



September 2003

Making the Most of the Mobile Media API (JSR-135) with Sony Ericsson handsets



Sony Ericsson T610/T616/T618



Sony Ericsson Z600

Contents

1. Purpose of this course.....	3
1. Course history	3
2. 3	
3. Course conventions	4
3.1. What you need	4
4. Scope	4
5. Technology Overview	4
6. JSR-135 Architecture and Interfaces.....	5
6.1. Manager Methods.....	6
6.1.1. Example MIDlet InfoTest.....	9
6.1.2. Playing a Tone.....	10
6.2. Control Interface	10
6.2.1. Playing Tone Sequences	11
6.3. The Player Interface	15
6.3.1. Player States.....	15
6.3.2. Player Life Cycle	18
6.3.3. Threading Issues	19
6.3.4. Resource and Latency Issues	20
6.3.5. Player Implementation and Behaviour	21
7. Example MIDlet	23
8. Appendix A.....	26
8.1. Running JSR-135 MIDlets in the Emulator	26
9. Appendix B.....	27
9.1. Comparison between the Sony Ericsson JSR-135 and the MIDP 2.0 Media APIs.....	27
10. Appendix C.....	28
10.1. Accessing Special Device Features Through iMelody	28
11. Appendix D.....	30
11.1. Converting Sampled Sounds to AMR Format	30
12. Resource Information	32
12.1. Abbreviations.....	32
12.2. Further Information and Links	32

Preface

1. Purpose of this course

This course is designed for Java™ 2 Platform, Micro Edition (J2ME™) programmers who wish to use Mobile Media APIs (MMAPIs) to add multimedia capabilities to mobile applications. The MMAPIs are also known as Java Specification Request 135, or JSR-135. These J2ME APIs manage time-dependent media, such as audio playback, tone generation, and sound mixing.

This course's goal is to first provide you a brief overview of these APIs. With this understanding, you next learn to write Java applications that utilize these APIs effectively, given the mobile device's limited resources. It is assumed that you are familiar with J2ME's Mobile Information Device Profile (MIDP), and how to write MIDP-based applications (MIDlets). Sample Java MIDlet code is provided that illustrates the use of these APIs.

- Note: At the time this course was prepared, the Sony Ericsson devices that implements the JSR-135 are the T610/T616/T618 and Z600 handsets, which for the rest of this paper will be denoted as the T610 and Z600 series.

2. Course history

Change history		
2003-09-18	Version 1.0	Initial Release

This document is published by:
Sony Ericsson Mobile Communications AB
SE-221 88 Lund, Sweden
Phone: +46 46 19 40 00
Fax: +46 46 19 41 00
www.SonyEricsson.com

First edition (September 2003)

This document is published by **Sony Ericsson Mobile Communications AB**, without any warranty. Improvements and changes to this text necessitated by typographical errors, inaccuracies of current information or improvements to programs and/or equipment, may be made by Sony Ericsson Mobile Communications AB at any time and without notice. Such changes will, however, be incorporated into new editions of this document. Any hard copies of this document are to be regarded as temporary reference copies only.

3. Course conventions

This course contains explanations of various Java classes and methods. The class and method names are denoted by use of a `mono-spaced` typeface in the course text, as is source code. Example: JSR-135 consists of three classes, `Manager`, `Player`, and `Control`. File names and directory path names are also denoted with this typeface.

Descriptions of developer tool GUI elements, such as button and menus, and user's interactions with them are presented in **bold** typeface. Example: "Choose **Sony Ericsson 1.1 Device** from the **Virtual Machine** component on the CodeWarrior Java target panel. Click **Apply**."

3.1. What you need

To develop Java MIDlets that use JSR-135 on Sony Ericsson devices, this course requires that you have the following SDKs installed on your Windows PC:

- Java 2 Standard Edition (J2SE) SDK 1.4.1 or later.
- Java 2 Runtime Environment (JRE) Standard Edition 1.4
- Sony Ericsson's J2ME SDK, version 1.2 or later
- Sony Ericsson T610 and Z600 series handset and connection to host PC for program writing and debugging. Connections are possible through a DSR-11 Enhanced RS 232 Cable, DCU-10 USB cable, infrared, or Bluetooth interface.

The Java SDK and JRE are available from <http://java.sun.com/j2se/1.4.1/download.html>.

Note: this paper assumes the use of Metrowerks CodeWarrior. However, you can use Sun ONE Studio 4; Borland JBuilder or other Java development tools to write MIDlets.

4. Scope

This paper covers the Sony Ericsson implementation of the MMAPI interfaces described in the JSR-135 specification. Note that the Sony Ericsson JSR-135 implementation for the T610 and Z600 series handsets supports only the audio components of this specification. While the T610 and Z600 series JSR-135 implementation appears similar to the MIDP 2.0 Media API specification (which only supports the audio components of the MMAPI), it is not.

- If you are familiar with the MMAPI or the Media API, you can go directly to Appendix B of this document. The differences between these two specifications and the T610 and Z600 series JSR-135 implementation are explained in detail there. These differences are also highlighted throughout the material in this document.

5. Technology Overview

JSR-135 APIs manage time-based data on J2ME-enabled devices. Such devices range in capabilities from high-end set-top boxes with few constraints in power, CPU processing capacity, and memory, to low-end mobile phones with severe constraints on power, CPU processing, and memory. JSR-135 APIs provide an abstraction layer that enables you to manipulate and present various types of multimedia data without concern as to a J2ME device's hardware capabilities, the media's format, or how the media content is delivered. For example, the APIs provide a consistent, platform-independent scheme that describes whether the data resides on a storage device such as hard drive or a DVD, or is obtained via the wired HTTP protocol or the WAP and SMS wireless protocols.

These APIs provide the following capabilities:

- **Playback or recording of time-based media without regard to format** – you use MIME types to describe the media's data format, and use Uniform Resource Identifiers (URIs) to specify where the media is located and its delivery mechanism. Do not confuse a URI with a URL: a URI is an abstract superclass of URLs. However, a URI can use a URL to locate media content. Content can also be stored as resources in a JAR file, and you use the Java `InputStreams` to access them. JSR-135 thus presents a unified interface that you use to obtain and process content in any format, and from any type of storage device.
- **Extensible** – new media types can be added by using plug-in style software modules known as *Players* without disrupting existing media capabilities. This feature is available only to the implementation: an application cannot add a new player or redefine the media type of an existing player. However, when the implementation does add new media players, you use the same interface and methods to access the new media content.
- **Tone generation** – you can generate a tone of a specific frequency and duration. You can also define tones sequences that play a sophisticated melody.
- **Permits a subset of functions** – Designers of resource-constrained devices such as mobile phones can choose to implement only those APIs that handle audio content.

The release of the MIDP 2.0 Media API specification in November 2002 provides an interface that is a subset of JSR-135 APIs. While JSR-135 supports various types of content, such as audio, graphics, and video, the Media APIs only support audio content.

- Because the MIDP 2.0 Media APIs are an exact subset of JSR-135, most programs written to use the Media APIs will function with little or no modifications on the T610 and Z600 series. Likewise, most programs written for the T610 and Z600 series with JSR-135 APIs will function on a MIDP 2.0 device with little or no modifications. However, there are differences that can impact your application's design. Consult Appendix B for a comparison between the MIDP 2.0 Media API and the T610 and Z600 series JSR-135 implementation. Also, the T610 and Z600 series handle only certain sound formats. Consult the section, "Manager Methods", for the supported audio formats.

This Developer's course will focus on those JSR-135 classes and interfaces as implemented in Sony Ericsson T610 and Z600 series phones.

To understand how to use JSR-135 effectively, it helps to understand how these APIs operate. This is covered in the next section.

6. JSR-135 Architecture and Interfaces

For audio playback, JSR-135 provides several classes and interfaces. These are the `Manager`, `Player` and `Control` classes. Table 1 summarizes the purpose of these classes.

Table 1. JSR-135/MIDP 2.0 classes that manage audio content.

Object	Type	Purpose
Manager	Class	Creates an instance of a <code>Player</code> that can interpret the specified audio data.
Player	Interface	Provides the interface that enables the application to control the playback of the media
Control	Interface	Provides a base set of control interfaces that an application uses for special-purpose, media manipulation capabilities. <code>ToneControl</code> is the only control supported by the Sony Ericsson T610 and Z600 series.

From the programmer's perspective, the `Manager` is the top-level access point to the handset's audio resources. It has methods that you use to query the device for the protocols and media formats it supports for audio playback. A `Manager` generates an instance of a `Player` that processes the requested data format. Finally, the `Manager` implements a simple tone generator that emits a single tone upon command.

The `Player` interface manages the generation of audio from time-based data. A `Player` implements the data processing and sound generation for a specific media format. As a consequence, a JSR-135 implementation might have several or more `Players`, each dedicated to managing a particular data type. When a request is made to the `Manager` to process a certain media type (say, MIDI), it checks to see if a `Player` capable of handling such data exists. If so, the `Manager` makes an instance of that `Player`. `Players` present methods that you use to operate the `Player` and control media playback.

The `Control` interface controls a `Player`'s media processing operations. The `Control` object for a `Player` can implement multiple controls that manage the tempo, pitch, data rate, and other aspects of media playback and recording.

- . Sony Ericsson's T610 and Z600 series implements only the `Control ToneControl`.

Generally, if any of the methods in these objects encounter an error condition related to accessing or controlling the media, they throw a `MediaException`.

To use JSR-135, you import that package `javax.microedition.media`, which contains the `Manager` and `Player` classes. The package `javax.microedition.media.control` contains a description of the various `Controls`.

Now let's look at each class in more detail.

6.1. Manager Methods

Table 2 summarizes some of the methods that the `Manager` class implements.

- Note: The table lists only those methods common to T610 and Z600 series JSR-135 implementation and MIDP 2.0 Media APIs.

Table 2: Methods implemented by the Manager class.

Manager methods	Description
<code>static Player createPlayer(java.io.InputStream stream, String type)</code>	Creates a Player to play back media from an InputStream.
<code>Static Player createPlayer(String locator)</code>	Creates a Player from an input locator.
<code>Static String[] getSupportedContentTypes(String protocol)</code>	Return the list of supported content types for the given protocol.
<code>Static String[] getSupportedProtocols(String content_type)</code>	Return the list of supported protocols for the specified content type.
<code>Static void playTone(int note, int duration, int volume)</code>	Play back a tone of a certain frequency, duration, and volume.

The `createPlayer()` method instantiates a Player that processes the requested data type. You describe the media's data type to `createPlayer()` either of two ways. In the first form of `createPlayer()` shown in the table, you pass the method an input stream and a data type argument. The type argument uses a MIME string to describe the data's format. Examples of JSR-135 data types are:

Data Format	MIME Type
Adaptive Multi-Rate (AMR) format sampled audio	audio/amr
Tone sequence data	audio/x-tone-seq
MIDI music data	audio/midi
iMelody tone data	audio/imelody

Here's an example that uses the first form of `createPlayer()` to specify MIDI data:

```

Player player;

try {

    String location = "Ashgrove.mid";

    InputStream is = getClass().getResourceAsStream(location);

    String type = "audio/midi";

    player = Manager.createPlayer(is, type);

} catch (Exception ex) { }

```

The input stream argument, `location`, references the MIDI data file `Ashgrove.mid` as a resource in the MIDlet's JAR file, while the `type` string instructs the `Manager` to make a MIDI `Player`.

The second form of `createPlayer()` takes a *media locator* in the URI syntax to both reference and identify the data content. A media locator takes the form:

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

where the `<scheme>`: portion of the media locator describes the protocol that delivers the data, and the `<scheme-specific-part>` portion tells where the data is located and describes its format, using a Uniform Resource Locator (URL). Recall that the protocol might deliver the data from a storage device, or a file on a remote server. A protocol can also reference the hardware on the device that's responsible for playing the data content. Here's an example where you make a `Player` that uses the J2ME device's built-in tone generator to generate audio:

```
Player player = Manager.createPlayer("device://tone");
```

- Your code should always check to see if the requested `Player` or `Control` is returned. If the media type or specified control is not available, the request returns null. Also, be sure to watch for and handle any exceptions that might occur.

Here's another example that obtains WAV-formatted audio from a server using the HTTP protocol:

```
Player player = Manager.createPlayer("http://java.sun.com/products/java-media/mma/media/test-wav.wav");
```

One question that always occurs to the developer at this stage is: "Of course, this wide support for different media types is nice. But what protocols and media types does the handset that I'm working with support?" Fortunately, a `Manager` has methods that let you get answers to those questions.

You use the `getSupportedContentTypes()` and `getSupportedProtocolTypes()` methods to query a J2ME device as to the types of media content and delivery protocols it supports.

Use `getSupportedContentTypes()` to report the types of data content that a particular protocol supports. For example, if you need to know what data formats the device's HTTP protocol handles, you'd write:

```

String mediaTypes[];

mediaTypes = Manager.getSupportedContentTypes("http");

```


The method returns the names of the supported data types in the String array `mediaTypes`. If the specified protocol is invalid or doesn't support any data types, the method returns an empty array.

Likewise, use `getSupportedProtocols()` to determine which protocols support a specific media type. For example, if you wanted to know which device protocols supported AMR formatted media, you would write:

```
String protocolTypes[];  
  
protocolTypes = Manager.getSupportedProtocols("audio/amr");
```

If the device doesn't support AMR, the method returns an empty array.

If you supply a null argument to both of these methods, the methods report all of the device's supported media types and protocols.

The Sony Ericsson T610 and Z600 series handsets support the following protocols and media types:

Protocols
http
device
Media Types
Tone sequence
MIDI
AMR
iMelody

- TIP: The Sony Ericsson AMR Converter 1.2 converts existing WAV sound files (.wav) into Adaptive Multi-Rate (AMR) formats. See Appendix D for further information.

6.1.1. Example MIDlet InfoTest

The demo MIDlet named `InfoTest` uses the `getSupportedContentTypes()` and `getSupportedProtocolTypes()` methods to query a T610 and Z600 series handset as to all of its content types and protocols. `InfoTest` displays this information on the screen. `InfoTest` and any other example programs are supplied with this course file. Expand the archive `InfoTest.zip`. Next, open the JAD file `infoTest.jad` by double-clicking on it. If your Windows PC is configured properly, this starts the Sony Ericsson simulator with the T610 skin. The simulator screen displays the JAR file name `InfoTest` and asks you to launch it. Click the **Launch** softkey to run the MIDlet.

Figure 1 shows the output of the `InfoTest` MIDlet on the T610 simulator's screen. Click **Exit** to terminate the MIDlet.

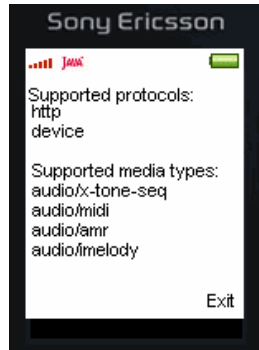


Figure 1. The protocols and media that the T610 supports.

6.1.2. Playing a Tone

The `Manager` implements a simple tone player for use with those J2ME devices with minimal audio capabilities. You use the method `playTone()` to generate a single tone of a specific frequency, duration, and volume. To play a note corresponding to middle C for five seconds at middle volume, you'd use the following code:

```
try {

    Manager.playTone(ToneControl.C4, 5000, 50);

    // 5000 milliseconds = 5 secs, 0 = no volume, 100 = max volume

} catch (MediaException me) {}
```

- `ToneControl.C4` is a JSR-135 constant that corresponds to the MIDI note 60. This MIDI note generates the frequency of 261.63 Hz. For those with a musical background, this corresponds to middle C on a piano keyboard.

6.2. Control Interface

A `Control` enables you to access any controls a `Player` might support. For JSR-135, there are a large variety of pre-defined `Controls` that manage MIDI, video, pitch, tempo, and recording functions, to name a few. The MIDP 2.0 Media API has only two `Control` subinterfaces: `ToneControl` and `VolumeControl`.

- The T610 and Z600 series implementation of JSR-135 only supports the `ToneControl` subinterface. For more information on what features JSR-135 supports, consult Appendix B.

Table 3 lists the methods available in `ToneControl` and some useful fields and constants.

Table 3: ToneControl method and fields.

ToneControl method	Description
<code>void setSequence(byte[] sequence)</code>	Assign a tone sequence array for playback.
ToneControl fields	Description
BLOCK_END	Defines the end point for a block (group of notes) in a tone sequence array. This is followed by the block's assigned number.
BLOCK_START	Defines a block's starting point. It is followed by a value the assigns the block a number. Block numbers can range from 0 to 127.
C4	Defines a constant for middle C, a frequency of 261.63 Hz. Its value is 60, which corresponds to the MIDI note for middle C.
PLAY_BLOCK	Play the specified block. You specify the block using a number between 0 and 127 that follows this tag.
REPEAT	The repeat event tag.
RESOLUTION	The resolution event tag. You specify the note's type with a value between 1 (whole note) and 127 (1/127 note) that follows this tag.
SET_VOLUME	The set volume event tag. The value that follows must be within the range 0 (mute) to 100 (maximum volume).
SILENCE	Defines a constant that indicates no sound is played. It is used to generate the rests (intervals of silence) in a tone sequence.
TEMPO	Defines the tempo event tag. It is followed by a value between 5 and 127 that, when multiplied by 4, specifies the tempo in beats per minute (bpm).
VERSION	Defines the version attribute tag. Currently should be set to 1.

For this course, we will only cover the `ToneControl` method, `setSequence()`. This method allows you to assign a monotonic tone sequence for playback. Tone sequences are packaged in byte arrays.

6.2.1. Playing Tone Sequences

Let's discuss handling tone sequences further. It's important to understand that playing a tone sequence requires the use of a `Player`, because the `Manager`'s `playTone()` method only generates a single note.

A *tone sequence* is a byte array that describes an arbitrary series of note-duration pairs: that is, notes and the intervals that they are played. A note-duration pair can also designate rests, or intervals of silence. You can also define sequence blocks—that is, groups of note-duration pairs. You can assemble complex melodies by defining sets of blocks and playing them on command.

A note's frequency is determined by a note value. Note values are integers with values between 0 and 127.

- JSR-135's note values are identical to MIDI note values.

MIDI number	Note name	Keyboard representation	Note frequency
21	A0		27.500
23	B0		30.868
24	C1		32.703
26	D1		36.708
28	E1		41.203
29	F1		43.654
31	G1		48.999
33	A1		55.000
35	B1		61.735
36	C2		65.406
38	D2		73.416
40	E2		82.407
41	F2		87.307
43	G2		97.999
45	A2		110.00
47	B2		123.47
48	C3		130.81
50	D3		146.83
52	E3		164.81
53	F3		174.61
55	G3		196.00
57	A3		220.00
59	B3		246.94
60	C4		261.63
62	D4		293.67
64	E4		329.63
65	F4		349.23
67	G4		392.00
69	A4		440.00
71	B4		493.88
72	C5		523.25
74	D5		587.33
76	E5		659.26
77	F5		698.46
79	G5		783.99
81	A5		880.00
83	B5		987.77
84	C6		1046.5
86	D6		1174.7
88	E6		1318.5
89	F6		1396.9
91	G6		1568.0
93	A6		1760.0
95	B6		1975.5
96	C7		2093.0
98	D7		2349.3
100	E7		2637.0
101	F7		2793.0
103	G7		3136.0
105	A7		3520.0
107	B7		3951.1
108	C8		4186.0

 = Note used for tuning
 = Note defined by ToneControl.C4

Figure 2. JSR-135 and MIDI note value mappings to tone frequencies.

Figure two shows the relationship between MIDI note values and frequencies. You use these note values to build a tone sequence that represents a melody.

A tone sequence's resolution, tempo, and the duration value determine the actual time interval that a note plays. This can relationship among the values can be expressed by the equation:

$$\text{duration} * 60 * 1000 * 4 / (\text{resolution} * \text{tempo})$$

where duration is in milliseconds. Duration values are integers with values between 0 and 127. Tempo values can range between 5 and 127, and represents beats per minute. Resolution values range between 1 to 127 and represent 1/resolution notes. The resolution and tempo information is encoded within the tone sequence array, along with the note-duration pairs. Table 4 shows some common values used to configure note durations.

Table 4: Duration and tempo values used to configure note durations in a tone sequence.

Note Length	Duration, Resolution=64	Duration, Resolution=96
1/1	64	96
1/4	16	24
1/4 dotted	24	36
1/8	8	12
1/8 triplets	N/A	8

Now that you understand the fundamental parts that make up a tone sequence, let's see how it is done. We'll start by defining some constants:

```
byte d = 8; // eighth note

byte d2 = 16; // quarter note

byte A3 = ToneControl.C4 - 3; // MIDI note 57

byte B3 = ToneControl.C4 - 1; // MIDI note 59

byte C4 = ToneControl.C4; // MIDI note 60

byte D4 = ToneControl.C4 + 2; // MIDI note 62

byte E4 = ToneControl.C4 + 4; // MIDI note 64

byte F4 = ToneControl.C4 + 5; // MIDI note 65

byte G4 = ToneControl.C4 + 7; // MIDI note 67

byte A4 = ToneControl.C4 + 9; // MIDI note 69

byte B4 = ToneControl.C4 + 10; // MIDI note 70

byte rest = ToneControl.SILENCE; // eighth-note rest
```

These declarations establish several useful delay intervals, and define some notes. The note definitions use the constant `ToneControl.C4` and a little math to map user-friendly note names (such as A3, B3, C4, and so on) to MIDI note numbers that comprise the tone sequence. Also, notice that there is another constant, `ToneControl.SILENCE`, which defines a rest value (that is, no tone played). Rests, like notes, require a duration value to specify the interval.

With the definitions in place, we can write a melody using tone-duration pairs and block commands:

```
// Build tone sequence for "Song of the Wind"

byte[] tuneSequence = new byte[] {

    ToneControl.VERSION, 1,

    ToneControl.TEMPO, 30,

    ToneControl.RESOLUTION, 64,

    A3,d,B3,d,C4,d,D4,d,E4,d,E4,d,E4,d,E4,d,

    ToneControl.BLOCK_START, 0,

    F4,d,D4,d,A4,d,F4,d,E4,d2,rest,d2,

    ToneControl.BLOCK_END, 0,

    ToneControl.PLAY_BLOCK, 0, // play block 0

    ToneControl.PLAY_BLOCK, 0, // play it again, Sam

    ToneControl.BLOCK_START, 1,

    E4,d,D4,d,D4,d,D4,d,D4,d,C4,d,C4,d,C4,d,C4,d,B3,d,B3,d,B3,d,

    ToneControl.BLOCK_END, 1,

    ToneControl.PLAY_BLOCK, 1,

    A3,d,C4,d,E4,d2,

    ToneControl.PLAY_BLOCK, 1, // repeat the long phrase

    A3,d2,rest,d2

}; // end tuneSequence
```

Observe how the constants `ToneControl.BLOCK_START` and `ToneControl.BLOCK_END` group tone-duration pairs into blocks. You can have a maximum of 127 blocks, each which is referenced by a number. The `ToneControl.PLAY_BLOCK` command has the device play the specified block. As the example shows, the block feature lets you assemble a complex melody while using very little memory.

The constant `ToneControl.TEMPO` specifies that the byte following it represents a tempo value, and the constant `ToneControl.RESOLUTION` indicates that the succeeding byte is a resolution value.

Finally, let's play the tune:

```
tonePlayer = Manager.createPlayer(Manager.TONE_DEVICE_LOCATOR);

tonePlayer.setLoopCount(1);      // Loop only once

tonePlayer.realize();            // Gather up the needed information

ToneControl tc =
(ToneControl)tonePlayer.getControl("javax.microedition.media.control.ToneControl");

tc.setSequence(tuneSequence);    // Make reference to tone sequence array

tonePlayer.start();              // Play it
```

The code uses the `Manager` to make a `Player` object that plays the tone sequence. The reference to the `tuneSequence` array is established using the `ToneControl` interface. Finally, calling the `Player`'s `start()` method initiates audio playback.

If the `Player` throws an exception for a tone sequence, check for the following coding errors:

- A broken tone-sequence pair. Put another way, each tone played must have both a note number and a duration value, with the note number defined first. Check your tone sequence and insert any missing notes or durations.
- A block definition that's not terminated. Search every block for missing `BLOCK_CLOSE` commands.

The example code that plays a tone sequence makes use of methods that initialize the `Player` and generate the audio. A more thorough description of these methods can be found in the next section.

6.3. The Player Interface

As discussed previously, a `Player` is the engine that processes time-dependent media data. You use the `Manager` to make an instance of the `Player` that handles the specified data format. To make the best use of JSR-135, it helps to understand how a `Player` operates and what methods are available to you.

6.3.1. Player States

A `Player` can be in one of five operating states. These states determine whether the `Player` has sufficient information to play the requested media, if it should prefetch or release the requested data, if it is processing data or halted, or if it is terminated. Table 5 provides a summary of these states.

Table 5: The various operating states of a Player object.

Player State	Transitioned to by calling	Description
UNREALIZED	<code>Manager.createPlayer()</code>	If a <code>Player</code> for the specified media exists, the <code>Manager</code> makes an instance of it. The <code>Player</code> has not yet resolved any of the references to obtain the necessary resources to perform the request.
REALIZED	<code>realize()</code> or <code>deallocate</code>	The <code>Player</code> has the information necessary to acquire the resources and service the media request.
PREFETCHED	<code>prefetch()</code> or <code>stop()</code>	The <code>Player</code> acquires the resources to perform the operation. These might be files fetched from a Web server or as a resource on the device's file system. It also reserves the audio hardware for the operation.
STARTED	<code>start()</code>	The <code>Player</code> processes the media data. It stops when the end of the media is reached, or <code>stop()</code> is issued.
CLOSED	<code>close()</code>	The <code>Player</code> releases all of its resources. It can not be used again.

The various `Player` states allow fine-grained control over a device's media processing and playback, particularly in a mobile device's resource-constrained environment. For example, a `Player` doesn't reserve the audio hardware until it enters the `PREFETCHED` state. This is because such an allocation can easily cause conflicts for an incoming call that needs the audio hardware.

- **Note:** For the T610 and Z600 series handsets, the audio hardware is not reserved until you invoke the `start()` method.

The different states thus allow you to make a `Player` while avoiding any time-consuming data delivery or scarce hardware allocation until you're ready to use it. Conversely, if you must reduce the latencies in the media processing request, you would call `prefetch()` to obtain the data content before starting the `Player`.

The discussion of changing `Player` states neatly segues into the description of `Player` methods. Table 6 lists the `Player` methods in the T610 and Z600 series JSR-135 implementation that are common to both MMAPI and MIDP 2.0 Media APIs.

Table 6: T610 and Z600 series Player methods that are common to MMAPI and MIDP 2.0 APIs. Methods highlighted in grey change the Player's operating state.

Player Method	Description
<code>void addPlayerListener(PlayerListener playerListener)</code>	Add a player listener for this Player.
<code>Void close()</code>	Close the Player and release its resources.
<code>Void deallocate()</code>	Release scarce and exclusive resources, such as memory and audio hardware owned by the Player.
<code>String getContentType()</code>	Get the content type of the media being processed by this Player.
<code>Long getDuration()</code>	Get the media's duration (time interval). Not implemented by the T610 and Z600 series, returns <code>TIME_UNKNOWN</code> .
<code>Long getMediaTime()</code>	Gets this Player's current media time. Not implemented by T610 and Z600 series, returns <code>TIME_UNKNOWN</code> .
<code>Int getState()</code>	Gets the current state of this Player.
<code>Void prefetch()</code>	Acquire scarce and exclusive resources for the Player, and process as much data as required to reduce the start latency.
<code>Void realize()</code>	Constructs portions of the Player without acquiring any scarce resources required for the request. Some data may be loaded into buffers.
<code>Void removePlayerListener(PlayerListener playerListener)</code>	Remove a player listener for this Player.
<code>Void setLoopCount(int count)</code>	Set the number of times the Player loops and plays the content.
<code>Long setMediaTime(long now)</code>	Sets the Player's media time.
<code>Void start()</code>	Starts the Player as soon as possible.
<code>Void stop()</code>	Stops the Player.

Those methods highlighted in grey transition a Player from one operating state to another. Once you understand the implications of each state and how to switch to them using various Player methods, you can manage the Player's life cycle so as to make effective use of a device's limited resources while servicing a media request.

6.3.2. Player Life Cycle

Figure 3 depicts the lifecycle of a `Player`, and the various states it traverses as it executes.

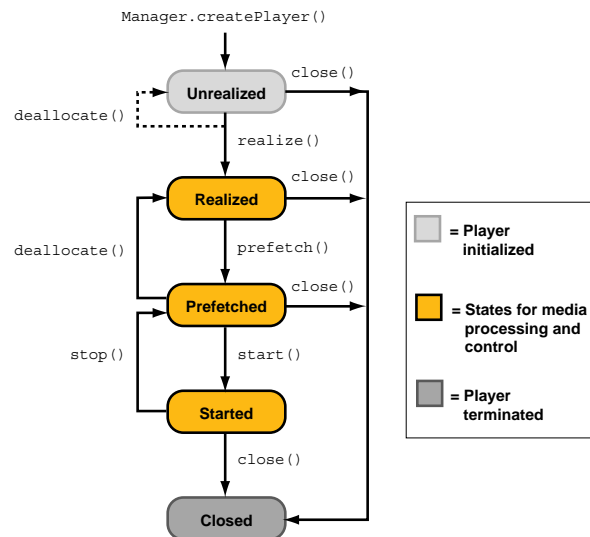


Figure 3. A Player's life cycle. Most often you'll switch between the Realized, Prefetched, and Started states when processing media.

As the figure indicates, the `Manager` makes an instance of the `Player`, placing it in the UNREALIZED state. Calling `realize()` has the `Player` resolve any references to obtain the data it needs to process a request, and the `Player` transitions to the REALIZED state. Once in the REALIZED state, the `Player` can't return to the UNREALIZED state, it can only shuttle among the PREFETCHED and STARTED states. The one exception is if you call `deallocate()` before the `Player` enters the REALIZED state, it returns to the UNREALIZED state (the dotted arrow in the figure). Note that this particular situation isn't of concern the application programmer, since they can't register new `Player` types. It is mentioned only for the sake of completeness.

In the REALIZED state, the `Player` may fetch some of the media data, but it will not allocate and reserve scarce resources such as the audio hardware. The `Player` won't fetch any data if it is a time-consuming task, such as reading a file from a Web server.

Calling `prefetch()` switches the `Player` to the PREFETCHED state. The `Player` fetches data and fills any buffers (if it hasn't already), and reserves the scarce resources. Recall that for the T610 and Z600 series implementation that the audio hardware is not reserved until `start()` is invoked.

- The T610 and Z600 series handsets don't support data streaming. Therefore, requests made from remote URLs are downloaded completely during prefetch. This delay can make the playback of long audio clips impractical. Applications that play short audio clips (such as games) should prefetch all of the data during program initialization to reduce latency.

Invoking `start()` transitions the `Player` to the STARTED state, and it begins audio playback.

The `Player` processes the media until it reaches the end of the media, whereupon the `Player` returns to the PREFETCHED state. Media processing can be halted by calling `stop()`, which also transitions the `Player` to the PREFETCHED state. Calling `start()` again in this situation has data processing resume from where it was halted in the media file.

Call `deallocate()` on a `Player` to release any memory and hardware resources that it owns. This returns the `Player` in the REALIZED state. To reuse the `Player`, call its `realize()` and `start()` methods again.

Once you're done with a `Player`, call `close()` to send the `Player` on a one-way trip to the CLOSED state, where the `Player` is terminated and any remaining resources that it owns are released.

What happens if you call a particular method and the `Player` isn't in the appropriate operating state? In this situation, implicit calls are made automatically to the other methods, which transition the `Player` to the state that the method requires. For example, if you call `start()` on a `Player` in the `UNREALIZED` state, the `realize()` and `prefetch()` methods are called before `start()` executes. Therefore, playing a sound with a `Player` becomes a simple task, like so:

```
try {

    Player p = Manager.createPlayer("http://www.fauxserver.com/fanfare.amr");

    p.start(); // Other state methods called implicitly

    } catch (MediaException me) {

} catch (IOException ioe) {}
```

While this code is very simple, bear in mind that it provides little control over how the resources are managed. More often you will make a `Player`, and then use the `prefetch()` and `deallocate()` methods to determine precisely when certain resources are reserved and released. In addition, you'll use the `close()` method to dispose of a `Player` when you're done with it.

6.3.3. Threading Issues

Once a `Player` is in the `REALIZED` state, invoking `start()` transitions the `Player` to the `STARTED` state and begins processing the requested time-dependent data. Once the playback begins, `start()` returns. However, methods such as `Manager.createPlayer()` and `prefetch()` can block on lengthy I/O operations or in allocating scarce resources. Therefore, you often want to create a `Player` within a separate thread to reduce the impact of these time-consuming operations on the MIDlet's responsiveness. As an example of this:

```
class PlayerCanvas extends Canvas implements Runnable {
Public void run() {      // The run method for the class.
    if (player == null) {
        try {
            createPlayer();
            player.realize();
            player.start();
        } catch (Exception ex){
            System.err.println("Problem running player");
        } // end catch
    } // end if
    while (!interrupted) {
        try {
            Thread.sleep(100);
        } catch (Exception ex){}
    } // end while
    } // end run
} // end PlayerCanvas class
```

This code snippet, of course, glosses over how to construct a thread and create a `Player`. However, the example MIDlet named `PlayAudio` shows in detail how this is done. Most of the code was obtained from SUN's MIDP 2.0 code example and adapted for use by the T610. We'll see how this MIDlet works by running it with `CodeWarrior` in the "Example MIDlet" section.

6.3.4. Resource and Latency Issues

On MIDP devices, conserving scarce resources versus extracting optimum performance involves compromises. However, now that you understand the care and feeding of `Players`, you can make intelligent decisions as to how to best manage a device's resources while achieving decent performance.

You can reduce the latency of audio playback—which is a critical factor in games—by invoking the `prefetch()` method to ensure that a `Player`'s buffers are filled with data. Thus, to cache an audio clip that plays without delay becomes a matter of:

```
try {  
  
    createPlayer();  
  
    player.realize();  
  
    prefetch();    // Have the player fill its buffers with audio data  
  
} catch (Exception ex){  
  
    System.err.println("Problem setting up player");  
  
} // end catch
```

Now when `start()` is called, the sound plays immediately.

The downside of this scheme is that caching the data this way consumes memory. This shouldn't be a problem unless you're working with media composed of large sampled sounds. If the amount of available memory is an issue, coordinate the use of your `Players` carefully. For those `Players` that aren't required in this portion of the MIDlet's code, you should call `deallocate()` to release any memory they may have reserved, or `close()` if you are through with them.

In certain situations, one technique to conserve nonvolatile memory would be to download the audio data from a server. Thanks to JSR-135's emphasis on a hardware-independent interface, this is quite easy to accomplish:

```
try {  
  
    Player player = Manager.createPlayer("http://homeserver.SEMC.com/Mozart.mid");  
  
    Player.setLoopCount(1);    // Play it once  
  
    player.realize();  
  
    player.start();            // prefetch() called implicitly  
  
} catch (MediaException mex){  
  
    System.err.println("Problem downloading data");  
  
} // end catch
```

Of course, be aware of the pitfalls that accompany this particular solution. As discussed previously, this technique works only if latency isn't an issue. This solution isn't recommended for games, where you should store the audio clips as resources in the MIDlet's JAR file.

Downloading audio from a remote server also poses another problem. Simply put, if the server is down or the device can't gain access to the wireless network, the audio can't be played. Ask yourself: can the application still operate if this occurs?

In addition, for security reasons J2ME might display an alert that requires you to authorize a network connection. This can be annoying if it occurs too often during the course of a business application, and is downright lethal if it happens in the middle of a game. Examine your application's audio requirements carefully to determine if you need to retrieve audio data this way. It might make more sense to store lower-quality audio data as resources in the MIDlet. This way, if the program can't access the server to fetch the higher-quality versions of the audio files, it can rely on the data stored in the MIDlet's resources.

Finally, if all else fails, be aware of how the different types of audio-formatted data consume device resources. The following list summarizes the trade-off between sound quality and resource use.

- **Tones** – Uses the least amount of memory and processor overhead. Lowest audio quality.
- **iMelody** – Uses little memory and processor overhead. Good audio quality for simple tunes and effects. Can also use special device features such as the back light, LEDs, and vibration. Consult Appendix C for more information on how to access these features.
- **MIDI** – Uses little memory and more processor overhead. Moderate to excellent audio quality for music. Processor overhead is a function of the number of voices that the MIDI sequence encodes. Certain MIDI notes may reproduce poorly due to limitations in the audio hardware's frequency response.
- **AMR** – Requires the most memory and most processor overhead. Best audio quality for sampled sounds.

Weigh carefully whether the application really needs those high-quality sampled-audio sound effects, or whether MIDI or tone sequences might make a reasonable substitute.

6.3.5. Player Implementation and Behaviour

While the JSR-135 specification provides a consistent programming interface that you can use to generate audio, be aware the JSR-135 is an optional standard. This means that a vendor may choose to implement only specific portions of the API. In addition, the behaviour of certain classes or methods may vary. For example, we've already noted that for the T610 and Z600 series, a `Player` reserves hardware when the `start()` method is invoked, and not the `prefetch()` method as described by the specification.

- Further information about which APIs the T610 and Z600 series supports can be found in Appendix B.

To deal with situations where a portion of the API may not be implemented, a MIDlet should be well-behaved in that it queries the device as to what capabilities are available before attempting to use them. This is accomplished by using the `System.getProperty(String key)` method. JSR-135 defines a number of key strings that you can use to obtain this information. Table 7 shows these key strings and the capability associated with each.

Table 7: The key strings used to query a device for its JSR-135 characteristics.

Key	Description	T610 and Z600 Series Returns
<code>supports.mixing</code>	Checks if audio mixing is supported. Returns <code>true</code> if supported, <code>false</code> if not. Mixing is supported if: <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 while at least one <code>Player</code> plays back audio. At least two <code>Players</code> can play back audio simultaneously. 	Returns <code>false</code> , because for the T610 and Z600 series, two <code>Players</code> can't play back simultaneously.
<code>supports.audio.capture</code>	Checks if audio capture is supported. Returns <code>true</code> if supported, <code>false</code> if not.	<code>false</code>
<code>supports.video.capture</code>	Checks if video capture is supported. Returns <code>true</code> if supported, <code>false</code> if not.	<code>false</code>
<code>supports.recording</code>	Checks if recording capabilities are supported. Returns <code>true</code> if supported, <code>false</code> if not.	<code>false</code>
<code>audio.encodings</code>	Returns either a string that describes the encoding format for captured audio, or <code>null</code> if not supported.	<code>null</code>
<code>video.encodings</code>	Returns a string that describes the encoding format for captured video, or <code>null</code> if not supported.	<code>null</code>
<code>video.snapshot.encodings</code>	Returns either a string that describes the encoding format, or <code>null</code> if not supported.	<code>null</code>

The T610 and Z600 series implementation permits only one active `Player` at a time. However, it does allow you to have multiple `Players` configured and their data buffers filled. Knowing this, you can write code that provides background music while permitting the playback of short sounds.

For example, assume you have a game with background music and brief sound effects. `Player1`, which provides the background music, has its loop count set to `-1`, so that it plays music continuously. `Player2`, which manages the short sound effect, would be ready with its data prefetched. By default, `Player2` plays the sound effect only once. When the `MIDlet` calls `Player2`'s `start()` method, `Player1` is stopped and `Player2` plays its sound effect. When `Player2` is done, it sends an `END_OF_MEDIA` event to itself and `Player1`. `Player1` now starts playing again, playing from the beginning of its media file. Note that `Player1` doesn't resume where it was interrupted in the media file, it starts from the beginning of the media. However, with a carefully designed background music loop, such an effect won't be noticed. The following code illustrates how this is done:

```

// player initialization

Player p1 = new Player("BackGroundMusic.mid");

p1.setLoopCount(-1);    // Play the background music indefinitely

p1.prefetch();          // Fill the buffers


// Sound effect that indicates to the user that he has picked up an
//    object or performed a certain action in the game

Player p2 = new Player("shortsound.mid");

p2.prefetch();


// Sometime later, start playing a background sound

p1.start();


// Game progresses...


// Play the second sound as required to provide feedback

p2.start();


// Once p2 finishes playing, p1 will restart

```

7. Example MIDlet

This section assumes that you are familiar with the operation of Metrowerks' CodeWarrior Integrated Development Environment (IDE), and have installed CodeWarrior Wireless Studio and the Sony Ericsson J2ME SDK on your system.

Locate the archive `PlayAudio.zip` and expand it, which makes the directory `SEMC PlayAudio`. Open the project file `PlayAudio.mcp` with CodeWarrior Wireless Studio. The IDE displays a **PlayAudio.mcp** Project window similar to the one shown in Figure 4.

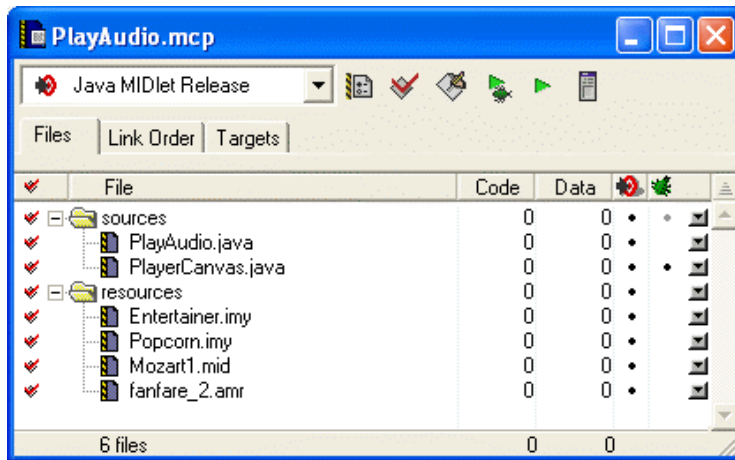


Figure 4. The CodeWarrior Project window for the PlayAudio MIDlet code example.

The source file `PlayAudio.java` implements the MIDlet's GUI. The source file `PlayerCanvas.java` implements the various methods used to create a `Player`, switch its operating states, and run it in a separate thread. The files `Entertainer.imy`, `Popcorn.imy`, `Mozart1.mid`, and `fanfare_2.amr` represent audio media files that are stored as resources in the MIDlet's JAR file.

A glance at the **Java Target** settings panel is useful before you start. Select **Edit | Java MIDlet Release Settings...** and a **Java Release Settings** window appears. Click on **Java Target** in the **Target Settings Panels** list located in a pane on the left side of this window. Figure 5 shows this panel's Java settings for the PlayAudio project.

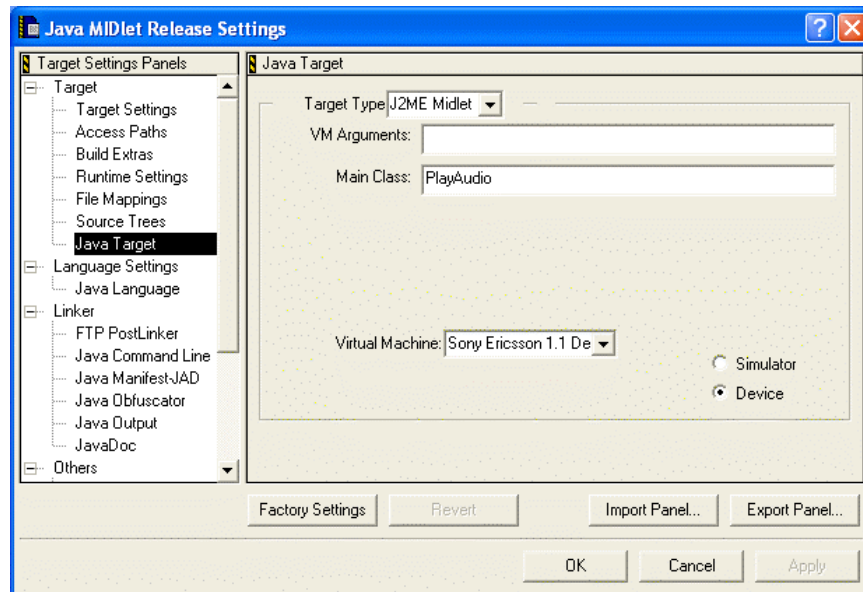


Figure 5. The Java Target settings panel.

Note that **Virtual Machine** component on this panel is set to **Sony Ericsson 1.1 Device**, and the **Device** radio button is selected. These settings instruct CodeWarrior to download the example code to the T610 and Z600 series handset via a serial cable. Other interface connections via USB, infrared, or wireless Bluetooth are available, but for this example the serial cable is used.

Attach the DSR-11 Enhanced RS 232 Cable to a serial port on your Windows PC, with the other end connected to the cable interface on the bottom of the T610 and Z600 series handset. Verify that the serial port (COM1: is assumed) is configured to the following settings:

- Baud Rate 115200
- Data bits: 8
- Parity: None
- Stop Bits: 1
- Flow Control: None

Click on the green VCR-style **Run** button on the **PlayAudio.mcp** Project window. The CodeWarrior IDE compiles the Java source, and merges the bytecodes and audio resources into a JAR file. The **Project** window now displays size values in the code and data columns of the Project window. The Java runtime should start (this might take a while). Next, a **Sony Ericsson J2ME SDK – Connection Proxy** window appears, as Figure 6 shows.



Figure 6. The serial Connection Proxy window manages serial transfers to the T610.

The green arrows in this window blink as data is sent to the handset, and it acknowledges the safe receipt of the data. If all goes well, the T610 and Z600 series displays the GUI for the PlayAudio MIDlet, as shown in Figure 7.



Figure 7. The PlayAudio MIDlet's GUI. Use the navigation key to make a choice and press the Play softkey to play the audio.

Use the handset's four-way navigation key to move through the menu and select the type of data to play. Press the **Play** softkey or depress the navigation key, and the handset plays the selected audio data. Press the **Exit** softkey to terminate the MIDlet.

- **Note:** The audio data is transferred from the PC to the handset via the serial cable. There will be a delay while this transfer occurs. Once the transfer completes, the handset plays the audio.

This MIDlet's source code demonstrates how to use the Manager's `playTone()` method, and how to create Players that play a tone sequence, or MIDI-, iMelody-, and AMR-formatted sound files. It also shows how to invoke the methods that change the Player's operating state. Use the example code to guide you in writing your own audio applications.

8. Appendix A

8.1. Running JSR-135 MIDlets in the Emulator

The release notes indicate that the emulator doesn't support iMelody or AMR playback. This capability is only supported in the hardware (that is, you need the T610 and Z600 series handset). Since that handset might be a scarce resource, how can you write and debug code?

If you've looked at the example code in the PlayAudio MIDlet, you may have noticed that the same methods that play MIDI-formatted data also work with iMelody- and AMR-formatted data. Therefore, you can write and test your custom playback methods using MIDI data in the emulator. However, if you configure the CodeWarrior IDE to build the MIDlet and use the emulator, you get the following error message: `Package javax.microedition.media.* and javax.microedition.media.control.* not found in import.`

This message occurs because of how the SDK components are registered with CodeWarrior. The registration assumes that any use of JSR-135 APIs might involve AMR or iMelody playback, and so any effort to build code using them is blocked. That is, the registration assumes the worst-case scenario. To work around this, you compile the MIDlet using the device settings. To run the MIDlet, you then change the settings so that the MIDlet runs in the simulator. Here how it's done:

Step 1: Have CodeWarrior build the MIDlet for the hardware target. Open the **Java Target** settings panel by choosing **Edit | Java MIDlet Release Settings...** and then clicking on **Java Target** in the **Target Settings Panels** list. Ensure that the **Java Target** panel settings are configured as follows:

- **Target Type:** J2ME MIDlet
- **Main Class:** The name of your MIDlet class
- **Virtual Machine:** Sony Ericsson 1.1 Device
- **Device** radio button selected

The panel should appear similar to the one in Figure 5. Change the settings if necessary, and click **Apply** to save these changes. Leave this panel open.

Step 2: Build the MIDlet by choosing **Project | Make**. (Do NOT run the project, just compile and link it.)

Step 3: Change the **Java Target** panel settings as follows:

- **Virtual Machine:** Sony Ericsson 1.1 Emulator
- **Simulator Config File:** SonyEricsson_T610
- **Simulator** radio button selected

The **Java Target** settings panel should resemble Figure 8. Click **Apply** to save your changes. Leave this panel open.

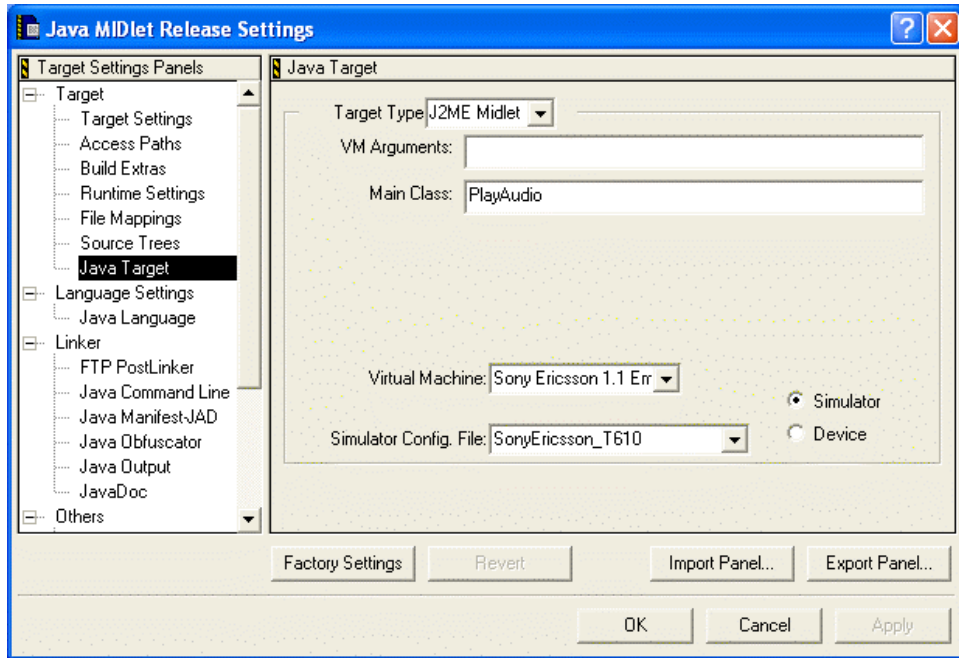


Figure 8. The Java Target settings that allow the MIDlet execute in the emulator.

Step 4: Click on the **Run** button on the Project window. The Java runtime starts up (this may take a while), and then the phone simulator window appears. Test your code with the simulator, or execute the MIDlet using the CodeWarrior debugger. When you discover a coding error, make changes to the source code with the IDE's editor.

Step 5: Return to Step 1, change the Java Target settings, and recompile your code.

If the emulator attempts to play an iMelody or AMR-formatted audio file, the Java runtime gives you a warning message.

9. Appendix B

9.1. Comparison between the Sony Ericsson JSR-135 and the MIDP 2.0 Media APIs

The T610 and Z600 series handsets support only the audio portions of the JSR-135 API suite. The MIDP 2.0 Media API is a subset of the JSR-135 API that implements the audio-only APIs. Therefore, a comparison between the two APIs is inevitable as you, a developer writing T610 and Z600 series handset applications, want to exploit features common to both APIs.

On the surface, Sony Ericsson's T610 and Z600 series JSR-135 implementation might be considered equivalent to the MIDP 2.0 Media API. This assumption is false. The following is a summary the differences between these two APIs.

- The Media API provides two audio-based Controls: `ToneControl` and `VolumeControl`. The Sony Ericsson JSR-135 only implements one audio-based Control, `ToneControl`.
- While the MMAPI specification supports multiple `Players`, the T610 and Z600 series implementation of JSR-135 does not. Starting a new `Player` interrupts and terminates the playback of the currently active `Player`. Sound mixing is not permitted. See the section "Player Implementation and Behavior" for more information.

- The Media API doesn't support the `TimeBase` class. Sony Ericsson's T610 and Z600 series implementation does support this JSR-135 class. Two methods that utilize `TimeBase`, `getTimebase()` and `setTimebase()`, are available to a `Player`. These methods are often used to synchronize the output of multiple `Players`. However, since the T610 and Z600 series JSR-135 doesn't support multiple players, these methods do nothing. For example, the method `getTimebase()` returns the constant `TIME_UNKNOWN`.
- The T610 and Z600 series JSR-135 implementation does not support the `Player` methods `getDuration()` and `getMediaTime()`. Although present, they only return a value of `TIME_UNKNOWN`.
- Both the T610 and Z600 series JSR-135 implementation and the Media API do not implement the `DataSource` class, which is used to write custom protocols to retrieve media content.

Table 8 shows those `VolumeControl` methods supported by the MIDP 2.0 Media API. Since the T610 and Z600 series JSR-135 implementation doesn't support `VolumeControl`, these methods are not available to applications written for the T610 and Z600 series handset.

Table 8: VolumeControl methods that are only available to the MIDP 2.0 Media API.

VolumeControl methods	Description
<code>int getLevel()</code>	Get the current volume level setting.
<code>Boolean isMuted()</code>	Get the mute state of the signal associated with this <code>VolumeControl</code> .
<code>Int setLevel(int level)</code>	Set the volume using a linear point scale with values between 0 and 100.
<code>Void setMute(boolean mute)</code>	Mute or unmute the <code>Player</code> associated with this <code>VolumeControl</code> .

10. Appendix C

10.1. Accessing Special Device Features Through iMelody

The Infrared Data Association (IrDA) manages the iMelody audio standard. The current version number of the standard is 1.2. This audio format defines audio elements such as volume level, rest delays, notes, and a tempo. You use these elements to construct a tone sequence. These tone sequences can be further combined into repeat blocks that are played one or more times. This latter capability allows the assembly of complex melodies while using little memory: music pieces of several minutes duration or longer can be encoded in iMelody using just a few kilobytes. Currently all iMelody information is encoded in ASCII text. While this seems limiting, the text-only format allows iMelody files to be readily transferred between devices using simple text-based messaging protocols.

An iMelody file starts with a header that describes the file's version and format, plus optional information such as identifying information and the melody's beat and volume. A payload section follows the header, and contains the notes, rests, repeat blocks, and other control elements, as described below. The file ends with a footer whose purpose is to signal the end of both the payload and the file. The file extension `.imy` designates an iMelody-formatted audio file.

In summary, a very basic iMelody file would look something like this:

```

BEGIN:IMELODY

VERSION:1.2

FORMAT:CLASS 1.0

MELODY:V7&b2#c3V-c2*4g3d3V+#d1r3d2e2:d1V+f2f3.

END:IMELODY

```

Notes are encoded as follows: a note value, followed by a duration value. The ASCII characters "c", "d", "e", "f", "g", "a", and "b" represent regular notes, while characters "#c", "#d", "#e", "#f", "#g", "#a", and "#b" encode sharp notes, and the characters "&c", "&d", "&e", "&f", "&g", "&a", and "&b" represent flat notes. An optional prefix value chooses the note's octave. Octaves values range between *0 and *8, where *0 has the A note generate a 55Hz tone, while *8 has the A note generate a 14080 Hz tone. By default, the octave value is *4, so the A note generates 880 Hz.

Durations are a numeric value between 0 and 5, with 0 representing a full note, and 5 representing a 1/32 note. To play a half note at 440 Hz, you'd write "*3a1" in the file's payload section.

Of interest are several elements that represent commands. These commands control special features on the device. Table 9 lists these commands.

Table 9: iMelody commands that access special device features.

Command	Purpose
ledoff, ledon	Turns the device's LED off and on
backoff, backon	Turns the device's screen back light off and on.
vibeeoff, vibeon	Turns the device's vibration mechanism off and on.

The LED on the T610 and Z600 series is a red LED embedded in the tip of the joystick, and is different on other devices. The following iMelody plays an ascending scale of half notes in the A4 octave, and turns the LED on, off, and back on as the notes play:

```

BEGIN:IMELODY

VERSION:1.2

FORMAT:CLASS1.0

BEAT:200

MELODY:LEDON*4c1d1e1lledooff1g1a1b2ledon

END:IMELODY

```

You can embed the other commands so that the device vibrates and its back light flashes the tune plays. As an example of this, try the example program PlayAudio that came with the course and examine the iMelody file popcorn.imy. This iMelody file shows how to switch the back light off and on in synch with the music.

11. Appendix D

11.1. Converting Sampled Sounds to AMR Format

The T610 and Z600 series uses an adaptive multi-rate (AMR) speech codec that encodes and decodes sampled voice data for use through networks of limited bandwidth. The AMR codec was developed by the European Telecommunications Standards Institute (ETSI) and is a standard data transfer scheme for voice over GSM networks. It is also a required codec for third-generation (3G) phones.

The AMR codec is a multi-mode codec that implements eight narrow-band audio modes. These modes manage bit rates between 4.75 and 12.2 kbps, where higher bit rates yield higher-quality audio output. Table 10 shows the various narrow-band modes that the codec supports.

Table 10: The AMR modes. The mode numbers indicate bit rate.

Mode	Bit rate	Total speech bits
AMR 4.75	4.75 kbps	95
AMR 5.15	5.15 kbps	103
AMR 5.9	5.9 kbps	118
AMR 6.7	6.7 kbps	134
AMR 7.4	7.4 kbps	148
AMR 7.95	7.95 kbps	159
AMR 10.2	10.2 kbps	204
AMR 12.2	12.2 kbps	244

For all of the AMR modes, the sampling frequency of the input audio source is 8 kHz.

One of AMR's advantages is that any sampled audio, such as voice, music, or sound effects, can be stored in compact files. This is the format that the T610 and Z600 series uses to both store and playback sampled audio data.

Quite likely you have sampled audio files that you'd like to convert into the AMR format. To accomplish this, you first need to download the AMR converter archive from the Sony Ericsson web site (see the Further Information and Links table in the Resource Information section of this document). After you expand the archive, you'll get an executable file, `converter.exe`, and two DOS batch files, `con_arm2wav.bat` and `con_wav2arm.bat`. The batch files help control the AMR converter's operation.

To translate a WAV-formatted file with this converter, the input WAV data must be sampled at 8 kHz and stored as 16-bit data. You can use a number of sound conversion programs to translate WAV files sampled at different rates or data sizes into the proper format.

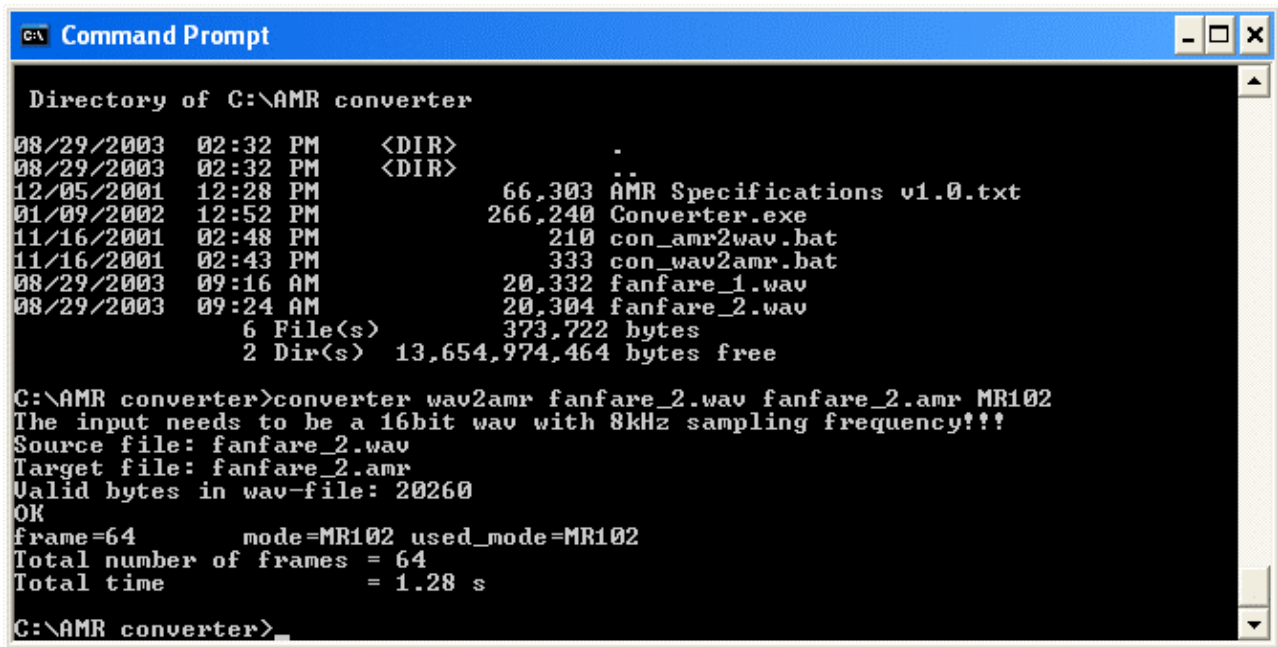
- Important: some audio conversion programs allow you to specify a sample rate of 8000 Hz or 8012 Hz. You must select 8000 Hz for the proper translation of the audio data.

The AMR converter is a CLI-based program that's run from a command prompt window. You supply the batch control file name, the source input file name, the output file name for the translated data, and the desired codec mode as arguments to the converter program. You specify the codec mode using the values MR475, MR515, MR59, MR67, MR74, MR795, MR102, and MR122, which correspond to the bit-rate modes in Table 7. Therefore,

the command line that instructs the converter program to translate a WAV-formatted file into AMR with a 7.4 kbps bit rate would look like:

```
converter wav2amr input_file.wav output_file MR74
```

Figure 9 shows the AMR converter program translating a file, `fanfare_2.wav`, with a 10.2 kbps sample rate.



```
C:\ Command Prompt

Directory of C:\AMR converter
08/29/2003  02:32 PM    <DIR>          .
08/29/2003  02:32 PM    <DIR>          ..
12/05/2001  12:28 PM                66,303 AMR Specifications v1.0.txt
01/09/2002  12:52 PM            266,240 Converter.exe
11/16/2001  02:48 PM                210 con_amr2wav.bat
11/16/2001  02:43 PM                333 con_wav2amr.bat
08/29/2003  09:16 AM            20,332 fanfare_1.wav
08/29/2003  09:24 AM            20,304 fanfare_2.wav
        6 File(s)          373,722 bytes
        2 Dir(s)  13,654,974,464 bytes free

C:\AMR converter>converter wav2amr fanfare_2.wav fanfare_2.amr MR102
The input needs to be a 16bit wav with 8kHz sampling frequency!!!
Source file: fanfare_2.wav
Target file: fanfare_2.amr
Valid bytes in wav-file: 20260
OK
frame=64      mode=MR102 used_mode=MR102
Total number of frames = 64
Total time      = 1.28 s

C:\AMR converter>
```

Figure 9: The AMR converter program in action.

12.Resource Information

12.1. Abbreviations

Acronym	Meaning
API	Application Programming Interface
IDE	Integrated Development Environment
IrDA	Infrared Device Association
J2ME	Java 2 Micro Edition
MMAPI	Mobile Media API
MIDP	Mobile Information Device Profile
SMS	Short Message Service
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
WAP	Wireless Application Protocol

12.2. Further Information and Links

Item	Link
IrDA specification	http://jcp.org/aboutJava/communityprocess/final/jsr118/index.html
J2ME MIDP 2.0 specification	http://jcp.org/aboutJava/communityprocess/final/jsr118/index.html
J2ME MMAPI (JSR-135) specification	http://jcp.org/aboutJava/communityprocess/final/jsr135/
T610/T616/618 White Paper	http://www.ericsson.com/mobilityworld/sub/open/devices/t610/docs/t610_wp
Z600 White Paper	http://www.SonyEricsson.com/developer
Metrowerks / CodeWarrior	http://www.metrowerks.com
Sony Ericsson J2ME SDK	http://www.ericsson.com/mobilityworld/sub/open/devices/p800/docs/j2me_sdk_p800
Sony Ericsson Java	http://www.ericsson.com/mobilityworld/sub/open/technologies/java/docs/java

Developer's Guidelines	_developers_guidelines
Developing MIDlets with Sony Ericsson J2ME™ SDK and Sun ONE Studio, Borland JBuilder or Metrowerks CodeWarrior	Included with Sony Ericsson J2ME SDK
Adapting your MIDlets to the Sony Ericsson T610/618	http://www.ericsson.com/mobilityworld/sub/open/technologies/java/docs/adapting_to_t610
Optimizing J2ME applications for the T610 Sony Ericsson series	http://www.ericsson.com/mobilityworld/sub/open/devices/t610/docs/opt_j2me_apps_t610
Sony Ericsson AMR converter	http://www.ericsson.com/mobilityworld/sub/open/technologies/messaging/tools/amr_converter