepts in the main `javax.microedition.amms` package, `GlobalManager`, `Spectator`, `Module`, and `MediaProcessor`.

GlobalManager

`GlobalManager` is like a `Manager` class for JSR 234. It provides methods to retrieve the specialized objects present in this specification. However, it doesn't replace or extend the `Manager` class, and you still use the `Manager` class for creating and managing `Player` instances.

Spectator

The `Spectator` class represents a virtual listener. It provides acoustic controls that can help create MIDlet applications that mimic sound coming from different locations and areas. You can create a `Spectator` class by using the `getSpectator()` static method in the `GlobalManager` class. This class implements the `Controllable` interface, which allows it to provide implementation of the `getControl()` and the `getControls()` methods.

Module

The `Module` interface is a logical grouping of multiple `Player` instances and/or MIDI channels. This functionality was missing from MMAPI, where you can treat a group of such instances as one entity. This interface is extended by two subinterfaces, `EffectsModule` and `SoundsSource3D`, which represent different types of groupings. `EffectsModule` represents `Player` instances and/or MIDI channels, to which a common effect can be applied, whereas `SoundsSource3D` represents this group in a virtual acoustical space.

Modules can be fetched from the `GlobalManager` class using the `createEffectModule()` or the `createSoundSource3D()` methods. Note that the `Module` interface also implements the `Controllable` interface.

MediaProcessor

`MediaProcessor` is an interface for postprocessing of media data. A `MediaProcessor` works very much like a `Player` interface and defines various states that mimic it as well. A `MediaProcessor` instance can be in one of UNREALIZED, REALIZED, STARTED, or STOPPED states.

You can retrieve an instance of `MediaProcessor` by using the `GlobalManager` class's static method `createMediaProcessor(String type)`, passing to it the MIME type of target processor.

This interface also extends the `Controllable` interface.

Controls

The rest of the interfaces in this API are various controls and are divided in various packages depending on the type of functionality that they provide.

Package javax.microedition.amms.control

There are two main control interfaces in this package that impact the rest of the packages as well.

`FormatControl` is used for specifying the format of the various media types. This interface is implemented by four different types of media formats using their own interfaces: `AudioFormatControl`, `ContainerFormatControl`, `ImageFormatControl`, and `VideoFormatControl`.

`EffectControl` is used to preset a filter on some media data to provide different types of effects. This interface provides a method, `setEnabled(boolean enable)`, to turn the effects on or off. This interface is supported with the `EffectOrderControl` interface, which controls the order in which multiple effects are applied.

Finally, this package contains the interface `PanControl`, which controls the panning of audio and video media data, and `PriorityControl`, which allows the manipulation of the priority given to various `Player` instances in a group situation.

Packages `javax.microedition.amms.control.imageeffect` and `javax.microedition.amms.control.audioeffect`

All of the interfaces in these two packages extend the `EffectControl` interface and provide various effects that can be applied to audio and image media data.

Package `javax.microedition.amms.control.audio3d`

All of the interfaces in this package provide controls for manipulation of audio in a 3D environment. The controls are retrieved from either the `SoundSource3D` or `Spectator` class, as the case may be.

Package `javax.microedition.amms.control.camera`

The controls in this package are retrieved from a `Player` instance that is created by using the `Manager.createPlayer("capture://camera")` method. These controls provide control over the exposure, the flash, focus, zoom, and burst shooting of snapshots (as opposed to single snapshots using the `getSnapshot()` method of the `VideoControl` interface provided in MMAPI). A `CameraControl` control provides further control over the camera by providing methods for shutter control, camera rotation, and so on.

Package `javax.microedition.amms.control.tuner`

This package contains two control interfaces for radio tuning. `RDSControl` is used for Radio Data System (RDS) tuning and provides methods to manage that. `TunerControl` isn't specifically targeted for radio tuning, but it can be used to tune into any `Player` instance that supports tuning, however unlikely.

JSR 234 Implementations

At this point, being a relatively new API, there are no known implementations on any devices, and neither is there any known emulator support. However, Nokia provides a reference implementation with some examples, which can be downloaded at *http://forum.nokia.com/java/jsr234*.

Index

C

Find it faster at http://superindex.apress.com/

D

E

F

G

Find it faster at http://superindex.apress.com/

K

L

M

Find it faster at http://superindex.apress.com/

Find it faster at http://superindex.apress.com/

Q

R

S

Find it faster at http://superindex.apress.com/

T

U

V

W

Find it faster at http://superindex.apress.com/