Understanding and using Java MIDI audio

Dan Becker

August 2001

This technical article was originally published a previous issue of the IBM DeveloperToolbox Technical Magazine. Subscribe to the IBM developerWorks journal for the latest technical articles on open standards-based technologies that are available to you offline in a printed publication.

In Sun Java 2 Version 1.3, the Java Sound programming interface improved the audio capability of the Java platform tremendously. The January 2001 issue of the IBM DeveloperToolbox Technical Magazine described the sampled audio capabilities of Java Sound and how to play and record Wave (WAV), Audio Utility (AU), Apple Interchange File Format (AIFF), and other sampled audio formats.

This article describes the other half of Java Sound, Music Instrument Digital Interface (MIDI) audio. MIDI plays a useful role in Java programs. It is the most compact form of audio, allowing as little as 25 KB to encode a typical four-minute pop song. Background music and user interface cues and confirmations use MIDI sound extensively. For those with big entertainment requirements and small space budgets, consider a MIDI-based solution. Additionally, the newer Rich Music Format (RMF) allows MIDI music to be encoded and played as the artist intended with rich, realistic sounding instruments and CD audio quality sound. This article provides an overview of the MIDI architecture and musical capabilities of the Java Sound programming interface and how to access the MIDI internal data using three example Java programs.

Java Sound history

First, a little bit of the Java platform audio history. Long ago in the Sun Java Version 1.02 days, Java technology only had the capability to play simple, 8 kHZ sample rate AU format sounds. To this day, the Java applet demos still come with such classic hits as spacemusic.au and yahoo.au.

In Sun Java 2 Version 1.2, Sun Microsystems improved the quality of Java audio by implementing the Headspace Audio Engine by Beatnik Corporation. This allowed Java programmers to use the same audio interfaces, but with the additional capability of playing AU, WAV, AIFF, MIDI, and RMF sounds. Although the sound quality was improved to CD audio levels (up to 44 kHZ sampling rates), there still was no programmatic way to pause and resume a sound, display a progress bar, or get a notification that your sound was completed.

Finally, with Sun Java 2 Version 1.3, the Java Sound programming interface introduced many new capabilities for the audio enthusiast including pause and resume, progress bars, and sound completion events. The software mixer of Java Sound can mix up to 64 channels of sampled or synthesized audio. The

MIDI synthesizer supports wave table synthesis that programmers can access by loading the programmable sound bank. There even is an interface to record and save sampled or MIDI files. Audio enthusiasts welcome these improvements.

However, there still is a way to go. Unfortunately, the audio hardware acceleration is limited, so the Java Sound engine will not take advantage of a fancy audio board. The synthesis and mixing are software based, so playing MIDI audio with Java technology will have more of an effect on your CPU usage than if you play the MIDI file with a native audio program. This is similar to how Java 2D performs great graphic manipulations but does not take advantage of a hardware accelerated video board. By doing the work in software, Java technology gives you cross-platform portability but at the expense of high performance and low CPU utilization.

MIDI demonstration

Let's now look at some demonstrations of what can be done with MIDI audio in Java technology. One of the best demonstrations of Java Sound MIDI capabilities is the Java Sound demo provided at the Sun Java Sound home page or in the demo directory of the Java Developer_s Kit (JDK). The MIDI Synthesizer demo (MidiSynth.java) allows you to play a mock keyboard and to select various instruments to play. As with many MIDI instruments, there are pianos, strings, woodwinds, horns, and a wide array of sound effects. If you are quick with the mouse and good with your timing, there is a record feature that allows you to save your work to disk. The MIDI file can be downloaded to an electronic keyboard or other MIDI instrument. Another demonstration is the Groove Box demo (Groove.java) that allows you to select various percussion instruments and create a looping drumbeat. This demo is useful for testing the polyphony and latency of the MIDI engine, creating entertaining dance grooves, and annoying your coworkers. To run the Java Sound demo, type:

```
java -jar JavaSound.jar
```

MIDI audio architectural overview

The Sun Java 2 Software Developer_s Kit (SDK) contains two excellent documents for understanding the Java Sound architecture and API located in the guide/sound directory:

- API Architecture Overview (guide/sound/arch_overview.html)
- API Programmer's Guide (guide/sound/prog_guide/title.fm.html)

The first document states the design goals and gives an overview of the major components used to create sampled and MIDI sounds. The second document has more in-depth discussion and use of each of the audio classes and shows how to use classes such as the Mixer, Sequencer, and Synthesizer to connect lines and create sound.

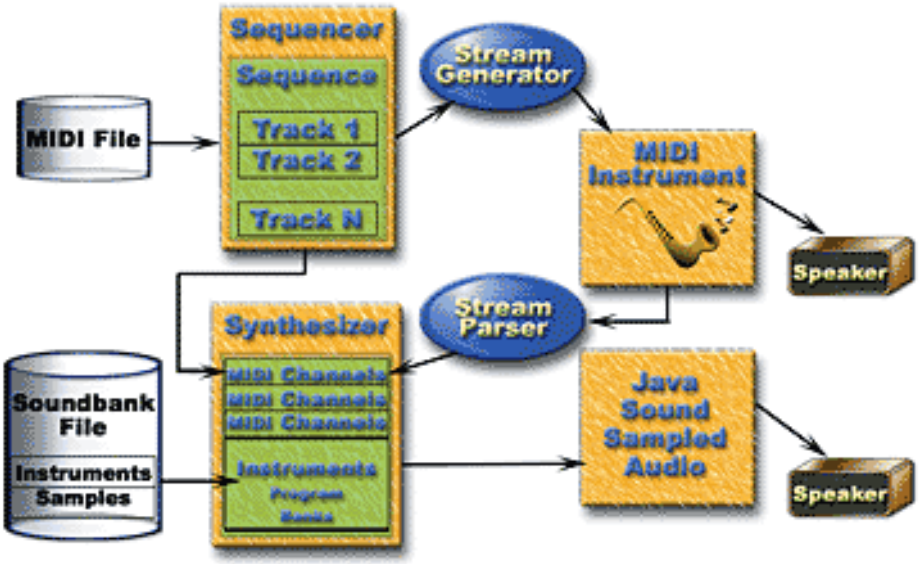
The Java Sound architecture is split into two worlds: sampled audio and MIDI audio. This split is evident in the packaging of the classes into javax.sound.sampled and javax.sound.midi. All public Java Sound classes fall into or under these high-level package names. Additionally, there are many helper and implementation classes in the com.sun.media package. Other audio classes, mostly legacy and older implementations, are located in the sun.audio package.

Because of the limited space of this article, from hereon the focus is specifically on the MIDI audio side of Java Sound and on learning the interfaces and classes to interrogate, edit, and play MIDI audio. The javax.sound.midi.MidiSystem class provides an entry point to the system MIDI resources. Use the MidiSystem static methods to get a Sequencer or Synthesizer object, get information about system devices,

or see what types of devices and files are supported.

Figure 1 illustrates the connections of various portions of the MIDI subsystem. A MIDI file is a collection of tracks with each track having a set of notes. The note has a time stamp, a note value, commands such as note on or note off, and other data associated with the note. The MIDI file is loaded into the Sequencer using the setSequence method. Once loaded, the tracks may be edited with the Java Sound programming interface. From the Sequencer, the MIDI tracks may be "sequenced" or merged together in time order. This stream may be sent to a MIDI instrument through a stream generator. The MIDI instrument can synthesize the MIDI stream into analog sound that can be heard over headphones or a speaker.

Figure 1. Connections of the MIDI subsystem



The MIDI sequence also may be sent to the Synthesizer. In the Synthesizer, the MIDI tracks are loaded into channels. These channels are merged with instrument sounds that are loaded from a sound bank. The instruments are stored internally into an array of Programs and Banks that are used to swap in and out logical groups of instruments. Together the tracks and the instruments may be synthesized into sampled audio. Java Sound takes the sampled audio and sends it to your computer sound card that plays it over your computer speakers or headphones.

A sound bank is loaded into the Synthesizer with the loadAllInstruments method. The quality of the MIDI audio playing on your speakers is dependent on the quality of the instrument sound bank. The default sound bank consists of low sample rate instrument sound to conserve space. Other more high-quality sound banks are available at the Sun Java Sound home page. The sound banks range in size from 0.35 MB to 4.92 MB and go from fair to extremely good quality. A good analogy is to think of a sound bank as an audio "font" for your music, and the MIDI file is the audio "document." Various MIDI files can be played with various sound banks and the results may vary from bad to good with all sorts of variations in between. Because MIDI implementations have dissimilar sound bank implementations, some MIDI file and sound bank combinations may sound positively strange, as one sound bank may contain a sitar instrument where another one has a banjo. This is why MIDI files sound different when played on different sound cards.

RMF differs from MIDI in that an RMF file contains both MIDI track information and its own sound bank. While these files tend to be larger than a regular MIDI file, because they contain instrument sounds specified by the composer, they sound closer to the musical expression the composer intended. Unlike a MIDI file, there is no problem with mixed up lead instruments such as having a sitar playing the lead

melody of a banjo.

The beauty of the Java Sound MIDI programming interface is that any of the MIDI subsystems may be queried, edited, and changed. Thus, all sorts of MIDI programs are possible, anything from beat boxes, to computer MIDI keyboards, to all sorts of endless programmatically generated music. This is very different from the early days of Java technology when MIDI sound was not available at all.

Let's look at some simple example programs to further understand how to take advantage of these audio capabilities in your Java programs. The following three example programs will run on all Sun Java 2 Version 1.3 platforms including all IBM 1.3 releases. They were developed on IBM Developer Kit for Linux, Java 2 Technology Edition, Version 1.3. However, the programs also run on IBM Java virtual machines for AIX (IBM Developer Kit for AIX), OS/2 (IBM OS/2 Warp Developer Kit, Java 2 Technology Edition, Version 1.3), Windows (IBM Developer Kit for Windows, Release 1.3.0), and OS/390 (IBM Developer Kit for OS/390) operating systems.

MIDI audio play example
First, let's see how to add MIDI audio to a Java program through a simple MIDI player. MidiPlay.java is a command-line program that takes one or more MIDI file names as an argument. The files are turned into streams and fed to the Sequencer that sets the sequence and then starts itself. A MIDI event listener listens for the end of the MIDI stream. Once the end of the stream occurs, the sequencer is stopped and the next MIDI file is opened and started.

The most common operations in the code are described here. First, opening a Sequencer for playing a MIDI stream is shown below. This is done with the getSequencer and open methods. Either of these two statements may throw a MidiUnavailableException. Any thrown exceptions are handled by the default exception handler, which will print a stack trace and exit the program. In a product environment, you would want to be more robust with handling errors, but in this example it should be adequate to diagnose any problems.

```
// These methods may throw MidiUnavailableException
Sequencer sequencer = idiSystem.getSequencer();
sequencer.open();
```

Once you have a Sequencer, the next thing you need to do is give it a sequence and start playing. In MidiPlay, a sequence stream is created from the file name given on the command line. Once you have the stream, use the setSequence and the start methods to begin playing. Like the previous example, any exception thrown is handled by the default exception handler and exits the program.

```
// These methods may throw InvalidMidiDataException or // IOException.
sequencer.setSequence( sequenceStream );
sequencer.start();
```

The final action concerns what to do when the MIDI sequence ends. MidiPlay adds MetaEventListener to the Sequencer with the addMetaEventListener method. An "end of stream" is a MIDI meta event that calls the meta method and runs the code of the MetaEventListener. Once the end of the MIDI stream is reached, you typically stop the Sequencer. Here you also close it down and exit the program. You may perform any action you want once your MIDI music ends.

```
// MetaEventListener role
public void meta( MetaMessage event ) {
  if ( event.getType() == 47 ) { // end of stream
     sequencer.stop();
     sequencer.close();
     System.exit( 0 );
}} // meta
```

The MidiPlay example is a good beginning MIDI program. The code is very short and adds a great benefit to any Java program. For instance, using the few short lines above, you may add a running sound track or background music to any program resulting in a more immersive environment. You also may use the code to add such user interface cues as button press confirmations to your Java programs.

MIDI file sequence information example

This next example shows how to inspect or edit the contents of a MIDI file. As stated in the MIDI architecture section, a MIDI file contains the information necessary to play a MIDI song with the instruments programmed in a MIDI synthesizer's sound bank. The MIDI file information primarily consists of a Sequence, which contains a series of tracks, each track having a series of time-stamped events such as notes, pitch bends, and other musical commands. The notes in the track are ordered in time by the MIDI sequencer and sent to a MIDI instrument or a synthesizer for further processing. Other information in the MIDI file includes length, rate, and other housekeeping information.

SequenceInfo.java is an example program that lists the information in a MIDI file. This is done in Java Sound by opening a MIDI file from a file name string, getting a Sequence object from the MidiSystem, and calling various methods to list the sequence information. The code listing below shows how to do this. Notice that once you have a valid Sequence object, there are many pieces of information you can list. These are shown in the various println statements in the listing below.

As in the other examples, opening a MIDI file may throw an IOException or InvalidMidiDataException. This simple program lets the default exception handler catch the mistake, which prints a stack trace and exits the program.

```
File midiFile = new File( fileName );
System.out.println( "Sequence file name: " + fileName );
// Using MidiSystem, convert the file to a sequence.
Sequence sequence = MidiSystem.getSequence( midiFile );
if (sequence != null) {
   // Print sequence information
   System.out.println( " length: " +
     sequence.getTickLength() + " ticks" );
   System.out.println( " duration: " +
     sequence.getMicrosecondLength() +
     " micro seconds" );
   System.out.println( " division type: " +
     divisionTypeToString( sequence.getDivisionType() )
); ...
```

One of the most interesting parts of a MIDI file is the track information and the notes that make up a song. The code in the listing below shows how a Track array is obtained from the MIDI Sequence. Once you have these tracks, you can look at them individually. In this example, the track.get method is used to obtain a series of MidiEvent objects. Each of these MidiEvent objects contains a time stamp (called a tick) and a MIDIMessage that are obtained with the getTick and getMessage calls.

```
// Print track information
Track [] tracks = sequence.getTracks();
if ( tracks != null ) {
   for ( int i = 0; i < tracks.length; i++ ) {
      System.out.println( "Track " + i + ":" );
      Track track = tracks[ i ];
      for ( int j = 0; j < track.size(); j++ ) {
             MidiEvent event = track.get( j );
                 System.out.println " tick "+ event.getTick() + ", " +
          MessageInfo.toString( event.getMessage() ) );
         } // for
    } // for
} // if
```

The MidiMessage class is the super class of one of three MIDI message classes:

- MetaMessage
- SysexMessage
- ShortMessage

A MetaMessage object usually contains information not needed to play the song. For example, this could be the song copyright, the instrument names, or song lyrics. This information is useful but is not essential to playing the song. A SysexMessage object contains system exclusive messages. These messages are rare, and they provide information specific to a subset of available MIDI instruments. These might be experimental messages that are not available on all instruments. Finally, a ShortMessage object contains song information. These messages turn notes on and off, bend the pitch, and change the key pressure. In the SequenceInfo example, the MessageInfo class is used to convert the MidiMessage object into an English text string, suitable for printing.

When you run the SequenceInfo program, you get the MIDI file information in an easy-to-read text file, an example of which is shown here. This is useful to look at, and not much effort is needed to go from just browsing the MIDI file to creating a full-fledged MIDI music editor. To change or edit the notes or the MIDI messages, use the corresponding setter methods to put new information into the MIDI tracks. The Java Sound programming interface allows you to get or set any of the music objects in the Java Sound class hierarchy.

```
Sequence file name: ../sounds/short.mid
  length: 206 ticks
  duration: 10300000 micro seconds
  division type: PPQ
  resolution: 10 ticks per beat
Track 0:
  tick 0, channel 1: program change 21
  tick 16, channel 1: note F3 on velocity: 64
  tick 23, channel 1: note off F3, velocity: 64
  tick 28, end of track
Track 1:
  tick 0, channel 2: program change 34
  tick 18, channel 2: note E2 on velocity: 64
  tick 26, channel 2: note off E2, velocity: 64
  tick 30, end of track
```

Programmatic MIDI music example

This last example shows several useful MIDI techniques. First, it shows how to open a sound bank, query an instrument name, and print the information to the screen. Second, it shows how to play MIDI music on any given instrument, without using a MIDI file as input. This opens the door to all sorts of computerized music generators. Using the basic techniques shown here, you could come up with an endless stream of music for any Java program.

The InstrumentTest.java program is a command-line program that plays a tune on the given range of instruments. For instance, you can play the tune on the first 10 MIDI instruments in the sound bank by saying InstrumentTest 0 9. If you do not know the names or positions of MIDI instruments in the sound bank, you may use the accompanying BankInfo program to list all available instruments. Run java BankInfo soundbank.gm to see what instruments are available. Or, you can just run InstrumentTest on every available instrument by using indexes 0 and 512, but be prepared for the same mind-numbing tune to be played on many instruments.

The InstrumentTest.java program gets and opens a MIDI Synthesizer in the usual manner presented in the earlier examples. The SoundBank.getInstruments method queries the available MIDI instruments from the sound bank. Once you have an array of all available instruments, the instrument is loaded into the Synthesizer with the following code. Notice that it is as simple as loading an instrument into the Synthesizer and calling programChange on the synthesizer Channel.

```
String name = instruments[ i ].getName();
if ( name.endsWith( "\n" ) ) name = name.trim();
System.out.println( "Soundbank instrument " + i + ": " + name );
synthesizer.loadInstrument( instruments[ i ] );
channels[ channelNum ].programChange( i );
playChannel( channels[ channelNum ], notes, velocities, durations );
```

The playChannel method call plays the tune on the MIDI instrument. In InstrumentTest, the tune is stored as an array of notes, velocities, and durations. The playChannel method shown below actually turns the notes

on and off with a sequence of noteOn, sleep, and noteOff method calls.

```
public static void playChannel( MidiChannel channel,
  int [] notes, int [] velocities, int [] durations ) {
  for ( int i = 0; i < notes.length; i++ ) {
     channel.noteOn( notes[ i ], velocities[ i ] );
     try { Thread.sleep( durations[ i ] ); } catch
        ( InterruptedException e ) { }
  } // for
  for ( int i = 0; i < notes.length; i++ )
     channel.noteOff( notes[ i ] );
} // playChannel
```

For your own experimentation, you might want to change the InstrumentTest program to come up with some computer-generated tune. You can modify the program arguments to take a tune as an input or generate the tune based on the day of the year. All sorts of tunes can be generated programmatically, and the ideas are limitless. All this is possible thanks to the programming interface of Java Sound.

Conclusion

This article discusses the Java Sound programming interface, presents the MIDI audio subsystem, and gives several examples for using MIDI audio in a Java program. If you have experience with previous Java releases, you can see how the new interface gives you greater control over the audio. Three simple examples are shown for playing MIDI files, browsing and inspecting MIDI data, and sending a tune to a MIDI instrument; but these are just stepping stones to providing your customers with a rich, immersive sonic environment. There are many more things you can do with Java audio thanks to the new Java Sound programming interface. Hopefully, this article has whetted your appetite for more.

Resources

- developerWorks Java Technology Zone
- Sun Microsystems Java Sound
- Java Sound Examples
- Beatnik Enhanced Audio Solutions

IBM DeveloperToolbox

Find the following article-related category, products, or tools on your DeveloperToolbox CDs.

- Category: Java Tools
- IBM AIX Developer Kit, Java 2 Technology Edition, Version 1.3
- IBM Developer Kit for Linux, Java 2 Technology Edition, Version 1.3
- IBM OS/2 Warp Developer Kit, Java 2 Technology Edition, Version 1.3
- IBM Developer Kit for Windows, Release 1.3

About the author

Dan Becker works in the IBM Software Division in Austin, Texas. He is responsible for the audio subsystem in IBM Java 2 Version 1.3 releases for AIX, Linux, OS/2, System/390, and Windows operating systems. Before that, Dan worked on porting previous Java virtual machines, the multimedia plug-ins for Netscape Navigator for OS/2, and the multimedia parts for OS/2 Warp Version 4.0. You can contact him at beckerdo@us.ibm.com or at his public Web page.