MMAPI Support For EAS

version 1.1 from 07/24/06

Table of Contents

1 Overview

This document describes the architecture and installation of the MMAPI wrapper for EAS. It is targeted to people porting or testing EAS's MMAPI support.

2 Architecture

2.1 General Limitations

Interfacing a C library from a J2ME VM brings some limitations, especially if a broad range of devices should be supported:

Single Thread VM
Some small VMs for devices are single threaded (“Green Threads”). This requires any native code to not block, otherwise the VM will appear frozen. This is important for the EAS_Render() function and length file I/O.

General Thread Handling
A VM may implement thread handling in a different way than native (stack pointers, etc.). Therefore, a native thread shouldn't interact with a Java thread, and vice versa. Also, the respective yield()/sleep() functions cannot be called for a thread of the “other side”.

One Way Native Interface
Some VM's (notably the KVM) only provide an interface to call native C functions from Java, but do not offer a standard way to call Java methods from a C function. This limitation requires a push architecture for data originating from Java, and polling for data originating from the C layer. The implementation uses complex pre-buffering for EAS, since Java data I/O can only push data to the native layer, while EAS implements a “pull” architecture for the media data.

2.2 Workspace Layout

The “mmapi” directory has two sub directories: “classes”, containing the Java source files and “native” for C source files.

2.2.1 mmapi/classes/com

This directory and subdirs contain the sources for the com.sonivox.mmapi package. This package provides the private Java implementation of MMAPI's classes and functionality. The public javax.microedition.media.* classes call into this package.

Only for the integration into Sun's MIDP RI there is one override of a class (com.sun.midp.lcdui.DefaultEventHandler) to remove “magic” code for Sun's ABB implementation.

Config.java

Central configuration for the Java implementation. Select here which control are active for which media type, which media types are activated, etc. Also some numerical settings are here, like the buffer size for streaming media. Most settings must be synchronized with the respective symbols in the native configuration file, eas_mmapi_config.h.

Constants.java
: This file contains constants for the Java implementation, like file type extensions and mime types.

ControlBase.java
: Base class for controls which are implementing the javax.microedition.media.Control interface, and providing a little common code.

ControlMetaData.java
: Implementation of MetaDataControl.

ControlMIDI.java
: Implementation of MIDIControl.

ControlPitch.java
: Implementation of PitchControl.

ControlRate.java
: Implementation of RateControl.

ControlRecord.java
: Implementation of RecordControl.

ControlStopTime.java
: Implementation of StopTimeControl. This is done by monitoring the current media time and stopping the player when the stop time is reached. Some of the implementation is in PlayerEAS.java.

ControlTempo.java
: Stub implementation of TempoControl.

ControlTone.java
: Implementation of ToneControl.

ControlVolume.java
: Implementation of VolumeControl.

DataSourceBase.java
: Base class for implementations of the abstract class DataSource. Provides common functionality, like connected/disconnected handling. Introduces abstract “*Impl” methods for subclasses to implement.

DataSourceCapture.java
: An implementation of DataSource for capture players. This class does not actually provide captured data, but it centralizes all Java code for capture functionality, notably the locator parser to gather the capture format from the locator.

DataSourceHTTP.java
: An implementation of DataSource for HTTP streams. It extends DataSourceInputStream, providing the INputStream from the HTTP connection. Furthermore it implements “SEEKABLE_TO_START” functionality: seek to the beginning of the file is realized by disconnecting and reconnecting to the HTTP file.

DataSourceInputStream.java
: An implementation of DataSource where the data originates from an InputStream. Used for players that are created with Manager.createPlayer(InputStream, String).

DataSourceNone.java
: A data source implementation that does not provide any data. This is used for players that do not get their audio data from Java, e.g. when playing native files. A separate type is necessary for various “instanceof” checks in the Player code.

EAS.java
: Bridge to the native EAS library. It handles initialization and shutdown of the synthesizer, as well as all other communication with the engine.

EventDispatcher.java
: Thread for asynchronously dispatching PlayerListener events.

ManagerEAS.java
: Class to implement javax.microedition.media.Manager methods in the private sonivox package. By implementing these methods in the sonivox package, all other sonivox classes can be made package private. This minimizes security risks.

PlayerBase.java
: A base class for Players. It implements some of the basic functionality like listener handling, state transitions, etc.

PlayerEAS.java
: The implementation of the Player interface for the EAS synth.

PlayTone.java
: Implementation of Manager.playTone() with device://midi (i.e. the MIDIControl). It uses a thread to start/stop the tones.

Security.java
: A basic security framework. Extend here for restricting recording, etc.

SystemTimeBase.java
: The default system time base, as returned by Manager.getSystemTimeBase().

Utils.java
: Common utility methods, especially for parsing URLs.

2.2.2 mmapi/classes/javax

This directory and subdirs contain the public Java source files for MMAPI. The interfaces and pure abstract class files are just taken from the specification. The other classes (like Manager.java) contain actual implementation, that is copyrighted by Sonivox. Usually, all methods just delegate the work to “shadow” classes in com.sonivox.mmapi package. For example, class javax.microedition.media.Manager delegates all methods to same-named methods in com.sonivox.mmapi.ManagerImpl. This architecture facilitates access to the package private classes and methods of package com.sonivox.mmapi.

2.2.3 mmapi/native

This directory contains the C source files for the native MMAPI implementation:

eas_mmapi_config.h
: Central location for selecting features and capabilities of the native MMAPI implementation. Note that some settings need to be synchronized with Config.java.

eas_mmapi_types.h
: Central type and constant definitions for the native implementation.

eas_mmapi.h
: Public declarations for the main functionality. Used for interfacing the native interface functions with the actual implementation.

eas_mmapi.c
: Portable implementation of the functions declared in eas_mmapi.h. Most of the functions directly call the equivalent EAS functions. It does some wrapping of handles to provide further data to the host dependent implementation. It also handles caching of data for the host functions for media data provided by the Java layer.

eas_mmapi_host.c
: Implementation of EAS's host functions. This file is rather complex, since it deals with 3 different modes of media data retrieval:
1. NATIVE (opened with native stdio functions like fopen())
2. MEMORY (media data is pre-loaded to a memory area)
3. STREAM (media data is provided in a circular buffer).
This reference implementation uses separate functions for each of the host functions, plus master functions that just dispatch to the respective NATIVE, MEMORY, or STREAM implementation.
This architecture is not exactly the most optimized one, but it can easily be adapted to different architectures. Also, it re-uses most of the code in the EAS example implementations eas_host.c and eas_hostmm.c, so improvements or existing ports of those files can be easily backported to eas_mmapi_host.c. At least in theory...

eas_mmapi_kvm.c
: This file is the bridge from Java to eas_mmapi.c. It is specific to the KVM and uses the KNI. It calls the respective functions in eas_mmapi.c. If a different native interface is

used, replace this file.

eas_mmapi_midp.c
: This file contains implementation specific to Sun's MIDP 2.0 RI, which requires 2 functions for the vibrator to be implemented. Note that these functions are not fully implemented. Either remove from build if not using Sun's MIDP 2.0 RI, or complete the implementation.

eas_mmapi_wave.c
: MMAPI WAVE writing support (RecordControl). It is based on eas_wave.c. Added the “size of optional data” field to the format chunk, since this is required by some WAVE reading software. This requires the writing capability of host streams.

eas_mmapi_wave.h
: Header file for eas_mmapi_wave.c.

eas_mmapi_windows.c
: This file has the windows specific implementation for the debug RI. It uses a simplistic API defined in eas_mmapi.h for output of the audio data to a file and/or the audio device (see eas_mmapi_conf.h for selection). It uses EAS' eas_wave.c and eas_waveout.c for audio output.

eas_wavein.c and eas_wavein.h
: Windows-specific implementation for accessing the audio capture device and providing raw PCM data from the audio input device.

eas_waveout.c and eas_waveout.h
: Windows-specific implementation for accessing the audio playback device and playing raw PCM data on the system's audio output device.

2.3 Notes about the Code

2.3.1 TODO and FIXME marks

Incomplete code or code that can be extended in future is marked with “TODO”. Development tools like Eclipse and Microsoft Visual Studio can generate a list of code lines with such TODO comments. The same applies to FIXME comments, which mark code that should be reviewed again for a possible bug or unclear intentions.

These marks should not be seen as lack of code quality.

2.3.2 DEBUG flags

Some java files have a boolean private static native DEBUG flag. A centralized DEBUG flag would be nicer, but this model allows fine-grained source level turning on/off debugging for specific modules. Furthermore, setting such a class-private DEBUG flag to false will cause the compiler to completely remove all code in a “if (DEBUG)” block completely from the compiled .class file, saving .class file size and optimizing it by removing these if statements.

The DEBUG code is deliberately kept in the code, because it is a somewhat natural documentation, and it may help for future debugging sessions.

2.3.3 Java Package Encapsulation / Code Security

All implementation classes are package private, where possible. That prevents public access and tightens security.

2.3.4 Java Field Initialization

Field initializations are omitted if they equal 0, null, or false to save .class file size. A comment like "// = null" documents the intentional default initialization.

2.3.5 File Character Encoding

All source files are with DOS newlines, and ASCII character encoding.

2.3.6 Native/Java Encapsulation

All native functions are located in EAS.java. It maintains the eas handle and provides static high level methods for each individual method. The native C function implementations are found in eas_mmapi_kvm.c, which contains a KVM implementation for each native Java method. The KVM implementation is merely a wrapper to call the appropriate functions from eas_mmapi.h. This architecture allows easy switching to a different native interface.

2.3.7 General Commands

A trick is used to reduce the number of native functions. Since every native function requires one public method and one private native method in EAS, plus the native interface function, plus the actual implementation from eas_mmapi.h, the “general command” trick reduces code size and increases maintainability: for functions with 0 or 1 integer parameter returning void or integer, one single native method is used, with an additional command parameter. The command parameter is evaluated in native, specifying the actual function to execute. In particular for getters/setters like EAS.getMode() this is saving code size.

For readability, this concept is not maxed out, but for optimization many more methods in EAS can be converted to use the general command native wrapper.

2.3.8 Usage of Host Functions from eas_mmapi.c

The native implementation in eas_mmapi.c calls some of the host implementation functions directly, like EAS_HWMalloc().

Since eas_mmapi.c does not have access to the instance handle, it cannot call these functions with a valid instance handle as first parameter. So for implementations requiring the instance handle in EAS_HWMalloc() and EAS_HWFree(), the code needs to be modified.

For some advanced functionality, eas_mmapi_host.c stores the host instance handle and host file handle in the locator structure, so that eas_mmapi.c can access the host file functions directly. This is used for switching the buffering mode from STREAM to MEMORY (function MMAPI_HWSwitchToMemoryMode()) and for writing to a native file for RecordControl.

2.3.9 Looping support

MMAPI mandates looping (repeat) support. At the end of media, the Player needs to loop back, sending an END_OF_MEDIA event followed by a STARTED event. So the Java layer needs to keep track of the end of media, and when EAS loops back to the beginning. For that, the maintenance thread regularly checks the current repeat counter from EAS_GetRepeat(). Each time that it changes, the events are sent to the listeners. This architecture guarantees the correct number of events even with very short media files (which repeat many times per second).

Now when setting the repeat count to infinite (-1), EAS will not decrement the repeat count as returned by EAS_GetRepeat(). This causes the algorithm above to not work. To work around this, the implementation in class PlayerEAS will set the EAS repeat count to a high value (PlayerEAS.INFINITE_LOOPCOUNT) instead. As a safeguard, the repeat count is bumped up if it gets too low.

2.3.10 Duration handling

Since EAS does not have a designated getDuration() function, and EAS_ParseMetaData() function is potentially time-consuming, an on-demand mechanism is used for retrieving the media duration:

Only if Player.getDuration() is explicitly called, the native function MMAPI_getDuration() is called, which will invoke EAS_ParseMetaData(). This is done from PlayerEAS.calcDuration().

Also retrieving meta data will first invoke EAS_ParseMetaData(). For this, ControlMetaData will call PlayerEAS.calcDuration() after retrieving meta data.

If the duration is retrieved in REALIZED state or higher, a DURATION_UPDATED event is sent to listeners of the Player. This is done in PlayerBase.setDuration().

2.3.11 Manager.playTone

This method is not well suited for a native synthesizer, because it has no methods for explicitly opening/closing the engine. Instead, a timer needs to care for opening the engine, playing the tone, wait, stopping the tone, and closing the engine. If additional tones are played, they need to be queued and played after the already queued/sounding tones.

In this implementation, Manager.playTone is implemented in the class com.sonivox.mmapi.PlayTone, which uses a MIDIControl instance retrieved by way of Manager.createPlayer(Manager.MIDI_DEVICE_LOCATOR). A Java thread is used to dispatch the queued tones as MIDI events to the engine. After the queue is fully played, the

thread waits for some more seconds, for the case that a new tone is issued shortly after the other tones. If that does not happen, the thread closes the player (which causes closure of the EAS engine) and finishes itself.

2.3.12 Recording

Recording in MMAPI means saving a currently playing stream to a file or a Java OutputStream. This is done with RecordControl, which can be armed either with a URL for saving to a local file (or possibly an http POST operation), or with an OutputStream which will receive the recorded data as it is being played back.

This implementation allows saving to a local file (with the file:// pseudo protocol in the URL), or recording to an OutputStream. Currently, it works with WAVE streams only, i.e. WAVE files or a WAVE capture stream from MMAPI's capture://audio special locator.

Recording is implemented mostly in the native implementation. It hooks into the host read operation (i.e. EAS_HWReadFile()): each time data is read, it is written to the record stream, too. Now the record stream is a host file directly opened with EAS_HWOpenFile(). For recording to a local file, the file is opened with the locator as parameter (mode OPEN_MODE_NATIVE), for the OutputStream operation, the file is opened in OPEN_MODE_STREAM, using a circular buffer which EAS_HWReadFile() writes the audio data to. A WAVE header is written before the first audio bytes are written.

This architecture, where all data is processed in native, has the advantage that for most combinations the data remains in native. Double-buffering is only necessary for recording to the OutputStream. This will be impossible to prevent, since pushing data from native to Java is not possible.

For file mode, the WAVE header is patched when calling commit(), so that the chunk size fields are corrected.

For OutputStream mode, the Java PlayerEAS class regularly reads the data from the circular buffer and writes it to the OutputStream. Since it is impossible to seek to the beginning of the OutputStream and patch the header upon completion (commit()) of the recording, the header will have 0xFFFFFFFF in the length fields of the RIFF file header, and the data chunk header. This follows the WAVE file format specification, and most players handle it gracefully. However, EAS defines 0x80000000 to denote “unknown length”. So, in order that files recorded to OutputStream can be played back with EAS, currently the header fields are set to 0x80000000 when recording to OutputStream.

2.3.13 Capture

In MMAPI terms, Capture means live capture from an audio input device. Capturing in MMAPI is done by opening a player with a magic locator “capture://audio”, which also allows selection of the audio device and capture format. This player will control the capture device, and when started, it will capture live audio data and immediately play it. Usually, a RecordControl is used to store the captured audio data.

The implementation assumes that captured audio data needs to be buffered by the host implementation. For the Windows implementation, this is not strictly necessary, since the WaveIn module already buffers data.

Capture is entirely implemented in native, therefore the DataSourceCapture class, which is used for capture, does not provide any audio data.

The capture device is opened along with the “normal” stream open. It is opened in STREAM mode. The Render thread is used to read data from the capture device and write it to the stream. Therefore, the handle to the stream is stored in MMAPI_DATA_STRUCT's field captureStream. This assumes that only one stream is capturing at any given time.

2.3.14 TCK Compatibility

This implementation was not tested with Sun's compatibility test suite. For an official port of MMAPI, an implementation must pass the TCK.

2.4 Known Problems

2.4.1 WAVE playback issues

1. **Distorted 8-bit files:** it is assumed to be a bug in EAS (interpreting 8-bit files as signed).
2. **Fast Forward/Rewind does not work**: Streaming WAVE playback (i.e. the data is downloaded while the file is already played) will not allow to change the playback position before the still buffered audio data, and after the already buffered data. The only exception is that you can rewind to the beginning, where the implementation will reopen the http stream and reinitialize EAS.
3. **Repositioning will always position to 0**: it is believed that this is a problem in EAS.

2.4.2 Recording Issues

1. When recording to a wave file, while playing a wave file, the **header has a wrong format** and causes distorted, slow or fast playback: this is due to a missing implementation of EAS_GetWaveFmtChunk(). A default format of 16-bit mono at 8000Hz will be assumed, unless you capture, where the capture format will be used for all subsequent recordings. Undefine the symbol MMAPI_DEBUG_USE_FORMAT_QUERY_STUB in eas_mmapi_config.h when EAS_GetWaveFmtChunk() is correctly implemented.
2. **Recording to OutputStream generates corrupt header**: This is due to the 0x80000000 flag for “unknown length” required by EAS. See Recording 2.3.12 above, and the symbolic constant MMAPI_CAPTURE_STREAMING_WORKAROUND in eas_mmapi_config.h and its usage in eas_mmapi.c for more information.

2.4.3 Capture time limit

Due to EAS' current internal handling of streaming wave files, capture is limited to 0x7FFFFFFF samples. In practice, this limit should rarely be encountered.

2.4.4 Buffer Underrun handling

Currently, the host implementation handles buffer underruns in STREAM mode by filling some 0 into the buffer. This will prevent that EAS closes down a stream only because streaming cannot keep up providing data. This brings 3 problems:

1. The underrun is not faded out, so an abrupt change to silence can cause a click.
2. 8-bit files usually use unsigned samples, so the inserted 0 will be interpreted as -128 samples. This will produce loud clicks at the beginning and end of the inserted data, and treat the speaker badly.
3. The private circular buffer for recording to OutputStream must not provide 0's, because there is no other synchronization, and the OutputStream would get endlessly filled with zeroes. To prevent this, a special flag in the MMAPI_MediaBuffer structure (noSilenceOnUnderrun) is used to prevent the host STREAM implementation to do any underrun handling if this flag is set.

All such underrun handling is only used if the symbolic constant MMAPI_PROVIDE_SILENCE_ON_UNDERRUN is defined in eas_mmapi_config.h.

2.4.5 ToneControl crash

When using interactive MIDI from the device://midi special locator (Simple Tones), the emulator crashes. This seems to be a bug in the latest EAS release. It worked until, including, the release from 7/20.

2.4.6 Hanging Notes with JTS files or ToneControl

This seems to be an issue in EAS: when locating during JTS or ToneControl playback, single notes may hang and continue to play until the player is closed.

2.5 Troubleshooting

2.5.1 Enabling Debugging

The following places are used to enable debugging:

1. %BASE%\make\Defs.gmk: modify to have this line:
   SONIVOX_DEBUG = true
2. Set EAS debugging level in eas_mmapi_config.h:
   #define MMAPI_DEBUG_EAS_DEBUG_LEVEL 2

3. private DEBUG fields in the following classes: DataSourceCapture, EAS, EventDispatcher, ManagerEAS, PlayerBase, PlayerEAS, PlayTone.
4. More debugging in eas_mmapi.c:
   SONIVOX_DEBUG_RENDER: debug output in the render function
   SONIVOX_DEBUG_STATE: more debugging of the current player state
5. More debugging in eas_mmapi_host.c:
   SONIVOX_DEBUG_IO: debug read/write calls
6. eas_mmapi_config.h also provides some other debugging flags. Use with care!
7. To activate debugging in the “mmademo” test MIDlet, set DEBUG to true in %BASE%\test\mmademo\src\example\mmademo\Utils.java.

2.5.2 Stuttering wave playback

If you encounter this, then streaming is not fast enough to keep up with the rate that EAS reads the wave data from the host interface. Increasing the wave stream buffers (MMAPI_STREAM_CIRCULAR_BUFFER_SIZE in eas_mmapi_config.h and STREAM_BUFFER_SIZE in Config.java) should help.

3 Windows Test Workspace

The test workspace embeds Sonivox's MMAPI implementation into Sun's publicly available MIDP 2.0 RI and Sun's publicly available CLDC VM (the KVM). Make sure to understand the license when downloading Sun's source code. You may get involuntarily “tainted”.

The build is embedded into the MIDP build by using hooks provided by the MIDP build system, and some magic where the hooks were not sufficient.

3.1 Development Tools

3.1.1 CYGWIN

Install Cygwin from `http://www.cygwin.com/` to e.g. `C:\cygwin`

Choose the default files (BASE), plus:

- Archive*
- Devel\make
- Devel\libiconv
- Devel\mktemp
- Utils\bzip2
- Utils\cygutils

For the build, add `C:\cygwin\bin` to the PATH:

```
set PATH=C:\cygwin\bin;%PATH%
```

3.1.2 Microsoft Visual C++

Install Visual Studio or Visual C++. This is necessary for compilation of the native C files. It should be possible to compile with GCC (some Makefile tweaking necessary), but this is not tested.

For the build, add VC to the PATH:

For Visual Studio 2003 .NET:

```
call "C:\Program Files\Microsoft Visual Studio .NET
      2003\Common7\Tools\vsvars32.bat"
```

For Visual C++ 2005 Express Edition (you may need to adapt paths):

```
call "C:\Program Files\Microsoft Visual Studio 8\VC\vcvarsall.bat"
echo Adding platform SDK (missing in VC 2005 Express)
set SDKDIR=C:\Program Files\Microsoft Platform SDK
set INCLUDE=%SDKDIR%\include;%INCLUDE%
```

```
set LIB=%SDKDIR%\lib;%LIB%
```

3.1.3 Sun's JDK 1.3.1

Download Sun's JDK 1.3.1 from `http://java.sun.com/j2se/1.3/download.html` and install to `C:\JDK1.3.1`

If you install to another directory, need to set `ALT_BOOTDIR` before compiling, e.g.

```
set ALT_BOOTDIR=C:/Sun/jdk1.3.1
```

NOTE: use forward slashes as directory separator

3.2 Workspace Setup

The development workspace contains Sonivox's source files and the source files of Sun's RI's.

Create an arbitrary directory where the workspace will be installed, e.g. `C:\MMAPI` (avoid spaces in the name). In the following, this base directory will be referred to as %BASE%.

3.2.1 Sonivox's MMAPI Files

Unzip sonivox_mmapi.zip to %BASE%. It'll create these dirs with subdirs and files:

```
%BASE%\make                    # makefiles
%BASE%\Sonivox\mmapi\classes   # Sonivox's MMAPI Java files
%BASE%\Sonivox\mmapi\native    # Sonivox's MMAPI native C files
%BASE%\Sonivox\java_lib        # MIDP lib for Java development
                               # tools integration (optional)
%BASE%\test                    # test MIDlet
```

3.2.2 Sonivox's EAS Files

Unzip/copy Sonivox's EAS workspace (EASWin32Lib.zip) to %BASE%\Sonivox so that it'll create at least these 2 required dirs:

```
%BASE%\Sonivox\docs            # EAS documentation
%BASE%\Sonivox\host_src        # EAS .h and .c files
%BASE%\Sonivox\lib             # EAS library in easwt.lib
```

If the EAS files should be picked up from a different location, use this define:

```
set ALT_EAS_DIR=C:/My_EAS_Workspace
```

This requires that `C:/My_EAS_Workspace/host_src` and `C:/My_EAS_Workspace/lib` exist.

NOTE: use forward slashes as directory separator.

3.2.3 KVM Source Code

Download Sun's CLDC 1.0.4 Reference Implementation (RI) from `http://java.sun.com/products/cldc/` . it'll be saved as j2me_cldc-1_0_4-src-winunix.zip .

Extract it in %BASE%\Sun so that it'll create

```
%BASE%\Sun\j2me_cldc
%BASE%\Sun\j2me_cldc\api
%BASE%\Sun\j2me_cldc\bin
...
```

Overriding this location is possible with ALT_KVM_DIR (again, use forward slashes as directory separator).

3.2.4 MIDP 2.0 RI Source Code

Download MIDP 2.0 from `http://java.sun.com/products/midp/` . It'll be saved as midp-2_0-src-windows-i686.zip .

Extract it in %BASE%\Sun so that it'll create

```
%BASE%\Sun\midp2.0fcs
%BASE%\Sun\midp2.0fcs\appdb
%BASE%\Sun\midp2.0fcs\bin
...
```

Overriding this location is possible with ALT_MIDP_DIR (also here, use forward slashes as directory separator).

3.3 Building the Emulator

The MIDP RI creates a phone emulator which can be used to test MIDP's features. This workspace will create a MIDP phone emulator with the EAS and Sonivox's MMAPI implementation embedded.

Compilation uses GNU Make, supplied with the Cygwin installation. The bootstrap Makefile is

in %BASE%\make\Makefile. It sets some constants, changes the current directory and delegates to %BASE%\make\Build.Makefile.gmk.

3.3.1 Make

Change directory to %BASE%\make and run make.

During the build, you will see several “deprecated” messages from the Java compiler. These messages are OK and cannot be circumvented.

All build output goes to %BASE%\build.

I use this script in %BASE% to initiate the build:

```
@echo off
set PATH=C:\cygwin\bin;%PATH%
call "C:\Program Files\Microsoft Visual Studio 8\VC\vcvarsall.bat"
echo Adding platform SDK (missing in VC 2005 Express)
set SDKDIR=C:\Program Files\Microsoft Platform SDK
set INCLUDE=%SDKDIR%\include;%INCLUDE%
set LIB=%SDKDIR%\lib;%LIB%
rem set ALT_BOOTDIR=E:/JDKs/jdk1.3.1_07
cd make
make
```

3.3.2 Other Make Targets

1. all or midp: full build
2. clean: remove the entire build output directory
3. midp_quick: can be called after a successful full build to only recompile the C files (using 2 concurrent compile processes to benefit of dual core machines).

3.4 Run the Emulator

The emulator's executable is %BASE%\build\bin\midp.exe. It can be started without command line arguments without much benefit.

3.4.1 Run mmademo

The directory %BASE%\test contains a MMAPI demo application: mmademo. It is taken from Sun's WTK, downloadable (including source) from http://java.sun.com/products/sjwtoolkit/ . The mmademo is considerably improved for extended testing. To run the emulator with the mmademo pre-installed, run this batch script from %BASE%:

```
set MMAPI=test\mmademo\bin
set NAME=mmademo
set MIDP=build\bin\midp.exe
%MIDP% -heapsize 2000k \
       -classpath %MMAPI%\%NAME%.jar
       -descriptor %MMAPI%\%NAME%Test.jad
```

3.4.2 Simple Tones

The first applet in mmademo is Simple Tones. It provides 4 test apps:

1. Short Single Tone: uses Manager.playTone() to play a short beep.
2. Long Single Tone: uses Manager.playTone() to play a long beep.
3. Short MIDI event: uses the device://midi player to play a chord, using MIDIControl.shortMidiEvent().
4. A small interactive MIDI app that maps all numeric phone keys to GM drum sounds, using MIDIControl.shortMidiEvent() from a device://midi player.

3.4.3 Simple Player

In the mmademo, choose the Simple Player. It'll present a list of pre-configured media which you can choose to test the MMAPI implementation. After selection of the media file, the player screen is displayed and the file is played back. If everything works smoothly. Use the numeric keypad's keys to conveniently control the player:

- **2** toggle start/stop (Player.start()/Player.stop())
- **1** rewind 10 seconds (or **left** cursor) (Player.setMediaTime())
- **3** fast forward 10 seconds (or **right** cursor) (Player.setMediaTime())
- **Up/Down**: transpose (if available) (PitchControl)
- *** or # (/)**: volume down/up (VolumeControl)
- **0**: mute/unmute (VolumeControl)
- **4/6** to reduce/increase playback rate (RateControl)
- **5**: stop and rewind to the beginning

Use the menu to access additional functionality:

- toggle loop mode (no repetitions, 3 repetitions, infinite) [Player.setLoop()]
- access meta data [MetaDataControl]
- fine tune rate, pitch, volume.
- record to file [RecordControl]

You can also enter an own URL to play back media from your server, or local files with the file protocol.

3.4.4 Adapt mmademo

To change the list of pre-configured files and URL's for the player, edit `%MMAPI%\%NAME%.jad` with a text editor. A useful jad file is `%MMAPI%\mmademoTest.jad`, which has a useful selection of media files for testing the MMAPI implementation. The http files are on the private server of Florian Bomers.

You can use the WTK to conveniently patch/modify the mmademo, build the jar (preverifier necessary!), and/or get other demos to run on the Sonivox MIDP emulator. For debugging, it is useful to set the DEBUG flag to true in `mmademo\src\example\mmademo\Utils.java`.

For compiling mmademo, open it as a project. Then select from the menu Project|Build. This will compile the Java source files and preverify them for the KVM. If that's successful, choose Project|Package|Create Package. This will create the mmademo.jar in the bin directory.

The WTK also has some documentation of the mmademo example application in `C:\WTK22\docs\UserGuide.pdf`, chapter *Application Documentations\A6.mmademo*.

3.5 Feature Tests

This chapter will show how to test each feature. All the tests are done with the mmademo (chapter 3.4), running the mmademoTest.jad file.

3.5.1 Protocol support

1. The **http** protocol is tested directly with the corresponding media entries.
2. For the **file** protocol, use “Enter URL” and enter a local filename, prefixed with “file://”, e.g. “file://C:\Media\test.wav”.
3. The media files marked with [jar] pseudo-protocol are taken from the dsitribution jar, and therefore they serve to test opening a player with an **InputStream** (i.e. Manager.createPlayer(InputStream, String).
4. The two capture entries in the playlist test the **capture** pseudo protocol, one without and one with parameters to define the capture audio format. When a capture player is active, you should hear the captured audio data on the speakers, with a slight delay. The capture player will use Windows' default device.

3.5.2 Encoding Support

The encodings **audio/x-wav**, **audio/midi** and **audio/xmf** can be tested with the respective files in Simple Player. The **audio/x-tone-seq** type can be tested with the ring tone files.

3.5.3 Streaming Sampled Audio

Every wave file is played back in streaming mode. If the file fits entirely into memory, it is converted to a “Memory” file. Currently, the memory is approx. 50KB. So for short wave files, it is possible to jump to any position, use the loop feature, etc. For larger files, repositioning is only possible in the limits of the currently buffered audio data.

This does not apply to files opened from file.

3.5.4 MIDI/mXMF

All MIDI and (m)XMF files are loaded first into native memory, and then played (MEMORY host type). This does not apply to files opened from file.

3.5.5 Manager.playTone

To test Manager.playTone, use the first MIDlet, SimpleTones. The first 2 menu entries will use Manager.playTone to play a short/long tone.

3.5.6 Tone Sequence

Tone sequences can be playaed back in 2 different ways: by opening a .jts file in the player, or by using ToneControl (see below). To test .jts support, open the horrible Beethoven rendition, listed as “JTS ringtone [jar]”.

3.5.7 Interactive MIDI (MIDIControl without query support)

See MIDIControl below.

3.5.8 Audio Recording (write streamed audio to file)

See RecordControl below.

3.5.9 VolumeControl

Volume control can be tested at any time by clicking on the * (softer), 0 (toggle mute) and # (louder) keys. Furthermore, there is a graphical “Volume” chooser in the menu.

3.5.10 StopTimeControl

Test the functionality of StopTimeControl by playing a media file longer than 5 seconds and selecting “Stop in 5 seconds” from the menu. After 5 seconds, playback should stop.

3.5.11 TempoControl for MIDI and/or Tone

This is not implemented in EAS. If it was available, it could be tested with a corresponding entry in the menu. This entry is only displayed if the player supports TempoControl.

3.5.12 PitchControl for MIDI and/or Tone

Open a MIDI or Tone file and press the up/down keys: pitch should shift up or down. There is also a “Pitch” selector in the menu.

3.5.13 RateControl for MIDI and/or Tone

Open a MIDI or Tone file and use the 4 (slower) and 6 (faster) keys, or use the menu's “Rate” function.

3.5.14 ToneControl

ToneControl enables to play jts files from an array: first a player is retrieved from Manager with the TONE_DEVICE_LOCATOR magic locator. This player cannot be started unless a ToneControl is retrieved from it and it is fed with a jts sequence from a byte array.

The Simple Player has a RTTTL to JTS converter, which uses ToneControl to create a player with the converted RTTTL file. Therefore it is sufficient to open a RTTTL file (must have .txt extension) in Simple Player to test usage of ToneControl. The playlist entry “ToneControl ringtone [http]” will load such an RTTTL file and it can be used to test the implementation of ToneControl.

3.5.15 MIDIControl

There are 2 ways to retrieve a MIDIControl instance:

1) by way of the magic locator “device://midi” (Manager.MIDI_DEVICE_LOCATOR).
2) by way of retrieving a MIDIControl from an existing MIDI file player.

The first way is tested with the Simple Tones applet: “Short MIDI event” and “MMAPI Drummer” will both use MIDIControl from the magic locator.

The second way is tested by opening a MIDI file in the Simple Player (e.g. MIDI scale), then choose “MIDIControl Test” from the menu. It will retrieve a MIDIControl from the player. If you hear a chord being played, then the test is successful.

3.5.16 MetaDataControl

Simple Player displays any meta data if you select “Meta Data” from the menu of a player.

3.5.17 RecordControl

RecordControl will record the currently playing audio stream to a file or an OutputStream. This works for any playback of audio/-x-wav encoding. In the current implementation this means WAVE playback and capture.

For testing **recording to a file**, open a capture player, or a wave file player, start playback and choose “Start Recording”. In the following field choose “file://C:\recording.wav”. From now on, everything that is played, is also written to the file. You can stop the player, rewind, etc. Only if you choose “Stop Recording” from the player menu will the file be finalized. If you quit the player before doing so, the recorded file will probably be useless (this is according to the spec). After selecting “Stop Recording”, go back to the Simple Player playlist and open the 3rd entry, “WAV C:\recording.wav” which will then play the just recorded audio data.

For testing **recording to an OutputStream**, open a audio/x-wav player as above, then as recording URL type “rms:/record.wav” (or any name with this pseudo protocol and this extension). RMS is a MIDP specific storage accessible from J2ME instead of a real file system. Once recording is started, all audio data is written to a ByteArrayOutputStream, so make sure to not record for a long time, otherwise you'll risk an OutOfMemoryException and the recording will be corrupt. When you choose “Stop Recording”, a RMS record store is created, and the ByteArrayOutputStream, converted to an array, is written to the record store. Upon successful completion, the record store index is displayed. To play back the file, go back to the Simple Player play list, and choose “Browse RMS”. You should see the entry “record.wav”. Selecting it will display the index to which you just recorded, along with its size in bytes. Selecting it will play the file. In the RMS browser you can also delete record stores.

3.5.18 RateControl for audio

Currently, this is not implemented by EAS. If it will be, you can use the same test procedure as for testing RateControl for MIDI/Tone.

3.5.19 Synchronization of two streams (GetTimeBase/SetTimeBase)

The Simple Player does not provide any means to test this functionality.

3.5.20 Security

Security is only implemented as stub and cannot be tested. It usually requires cooperation with the MIDP implementation (e.g. to hook into any security related functions and display a corresponding choice to the user, as is done in the test emulator for http connections).

4 Extending the Implementation

4.1 Enabling TempoControl

For TempoControl, stubs are implementred. This is the class ControlTempo in package com.sonivox.mmapi. To enable it, set Config.HAS_MIDITONE_TEMPOCONTROL to true. Then, in the native file eas_mmapi_config.h, uncomment the line which defines the symbol MMAPI_HAS_TEMPO_CONTROL. You will have to adapt eas_mmapi.c to use the correct function name for the equivalent EAS functions for setting/getting the tempo in milli-bpm.

4.2 Adding a new protocol

Quick check list:

1. Add a new PROTO_* constant in Constants.java
2. If you want to make this an optional protocol, add a switch HAS_* to Config.java.
3. If it will use Java based audio data, create a new DataSource implementation, based on DataSourceBase.
4. In the class ManagerEAS, add support for the new protocol in the methods getSupportedContentTypes() and getSupportedProtocols(). Add code to create the specific data source at the bottom of createPlayer(String). There you also may need to add special handling (as is done for the capture and device pseudo protocols).
5. In PlayerEAS, you will probably need to add handling of your protocol in method realizeImpl().
6. If you need to add a new content type, see bwlo “Adding a new Content Type”
7. If you need to add a new player type, see below “Adding a new File Type”

4.3 Adding a new Content Type

Quick check list:

1. Constants.java: add the constants for the content type(s)
2. if necessary, add the file extension(s) for the content type
3. in ManagerEAS.getSupportedContentTypes(), and ManagerEAS.getSupportedProtocols(), add support for this new content type
4. in ManagerEAS.guessContentType() add code for matching extension to the new content type(s)

4.4 Adding a new File Type

Quick check list:

1. Config.java: add a feature selector of the scheme HAS_*_PLAYBACK, e.g. HAS_XMF_PLAYBACK
2. Possibly add the content type(s) matching this file type (see 4.3)
3. in class PlayerEAS, add a new TYPE_* constant for this new file type
4. in ManagerEAS.getPlayerType() add code to return the new player type
5. in PlayerEAS.getMode(), add code for selecting the stream mode for this file type. Larger streamable file types should have STREAM mode, all others MEMORY.
6. if this file type supports any controls, modify PlayerEAS.getControlImpl()

4.5 Adding meta data types

The native layer is independent of the available meta data types in EAS. However, the Java layer needs to be updated for new meta data types:

4.5.1 EAS.java

In EAS.java, update the list of METADATA_ constants with the new types.

4.5.2 ControlMetaData.java

Check and update the function metaDataConstantToKey() that translates EAS meta data constants to String. Make sure to consult the specification of MMAPI's MetaDataControl to check for MMAPI's pre-defined meta data types.

4.6 Adding a Capture Format

The current implementation only allows PCM, 8-bit unsigned, 16 bit signed little endian, in mono or stereo as capture formats. To add support for other formats (e.g. ADPCM, ulaw, etc.), follow the following steps:

4.6.1 EAS.java

If a new encoding is required, add a new constant for the encoding in the format of CAPTURE_ENCODING_*.

4.6.2 DataSourceCapture.java

This is the class that parses the locator. Add support for the new format in the method parseLocator(). Use the EAS.CAPTURE_ENCODING_* constants for the encoding.

4.6.3 eas_mmapi.h

Add the equivalent encoding constant to the MMAPI_CAPTURE_ENCODING enumeration.

4.6.4 eas_mmapi.c

Verify that MMAPI_OpenCapture() correctly works with the new format.

4.6.5 eas_mmapi_wave.h and eas_mmapi_wave.c

If modifying a WAVE format, add the code to construct the correct fmt_ chunk in the function WAVE_FillFormat(). If you require a new WAVE encoding, define the corresponding WAVE_FORMAT_TAG_* and extend the “case” statement in WAVE_FillFormat(). You may need to adapt WAVE_WriteHeaderImpl() and FlipWaveHeader() to write extended header information.

4.7 Changing the default Capture Format

MMAPI's capture is initiated with a special locator “capture://audio”. The capture audio format can be encoded in the URL, but if not, the implementation provides a default capture format. This default format is hardcoded as constants in class DataSourceCapture.java. It must be asserted that at least the default capture format works on the target platform.

4.8 Adding Java security

Security in MMAPI is implemented by throwing the respective exceptions upon a security violation. A stub class, com.sonivox.mmapi.Security, is included where security checks can be made. This class provides stub methods for checking locators from which players are about to be created. Also, starting recording can be intercepted.

For user friendly security implementation, a tighter integration with the MIDP implementation is necessary.

4.9 Changing Settings in the Make System

Most defines and settings for the build are found in %BASE%\make\Defs.gmk. Additional defines can be added to the constant SONIVOX_DEFINES. New EAS source files can be added to the constant SONIVOX_EAS_SRC. %BASE%\make\Sonivox.gmk contains the main integration of Sonivox's MMAPI implementation. It is unlikely that you need to change anything here. Sonivox-post.gmk contains more code to integrate the Sonivox MMAPI implementation into the MIDP reference implementation.

5 Acronyms

Acronym	*Expanded name*	*Notes*
ABB	Audio Building Block	Subset of MMAPI, part of MIDP 2.0
API	Application Programming Interface	
BPM	beats per minute	
CDC	Connected Device Configuration	Mid-size J2ME (PDA's)
CLDC	Connected, Limited Device Configuration	Smallest J2ME (cell phones)
EAS	Embedded Audio Synthesis	
I/O	Input/Output	
J2ME	Java 2, Micro Edition	
JDK	Java Development Kit	
JNI	Java Native Interface	
JVM	Java Virtual Machine	
KNI	KVM Native Interface	
KVM	K Virtual Machine	Sun's VM for CLDC
MIDlets	Applications for MIDP	
MIDP	Mobile Information Device Profile	
MMAPI	Mobile Media API	
PDA	Personal Digital Assistant	
RI	reference implementation	
TCK	Technology Compatibility Kit	Sun's test suite to approve compatibility
WTK	Wireless Toolkit	Sun's latest binary MIDP RI