Porting MIDP

MIDP Reference Implementation, Version 2.0 FCS

Java™ 2 Platform, Micro Edition

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, California 95054
U.S.A. 650-960-1300

November, 2002

Contents

Tables

Preface

Porting MIDP describes how to port the MIDP Reference Implementation to a mobile device. By the end of chapter two you should have a simple, although perhaps not entirely functional, port of the MIDP Reference Implementation. It will not be tailored for your device. The remaining chapters will help you make your port fully functional and customized for your device. The chapters typically include details of how to port any native code associated with a module (such as persistent storage, or the audio building block) and suggestions for customizing it.

Because this guide includes customization information, Porting MIDP could be useful for optimizing a MIDP implementation for increased speed or decreased footprint, as well as being useful for porting. The organization of the guide, however, reflects the tasks for porting MIDP to a new device.

This guide assumes that you have already installed the product, as described in *Installing MIDP*, and that you are familiar with the *MIDP 2.0 Specification*. You can download the specification from `http://jcp.org/jsr/detail/118.jsp`.

How This Book Is Organized

This book has the following chapters:

Chapter 1 introduces MIDP and its architecture, and the high-level requirements for porting MIDP to a new device.

Chapter 2 describes the first steps in porting MIDP to a new platform.

Chapter 3 describes the event model.

Chapter 4 gives you information on porting the persistent storage module, the record management system (RMS)

Chapter 5 gives advice on making a port of the user-interface thread-safe.

Chapter 6 covers some topics in porting the MIDP user-interface, including the `game` package.

Chapter 7 discusses the security functionality of the MIDP Reference Implementation.

Chapter 8 discusses porting the MIDP network protocols.

Chapter 9 gives you information on porting the persistent storage module, the application management system (AMS).

Chapter 10 discusses porting the push functionality of MIDP.

Chapter 11 describes porting the audio building block.

Chapter 12 describes how to build MIDP.

Appendix A lists the targets in the build environment.

Appendix B lists the configuration options in the build environment.

Appendix C is the reference documentation generated by the Javadoc ™ tool for the `com.sun.midp.ssl` package.

Using Operating System Commands

This document may not contain information on basic UNIX® or Microsoft Windows commands and procedures such as opening a terminal window, changing directories, and setting environment variables. See the software documentation that you received with your system for this information.

Typographic Conventions

Typeface	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `% You have mail.`

`AaBbCc123`	What you type, when contrasted with on-screen computer output	`%` **`su`**
		`Password:`

AaBbCc123	Book titles, new words or terms, words to be emphasized	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be superuser to do this.
	Command-line variable; replace with a real name or value	To delete a file, type `rm` *filename*.

Shell Prompts

Shell	Prompt
C shell	`%`
Microsoft Windows	*directory*>

Related Documentation

The following documentation is included with this release:

Application	Title
All	*Release Notes*
Installing	*Installing MIDP*
Running and managing security for emulator	*Using MIDP*
Porting the MIDP Reference Implementation	*Porting MIDP*
Creating and building MIDlets	*Creating MIDlet Suites*
Viewing reference documentation created by the Javadoc™ tool	*API Reference*
Looking at examples	*Example Overview*

In addition, you might find the following documentation helpful:

- *The Java™ Language Specification* (Java Series), Second Edition by James Gosling, Bill Joy, Guy Steele and Gilad Bracha. Addison-Wesley, 2000, `http://java.sun.com/docs/books/jls/index.html`.
- *KVM Porting Guide*, available as part of the CLDC download package.
- The Java Specification Request (JSR), *J2ME™ Connected, Limited Device Configuration*, [JSR-000030] at `http://jcp.org/jsr/detail/30.jsp`.
- The JSR, *Mobile Information Device Profile 2.0*, [JSR-000118] at `http://jcp.org/jsr/detail/118.jsp`.
- A full list of JSRs for the J2ME platform, available at `http://jcp.org/jsr/tech/j2me.jsp`.
- *KVM Debug Wire Protocol (KDWP) Specification*, Sun Microsystems, Inc., available as part of the CLDC download package.

Accessing Sun Documentation Online

The Java Developer Connection^{sm} web site enables you to access Java platform technical documentation on the Web:

`http://developer.java.sun.com/developer/infodocs/`

Sun Welcomes Your Comments

We are interested in improving our documentation and welcome your comments and suggestions. You can email your comments to us at:

`docs@java.sun.com`

CHAPTER 1

Introduction

The *Mobile Information Device Profile* (MIDP) is a set of APIs (a profile) for the Java™ 2 Platform, Micro Edition (J2ME™). It is for small, resource-constrained devices such as mobile phones and personal organizers.

This product, MIDP Reference Implementation 2.0 FCS, complies with the *MIDP 2.0 Specification* produced by the Mobile Information Device Profile 2.0 [JSR-000118] available at `http://jcp.org/jsr/detail/118.jsp`.

MIDP Reference Implementation 2.0 FCS runs a device emulator on the Microsoft Windows 2000 platform. It has also been run on the Solaris™ 8 Operating Environment (Solaris OE) and RedHat Linux 7.2, although these are not supported platforms.

MIDP applications are called *MIDlets*. One or more MIDlets packaged together for installation onto a device is called a *MIDlet suite*.

This chapter introduces MIDP porting. It contains the sections:

- Product Architecture
- Porting Requirements

1.1 Product Architecture

MIDP works with the Connected Limited Device Configuration (CLDC) of the J2ME platform. The following figure shows the J2ME architecture from the perspective of MIDP:

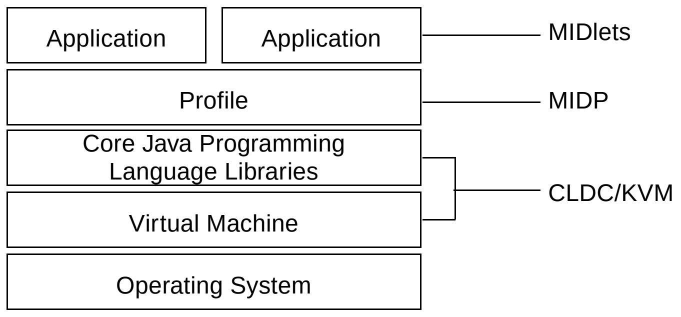

FIGURE 1 J2ME Architecture with MIDP and MIDP Applications

MIDP itself is made up of modules, many of which are implemented in two layers:

- Java programming language layer
  (This part typically can run without changes on a new device.)
- Native layer, implemented in the C programming language
  (This part typically needs to be changed so that it can run on a new device.)

The following figure shows the modules that comprise MIDP Reference Implementation. A module shown atop and overlapping another module uses code from that other module. (In some cases, the upper module might use only the native layer of another module.)

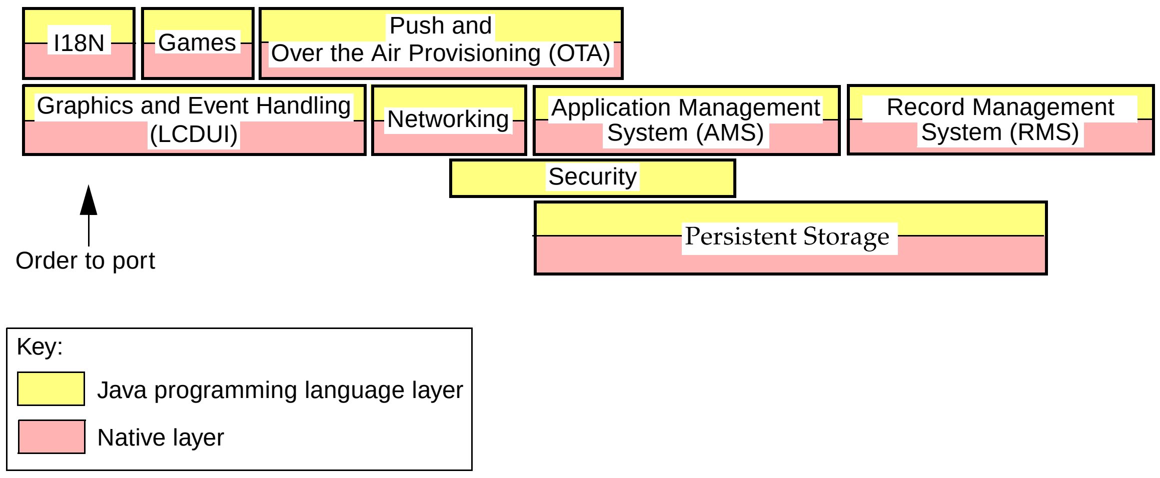
I18N
Games
Push and Over the Air Provisioning (OTA)
Graphics and Event Handling (LCDUI)
Networking
Application Management System (AMS)
Record Management System (RMS)
Security
Persistent Storage
Order to port
Key:
Java programming language layer
Native layer

FIGURE 2 MIDP 2.0 Modules

The chapters in this guide are organized in a way similar to the modules shown in the previous figure: they start with a module at the bottom of the stack, then work upwards. If you work in this order, you will always be ready to test a ported module because the layers underneath it will ready.

The modules are:

- **Event Handling** — Receives and processes events from the virtual machine and from MIDP. Events include key presses, requests for screen changes, and so on.

  See Chapter 3, "The Event Model" for more information.

- **Record Management System (RMS)** — Provides a platform-independent interface to MIDlets that need to store data on the device.

  See Chapter 4, "Persistent Storage" for detailed porting information.

- **Graphics** — Provides high-level graphical components, such as text boxes and forms, and a low-level component that includes graphics primitives such as lines and arcs.

  See Chapter 5, "Thread-Safety in LCDUI" and Chapter 6, "LCDUI Graphical User Interface," for detailed porting information.

- **Security** — Provides a policy that protects security-sensitive APIs by requiring permission to use them. It sets up a notion of trusted and untrusted MIDlet suites; these MIDlets typically have different sets of permissions.

  See Chapter 7, "Security" for more information.

- **Networking** — Provides the following communication protocols:
  - HTTP
  - HTTPS
  - Sockets
  - Server sockets
  - Datagram
  - Comm

  See Chapter 8, "Networking" for more information.

- **Application Management System (AMS)** — Installs, updates, and lists MIDlets on the device, removes MIDlets from the device, and manages the life cycles of the MIDlets on the device.

  See Chapter 9, "The Application Management System" for detailed porting information.

- **Over the Air Provisioning (OTA)** — Enables users to discover MIDlet suites on a network, download them using HTTP 1.1 or another protocol that implements the HTTP 1.1 functionality, and install or update them on their device.

  See Chapter 9, "The Application Management System" for detailed porting information.

- **Push** — Enables a MIDlet to be launched in order to accept data from a server at the server's request. When server data arrives, the user can accept or reject it. If the user accepts, the AMS exits any currently running MIDlet and launches the MIDlet that will receive the data.

  See Chapter 10, "Push Functionality" for more information.
- **Games** — Extends the low-level graphics component to provide game-specific capabilities such as an off-screen graphics buffer and the ability to query key status. This functionality can be the basis for the user interface of an action-style game.

  See Chapter 6, "LCDUI Graphical User Interface" for detailed porting information.
- **I18N** — Contains the input method handlers and resource message strings that provide a localized LCDUI interface. This porting guide does not cover I18N.

In addition, the MIDP Reference Implementation contains an optional module, the audio building block. The audio building block makes it possible to play single tones, tone sequences, and, optionally, synthetic audio such as WAV files. See Chapter 11, "Porting the Audio Building Block" for detailed porting information.

1.2 Porting Requirements

MIDP Reference Implementation 2.0 FCS can be ported to platforms that both meet the device requirements in the *MIDP 2.0 Specification* and have an ANSI C compiler. When you have completed your port, it must meet certain requirements to be compliant with the MIDP 2.0 specification. This section discusses these porting requirements in the topics:

- Hardware Requirements
- Software Requirements
- Compiler Requirements
- Requirements of the Completed Port

1.2.1 Hardware Requirements

The minimum hardware requirements for a device are:

- Display –
  - Screen-size: 96x54 pixels
  - Display depth: 1-bit
  - Pixel shape (aspect ratio): approximately 1:1

- One or more of the following user-input mechanisms –
  - One-handed keyboard
  - Two-handed keyboard
  - Touch screen
- Memory –
  - 256 kilobytes of non-volatile memory for the MIDP implementation, beyond what's required for CLDC.
  - 8 kilobytes of non-volatile memory for application-created persistent data
  - 128 kilobytes of volatile memory for the Java runtime
- Networking – Two-way, wireless, possibly intermittent, with limited bandwidth
- Sound – The ability to play tones, either with dedicated hardware or a software algorithm.

1.2.2 Software Requirements

The software requirements for a device are:

- A minimal kernel to manage the underlying hardware. It must:
  - Handle interrupts, exceptions, and minimal scheduling.
  - Provide at least one schedulable entity to run the virtual machine for the Java platform (JVM™).

  The kernel does not need to support separate address spaces (or processes) or make any guarantees about either real-time scheduling or latency behavior.
- A mechanism to read and write from non-volatile memory.
- Read and write access to the device's wireless networking.
- A mechanism to access the current date and the current time in milliseconds.
- Ability to write to a bit-mapped graphics display.
- A way to capture user input from one (or more) of the three user-input mechanisms previously listed.
- A way to manage the application life-cycle of the device.

Examples of MIDs include, but are not restricted to, cellular phones, two-way pagers, and wireless-enabled personal digital assistants (PDAs).

1.2.3 Compiler Requirements

The compiler must give users a way to indicate additional directories to be searched for included files. It must also define the basic types shown in the table below.

TABLE 1 Required Types in the C Programming Language

Type	Size	Semantics
`char`	8 bits	Signed or unsigned
`signed char`	8 bits	Signed
`unsigned char`	8 bits	Unsigned
`short`	16 bits	Signed
`unsigned short`	16 bits	Unsigned
`int`	32 bits	Signed
`unsigned int`	32 bits	Unsigned
`long`	32 bits	Signed
`unsigned long`	32 bits	Unsigned
`void *`	32 bits	Pointer

The compiler should, but is not required to, support 64-bit integers. (64-bit integer arithmetic is the only non-ANSI characteristic of the source code.)

MIDP Reference Implementation has been compiled with the following compilers:

- Sun Microsystems Inc. C Compiler 5.0 and 5.2 on the Solaris OE
- GNU C 2.95.2 compiler on the Solaris OE and Windows 2000, as well as 2.95.3 and 3.03 on Linux. For more information, see `http://www.gnu.org`.
- CygWin project version 1.3.1 with gcc. For more information, see `http://sources.redhat.com/cygwin`.
- Microsoft Visual C++ 6.0 Professional on Windows 2000.

The MIDP Reference Implementation has been tested on only machines that have 32-bit pointers and `int`s, and do not require far pointers.

When you port the MIDP Reference Implementation 2.0 FCS to a new device, you must ensure that there is a CLDC/KVM 1.0.4 implementation available for it. See the *KVM Porting Guide* for KVM porting instructions if an implementation is not already available.

1.2.4 Requirements of the Completed Port

For a port to be certified as compliant with the *MIDP 2.0 Specification*, it must pass the tests in the MIDP Reference Implementation Technology Compatibility Kit (TCK). For information on the MIDP TCK, see `http://java.sun.com/products/midp/`.

The *MIDP 2.0 Specification* lists these requirements, which are among those embodied in the TCK:

- MUST support MIDP 1.0 and MIDP 2.0 MIDlets and MIDlet Suites.
- MUST include all packages, classes, and interfaces described in this specification.
- MUST implement the OTA User Initiated Provisioning specification.
- MAY incorporate zero or more supported protocols for push.
- MUST give the user a visual indication of network usage generated when using the mechanisms indicated in this specification.
- MAY provide support for accessing any available serial ports on their devices through the `CommConnection` interface.
- MUST provide support for accessing HTTP 1.1 servers and services either directly, or by using gateway services such as provided by WAP or i-mode.
- MUST provide support for secure HTTP connections either directly, or by using gateway services such as provided by WAP or i-mode.
- SHOULD provide support for datagram connections.
- SHOULD provide support for server socket stream connections
- SHOULD provide support for socket stream connections.
- SHOULD provide support for secure socket stream connections.
- MUST support PNG image transparency.
- MAY include support for additional image formats.
- MUST support Tone Generation in the media package.
- MUST implement a tone control and SHOULD implement a volume control for tone generation. (This is a requirement of the Multi Media API [JSR 000135]. The MIDP 2.0 media support is a direct subset of the *Multi Media API Specification*. See `http://jcp.org/jsr/detail/135.jsp` for more information.)
- MUST support 8-bit, 8 KHz, mono linear PCM `wav` format IF any sampled sound support is provided.
- SHOULD implement a volume control if sampled sound support is provided. (This is a requirement of the Multi Media API [JSR 000135]. The MIDP 2.0 media support is a direct subset of the *Multi Media API Specification*. See `http://jcp.org/jsr/detail/135.jsp` for more information.)
- MAY include support for additional sampled sound formats.
- MUST support Scalable Polyphony MIDI (SP-MIDI) and SP-MIDI Device 5-to-24 Note Profile IF any synthetic sound support is provided.

- MAY include support for additional MIDI formats.
- MUST implement the mechanisms needed to support “Untrusted MIDlet Suites”.
- MUST implement “Trusted MIDlet Suite Security” unless the device security policy does not permit or support trusted applications.
- MUST implement “Trusted MIDlet Suites Using X.509 PKI” to recognize signed MIDlet suites as trusted unless PKI is not used by the device for signing applications.
- MUST implement “MIDP x.509 Certificate Profile” for certificate handling of HTTPS and secure connections.
- MUST enforce the same security requirements for I/O access from the Media API as from the Generic Connection framework, as specified in the package documentation for javax.microedition.io.
- MUST support at least the UTF-8 character encoding for APIs that allow the application to define character encodings.
- MAY support other character encodings.
- SHOULD NOT allow copies to be made of any MIDlet suite unless the device implements a copy protection mechanism.

Definitions for the terms MAY, MUST, and SHOULD are defined in RFC 2119 at `http://www.ietf.org`.

CHAPTER 2

Beginning a MIDP Port

This chapter describes the first steps in porting MIDP. By the end of the chapter, you will have a quick port of the MIDP Reference Implementation that compiles and should run the Hello MIDlet, but may not be fully functional. You will need to do additional work to complete the port and to tailor it for your device. The chapters after this one discuss those possibilities.

This chapter contains the sections:

- Preparing to Port MIDP
- Porting Device-Specific Configuration Code
- Trying It Out

This chapter assumes familiarity with the MIDP overview described in the previous chapter and the *MIDP 2.0 Specification*. The MIDP specification is at `http://jcp.org/jsr/detail/118.jsp`.

2.1 Preparing to Port MIDP

Do the following tasks before beginning a MIDP port:

1. **Create source and build subdirectories for your device in the MIDP tree.**

   The directories to create are:

   - *midpInstallDir*`\build\`*newPlatform*
   - *midpInstallDir*`\src\`*newPlatform*

   where *newPlatform* is the name for the directory that hold the files for your device. For example, if *midpInstallDir* was `c:`\midp2.0fcs and your device was `deviceX`, you would execute the commands:

```
c:\midp2.0fcs\build> mkdir deviceX
c:\midp2.0fcs\build> cd ..\src
c:\midp2.0fcs\src> mkdir deviceX
c:\midp2.0fcs\src> cd deviceX
c:\midp2.0fcs\src\deviceX> mkdir native
```

2. **Copy the code from the existing** *midpInstallDir*`\src` **subdirectory that most closely matches your system into your** *midpInstallDir*`\build\`*newPlatform* **directory.**

   For example, you could copy the files from *midpInstallDir*`\src\solaris`:

   ```
   c:\midp2.0fcs\src> cp -r solaris/* deviceX
   ```

3. **Copy the makefiles for an existing architecture into your** *midpInstallDir*`\build\`*newPlatform* **directory.**

   Copy:

   - `GNUmakefile`
   - `platform.gmk`
   - The `makefiles` directory and its contents

   For example, you could copy the files from *midpInstallDir*`\build\solaris`:

   ```
   c:\midp2.0fcs\build> cp solaris/GNUmakefile deviceX
   c:\midp2.0fcs\build> cp solaris/platform.gmk deviceX
   c:\midp2.0fcs\build> cp -r solaris/makefiles deviceX
   ```

4. **Set configuration options in your platform's makefiles.**

   See Appendix B, "Configuration Options" configuration options that you might need to set for porting. Typical options that need updating are:

   - Compiler flags
   - `GCC`
   - `PLATFORM`
   - `MIDP_EXCLUDE_CLASSES`
   - `MIDP_INCLUDE_SRC`
   - `_BOOTDIR`
   - Configurable build parameters, such as `DEBUG`

   For example, to make the value of the `PLATFORM` option `deviceX`, you would update the `PLATFORM` definition in the `c:\midp2.0fcs\build\deviceX\platform.gmk` file to:

   ```
   PLATFORM = deviceX
   ```

   If you have already ported CLDC, use those makefiles as a guide.

5. **Set targets in your platform's makefiles.**

   The makefile targets have the MIDP executable run inside the device emulator with debugging symbols, without debugging symbols, with profiling code, and so on. See Appendix A, "Build Targets" for a list of existing targets. You might want to add targets that include or exclude device-specific features, such as optional communication protocols.

2.2 Porting Device-Specific Configuration Code

To start porting MIDP, do the following tasks:

1. **Implement the heap allocation macros.**

   There are five allocation macros for managing the memory of the native heap:

   - `midpMalloc`
   - `midpCalloc`
   - `midpRealloc`
   - `midpStrdup`
   - `midpFree`

   The macros are defined in *midpInstallDir*`\src\share\native\midpMalloc.h`. Change the definitions of these macros, if necessary, to work with your device.

   Note – Always use these macros to manage memory from the heap in any native code you write.

   ---

2. **Update native calls.**

   The files with native calls are in the following directories:

   - *midpInstallDir*`\src\share\native`
   - *midpInstallDir*`\src\`*newPlatform*`\native`

   For example, if your system does not support a native call, substitute in a call that has the same functionality, or create a stub for it. To create a stub, comment out or remove the function body that contains the call, and substitute a print statement. One way to find these calls is by trying to build MIDP to see which calls prompt errors. (For instructions on building MIDP, see *Using MIDP*.) The errors could look something like this:

   ```
   /tmp/ccTk6mVS.o(.text+0x7): undefined reference to `LCDUIdrawLine'
   ```

   The stub that you create for the call could look something like this:

   ```
   LCDUIdrawLine(...){ printf("in LCDUIdrawLine stub\n"); }
   ```

   This step enables you to quickly get your port compiling. As you go through the chapters that follow, you will fill in the stubs as you port each module.

3. **Update the destination of debugging output, if necessary.**

   Ensure that the `printf` function sends its output to a location from which it is easy for you to collect information. For example, one such destination might be the debug port of your emulator. (If you have already ported CLDC, this step should already be done.)

2.3 Trying It Out

Try out your port at this point:

1. **Build a ROMized version of MIDP.**

   In the MIDP Reference Implementation, the `midp` target in the build environment builds a ROMized version of MIDP. If you have not changed this aspect of the makefiles, then you would be able to use a command such as this:

   ```
   c:\midp2.0beta\build\deviceX> make midp
   ```

   For instructions on building MIDP, see Chapter 12, "Building Your Port."

2. **Create the following simple version of the Hello MIDlet to run on the device.**

   The following version of the HelloMIDlet does not use any graphics. It prints simple messages to the serial port of the device:

   ```
   import javax.microedition.midlet.*;

   public class HelloWorld extends MIDlet {
       public HelloWorld() {
           system.out.println("In HelloWorld constructor");
       }

       public void startApp() {
           System.out.println("In HelloWorld startapp");
       }

       public void pauseApp() {
       }

       public void destroyApp(boolean unconditional) {
       }

   }
   ```

3. **Build and package the MIDlet as described in** *Creating MIDlet Suites*.

4. **Run the MIDlet as described in** *Using MIDP*.

CHAPTER 3

The Event Model

This chapter describes the event-handling mechanism of the MIDP Reference Implementation. The mechanism is implemented primarily in the `com.sun.midp.lcdui.DefaultEventHandler` class. Because it is written in the Java™ programming language, it will run without changes on a new device. event-handling mechanism also has a few native calls that deliver events to MIDP that you might have to port. This chapter explains how the event model works so that you can better understand the reference implementation, and briefly discusses the C function that you might have to port.

This chapter contains the sections:

3.1 Event Model Overview

MIDP is one software layer in a Java 2 Platform, Micro Edition (J2ME™) implementation; multiple layers can generate events that MIDP must handle. MIDP's event handling mechanism enables it to receive and process events that come from two sources:

- Outside MIDP – Events, such as key presses, that originate in the device and propagate through the underlying layer to MIDP
- Inside MIDP – Events, such as screen changes, that originate in MIDP and have meaning only to it

MIDP processes all events, regardless of their source, with a single thread: the *event thread*. In addition to the event thread, MIDP also has a single queue, the *event queue*, that holds all pending events, regardless of their source. The event thread serially processes events from the event queue until the queue is empty.

When the event queue is empty, the event thread goes to sleep by calling the `wait` method. When the event queue receives an event, it calls the `notify` method to wake up the event thread so that the event can be processed. The wait/notify mechanism of the Java programming language is an efficient way to process events, in terms of battery life. If MIDP used a mechanism that had threads busy-wait, the device's battery could be drained more quickly.

Having a a single event thread operating on a single queue simplifies event handling, although putting the events in the queue introduces some complexity because the events from the inside and outside MIDP are received differently. The following sections discuss how events from the two sources are received and processed.

3.2 Events From the Virtual Machine

Passing an event from the virtual machine (VM) to MIDP is not straightforward because the VM typically is written and operates in a different computer language than the MIDP layer. The VM is often in C or C++, and MIDP is largely written in the Java programming language.

MIDP solves the problem of getting events from the VM by implementing a special data stream between the VM and MIDP. The VM writes a series of integer values into the stream for each event that occurs, then MIDP reads the values from the stream to process the events. MIDP can receive the following predefined event types from the VM: key, pointer, command, and multimedia.

MIDP has two threads that work together to get an event from the stream. One thread, the *VM-event thread*, gets the initial integer of the series that describes an event. The event thread (the thread that handles the events on the event queue) gets the event’s remaining integers. Both threads use blocking reads. (That is, the `read` method returns only when there is data available on the stream; if the stream is empty, the thread cannot do other work.)

In more detail, getting and processing an event from the VM happens like this:

1. The VM-event thread attempts to read an integer from the stream. When there is data on the stream, the read method unblocks and returns a single integer value. The integer identifies the type of event in the stream.
2. The VM-event thread places the event identifier into the event queue and waits by calling the `wait` method. The VM-event thread waits because it does not know how many of the subsequent integers on the stream are part of this event. It cannot yet locate the next event identifier.

3. The event thread processes the event on the event queue:

   a. It reads the integers that are on the stream for the event it is processing. The event thread has access to the number of integers associated with each type of event, so it is able to read only the data for the current event. This read will not block because it is guaranteed that a specific number of integers are on the stream for each event ID.

   b. It uses the integers to process the event.

   c. It calls the `notify` method to allow the VM-event thread to read the next event identifier from the stream. (The read operation in Step a took enough data from the stream that the next integer on the stream will be an event identifier.)

   d. The VM-event thread awakens and the process repeats from Step 1.

There is a potential problem with the data stream: if a user caused events to happen at a rate that surpassed MIDP's ability to process them, the stream could overflow and incoming events would be ignored. In practice, it has not been an issue.

3.3 Events From MIDP

Getting events from MIDP onto the event queue is more straightforward: they are placed into the event queue by the MIDP system thread or MIDlet-created thread that is executing the call that generated the event. Unlike the VM-event thread, this thread does not wait for the event to be processed; it puts the event on the event queue and returns immediately. MIDP can receive the following predefined event types from MIDP application code: repaints, screen changes, serialized callbacks, content invalidations (such as a `Form`), and item state change notifications.

The event queue, from which the event thread serially processes elements, has special processing to keep MIDP’s events from causing an overflow. (The chances of overflow would otherwise be greater for application-generated events, since an application might have a very tight loop that generate events.) The event queue does the following special processing:

- It coalesces repaint events – If a repaint for region R1 has been scheduled but not yet processed, and a thread attempts to schedule a subsequent repaint for region R2, the end result is a single repaint event with a region that is the union of R1 and R2.
- It discards any prior, unprocessed screen-change event when a new screen-change event arrives – The event queue keeps only the most recent screen-change event for processing.

Both actions are explicitly permitted by the MIDP specification. In addition to keeping the event queue from overflowing, the special processing keeps the memory footprint of the event queue at or below a small maximum limit (basically the sum of one pending event of each type).

3.4 Native Functions

The virtual machine (VM) has its own event handling mechanism, from which MIDP gets events. Part of the virtual machine's event handling is the function `GetAndStoreNextKVMEvent`. Systems built on the VM, such as MIDP, must implement this method. (The Porting Guide that came with CLDC for more detailed information.)

The MIDP Reference Implementation implements the `GetAndStoreNextKVMEvent` function in the `nativeGUI.c` file, which is in the platform-specific directories *midpInstallDir*`\src\win32\native` and *midpInstallDir*`/src/solaris/native`. Because the method is device-specific, you will have to port it to your device. The following paragraphs describe its actions.

Because the MIDP Reference Implementation defines a number of events, one task that `GetAndStoreNextKVMEvent` performs is to determine whether an event should be passed to the Java platform event handler. The MIDP Reference Implementation typically passes key presses, button presses, mouse movement, mouse clicks (on the emulator screen and the keypad), multimedia, application shutdown, and timers. Events that are not passed to the Java platform event handler are those that occur when the system menu is displayed (the system menu is a native component controlled entirely by native code). In this case, key presses go directly to the native system menu event handler, not to the Java platform event handler.

If an event will be passed to the Java platform event handler, `GetAndStoreNextKVMEvent` calls the `StoreMIDPEvent` function. The function encodes the event and makes it available to the Java platform. The `StoreMIDPEvent` function is in the file *midpInstallDir*`\src\share\native\kvm\midpEvent.c`.

MIDP gains access to the event using the special data stream described in Section 3.2 "Events From the Virtual Machine" on page 14. The native methods to access the special data stream are defined in *midpInstallDir*`\src\share\native\kvm\midpEvents.c` and are called by the class `com.sun.midp.lcdui.Events`.

CHAPTER 4

Persistent Storage

This chapter discusses how to port and customize the persistent storage functionality. Persistent storage enables data to be kept on a device both across uses of a MIDlet suite and across shut downs and restarts of the device. The functionality is used by the Record Management System (RMS), by the application management system (AMS), by the native code that manages system configuration, by the security module, and can also be used by MIDlet suites.

The persistent storage implementation provides a *flat file system*, which is a file system that puts its files in a single directory. Because there are no subdirectories, each file must have a unique name.

The persistent storage implementation has both a native layer and a Java™ programming language layer (Java layer). The files in the native layer are `RandomAccessStream.c`, `storageFile.c`, `storage.h`, and `storage.c` in the `src/share/native/` directory. The classes in the Java layer are `RandomAccessStream` and `File` in the `com.sun.midp.io.j2me.storage` package.

To port persistent storage, you must make the native layer run on your device. To customize persistent storage, you also modify the Java layer.

This chapter contains the sections:

- Porting the Native Layer
- Customizing the Java Layer

4.1 Porting the Native Layer

The native layer of the persistent storage implementation is in the `src/share/native/` directory, in the files `storage.h`, `storage.c` and `storageFile.c`:

- `storage.h` — defines the macros and declares the functions for the persistent storage implementation. You should not need to change this file.
- `storage.c` — defines the functions that interact with the persistent storage of the device, including opening, reading, and writing files. This file requires porting.
- `storageFile.c` — defines the functions that mediate between the Java and native layers. The functions in this file use KNI; you should not need to change them. (For more information on KNI, see your CLDC documentation.)

The way that you port `storage.c` depends on the capabilities of your device. If your device has a file system interface that supports directory hierarchies, you could customize the existing code. To do this, start with the code between these lines:

```
#ifdef UNIX
#endif /* UNIX */
```

It uses the following system APIs for file manipulation:

- `open`
- `close`
- `read`
- `write`
- `stat`
- `lseek`
- `ftruncate`
- `fstat`

  Note that `fstat` is used only to get file size; if your device has another call that returns a file's size, that call is sufficient.
- `unlink`
- `rename`

It also includes the following directory operations:

- `mkdir`
- `opendir`
- `readdir`
- `closedir`

If your platform has another type of interface, such as record-oriented file system or a flat-file interface, reimplement `storage.c` to use it. (In these cases, you might also change the Java layer.)

4.2 Customizing the Java Layer

The Java layer runs without changes on a new device. You might want to customize it if your device has an alternate storage mechanism. For example, if the device has a record-based storage system (such as a database), the RMS APIs might map directly or nearly directly to native methods. Using native methods (instead of the RI's Java layer) would not only give MIDlets RMS performance that is similar to a native application, you could probably also remove much of the record handling code.

Note, though, that not all devices provide fast native storage operations. If your device does not, you should try to minimize the memory access operations done by your port by using caching.

Another example of an alternate storage mechanism is a device that has a full-featured file system. For this type of device, you might decide to map record stores to directories and records to files. (In the RI, a record store is a file and the records within them are blocks of data in the file.) This would allow the file system to handle most of the metadata maintained in each record store now (such as the modification time, version information, and so on) and therefore to remove much of the code in the `RecordStore` class.

As you customize the RMS, pay attention to any system storage limitations on your device. RMS gives record stores names that are case sensitive and can be up to 32 Unicode characters in length. If your device is case insensitive, or requires shorter names, you will have to use a different mechanism (such as a map table) to map a MIDlet suite's RMS record store name to a name on your device.

CHAPTER 5

Thread-Safety in LCDUI

This chapter describes the coding conventions that ensure thread-safety for *LCDUI*, the user interface of the MIDP Reference Implementation. The LCDUI classes are in the `javax.microedition.lcdui` package. Thread-safety means that access to shared data is *safe* (data never becomes corrupted, even in the presence of concurrent access) and that the system is *live* (threads do not deadlock). It is best to understand this chapter before beginning a port of the graphical user interface.

This chapter contains the sections:

5.1 Requirements

All ports of the MIDP Reference Implementation must obey the following thread-safety requirements:

1. All calls into LCDUI, from any thread, from any class outside LCDUI, must leave LCDUI in a valid and self-consistent state.

2. The *event delivery* calls made by LCDUI into the application must be serialized, as required by the *MIDP 2.0 Specification*. This set of calls is defined in class documentation for `javax.microedition.lcdui.Canvas`. (See Chapter 3, "The Event Model" for more information on the events delivered by LCDUI.)

3. Applications must be allowed to define and implement their own locking policy for their data structures, without any restrictions imposed by LCDUI, and be assured that they will run correctly (that is, deadlock-free and safe) when they interact with LCDUI. The application's locking policy should be free to use locks even on LCDUI objects. Applications should be able to create subclasses of LCDUI classes without violating any safety guarantees.

5.2 Design Approach

The MIDP Reference Implementation treats all LCDUI data (static variables of all classes and instance variables of all objects) as if it were a single object. It is *shared data* because this data is accessible to multiple threads. A single, global lock object `LCDUILock` is defined to protect concurrent access to all shared data.

Because all LCDUI data is treated as a single, shared object, there should not be multiple threads running simultaneously inside of LCDUI. To serialize, the general approach of the MIDP Reference Implementation is to apply locking around the perimeter of LCDUI. All methods that are called from the outside are responsible for ensuring that locking is done properly before making calls into other parts of LCDUI. Method calls into LCDUI enter through the following general routes:

- Application calls into the public API
- Events being delivered from the KVM
- MIDlet state changes from the scheduler

In general, methods internal to LCDUI need not be concerned with locking, and they may assume that their callers have handled locking correctly. In certain cases, internal methods are called without holding the lock, because of complications with the locking protocols. These cases are marked in the source code with a `SYNC NOTE` comment.

This design implies that LCDUI internal code should not call the same methods that are called from outside, otherwise lock nesting will result. While lock nesting is not a fatal problem, if left uncontrolled it can lead to performance problems. Care has been taken to refactor the code so as to avoid lock nesting. This refactoring has led to an idiom where a method intended to be called from outside (such as a public API method) simply takes the lock and then calls an internal method. Internal LCDUI code calls the internal method directly. By convention, these internal methods are named after their public counterparts, with `Impl` appended.

An exception to the global lock rule is for the `Graphics` object, where the API is structured so that painting to the screen is implicitly serialized. `Graphics` objects for offscreen `Image` objects do not make any access to LCDUI data, and so they use their own locking convention. See "Graphics Conventions" on page 29 for a description of this convention.

In order to preserve event serialization semantics (Requirement 2), another lock `calloutLock` is used when making calls subject to this requirement. In many cases it is unclear whether a particular method call will stay within LCDUI code or will find its way into application code. This is because much of the LCDUI implementation uses the same APIs that are exposed to applications. Thus, the system will hold `calloutLock` while calling any method that *might* end up in application code. For cases where the thread stays within LCDUI, taking `calloutLock` is unnecessary. It might be possible to optimize the system by avoiding taking the `calloutLock` in such cases, but it's likely that the cost of testing for this case will largely offset the savings from avoiding the lock.

Deadlock is prevented by establishing a total ordering of all locks in the system. The ordering of locks is as follows, from highest to lowest priority:

1. The `calloutLock` object
2. All application-defined locks
3. The `LCDUILock` object

The general rule is that code holding a lock must not acquire any lock that has a higher priority. However, code holding a lock may acquire any lock with lower priority. From LCDUI's point of view, all application locks have the same priority. LCDUI code must assume that any call into the application will take application locks, and any call coming from the application holds application locks.

This ordering is a consequence of the thread-safety requirements. Application code that is called by LCDUI must be allowed to take any lock, and application code calling into LCDUI must be allowed to hold any lock (Requirement 3). Since holding `LCDUILock` is required for access to LCDUI data in methods called from the application (Requirement 1), `LCDUILock` must have a lower priority than all application locks. Since `calloutLock` must be held in order to serialize calls into the application (Requirement 2), `calloutLock` must have a higher priority than all application locks.

5.3 Coding Conventions

This section covers thread-safety coding conventions for LCDUI. It has the topics:

- General Conventions
- Public Method Conventions
- Constructor Conventions
- Event-Handling Method Conventions
- Application Callout Conventions
- Graphics Conventions

5.3.1 General Conventions

The general thread-safety conventions for LCDUI are as follows:

- Any method that is called from outside LCDUI, such as public API methods, must take `LCDUILock` while making any access to LCDUI shared data (class or instance variables)
- Internal LCDUI methods may assume that LCDUILock is held when they are called, and they may thus read or write shared data without acquiring any locks
- Code that is called as if it were an application, even internal code, must take `LCDUILock` prior to manipulating any LCDUI data
- Constructors are generally left unlocked, unless they manipulate shared data, in which case they must hold `LCDUILock`
- Code that holds `calloutLock` may take `LCDUILock`
- Code that holds `LCDUILock` must release it before taking `calloutLock`
- Code that calls into the application must hold `calloutLock` around this call
- Code called from the application must not take `calloutLock`
- Code that holds `LCDUILock` must not call into the application
- Exceptions to these rules are documented with a `SYNC NOTE` comment

5.3.2 Public Method Conventions

Within each call in the public API, there is an appropriate synchronized block. For example, in `Gauge.java`:

```
public void setMaxValue(int maxValue) {
    synchronized (Display.LCDUILock) {
        setMaxValueImpl(maxValue);
    }
}
```

Methods that require the `Display.LCDUILock` cannot merely use the `synchronized` method modifier because this would indicate that synchronization is to be performed on the object itself, not on the `LCDUILock` object. The rule is that `LCDUILock` must be held whenever there is access to any shared data, that is, data that is potentially accessible to multiple threads. This includes class and instance variables belonging to this class or to any other LCDUI class. Local variables and method parameters are private to each thread and are not shared data. Access to them does not require locking.

Note that the synchronization block begins after argument checking. This is safe, because the argument checking does not involve any access to shared data. This is a tiny optimization. It would also have been correct to have the synchronization block around the entire body of the method.

Certain simple methods may not need synchronization at all. These cases are documented with a explanatory SYNC NOTE comment. For example, in `Gauge.java`:

```
public int getValue() {
    // SYNC NOTE: return of atomic value, no locking necessary
    return value;
}
```

In this case, the value being returned is an `int`, which the *Java™ Language Specification* requires to be manipulated atomically. (That is, a simultaneous reads and writes to this value will never result in a mixture of old and new bits. If the value were a `long`, this property wouldn't hold.) The value returned will either be the old value or the new value; which it will be is unpredictable. Adding locking doesn't change the situation, and so locking is omitted from these cases for purposes of efficiency. There are a handful of "getter" methods such as this that can avoid synchronization.

Strictly speaking, this code should be synchronized because of memory ordering issues. In practice, memory ordering issues arise only on multiprocessor systems, and it is very unlikely that this code will be executed on such systems. The MIDP Reference Implementation, therefore, made a design decision to avoid locking overhead in cases such as these by relying on the virtual machine (VM) to provide sequentially consistent memory semantics.

Locking must occur around *assignment* of atomic values. Consider the following hypothetical example:

```
public void setIndex(int newIndex) {
    synchronized (lock) {
        index = newIndex; // index is an instance variable
    }
}
```

One might be tempted to omit this locking, because the assignment is atomic, and locking would not seem to add anything. However, holding the lock is necessary to ensure that if another thread reads this variable several times while holding the lock, these reads will return consistent values.

5.3.3 Constructor Conventions

Constructors typically do not need any synchronization if all they do is initialize the newly created object. However, individual cases will need to be inspected carefully in order to determine whether they might affect other data structures. For example, in Form.java:

```
public Form(...) {
    super(title);                        // (a)

    synchronized (Display.LCDUILock) {    // (b)
        // long, complex initialization
    }
}
```

At `(a)`, the call to `super` ends up creating a new `Layout` object. Without tracing the entire call tree, it is difficult to determine whether the call to the superclass constructor has any access to shared data. It is therefore reasonable to want to put locking around this call. Unfortunately, a constructor's call to `super` must appear as its very first statement and cannot appear within a `synchronized` block. Therefore, the superclass constructor must be inspected carefully to ensure that there is no unsynchronized access to shared data.

The synchronization block beginning at `(b)` surrounds a complex initialization routine. Once again, without inspecting the entire call tree, it is difficult to tell whether this routine makes any access to shared data. Thus code like this has been placed within a synchronization block so that access to shared data is safe. Even if the code in its current state makes no access to shared data, a future modification to the code might add such an access. Leaving this code unlocked is therefore quite fragile.

Even if the body of this constructor and the body of the superclass constructor are synchronized, there is still a potential safety problem. Suppose the superclass constructor were to store a reference to the object being constructed into a shared data structure. There is a window of time between the release of the lock held by the superclass constructor and the reacquisition of the lock by the body of this constructor. During this window, other threads will have access to a reference to this object. If another thread were to call a method on this object, that method would be operating on the object in a partially constructed state, which might lead to misbehavior or errors. Therefore, constructors must be extremely careful not to store references to the newly constructed object into shared data, even if the store is done within a synchronization block.

5.3.4 Event-Handling Method Conventions

Event handling methods and MIDlet state change methods are considered to originate outside LCDUI and thus must also hold `LCDUILock` during access to shared data. The `EventHandler` thread makes calls on the `DisplayAccessor` object in order to deliver events to LCDUI. From the `DisplayAccessor` object, the call tree fans out to individual `Displayable` objects. Every method of `Displayable` that is called from within the `DisplayAccessor` must be inspected in order to determine whether it should be locked. In practice, this means that LCDUI classes that are subclasses of `Displayable` must add synchronization within their `showNotify`, `hideNotify`, `keyPressed`, `paint`, and other event delivery methods. Note that subclasses of the `Item` class have these methods but that they are not locked. This is because the `Form` object receives the event call, takes the lock, and then calls the corresponding method on the appropriate `Item` object.

5.3.5 Application Callout Conventions

In some cases, the processing of an event will end up calling out to the application. (These are callbacks from the point of view of the application, but this section takes the point of view of the system and calls them callouts.) The `LCDUILock` must not be held during any callout, because doing so may give rise to deadlock. However, if the current `Displayable` is not a `Canvas` object, then it will be a high-level UI component that is part of LCDUI. Each component will be responsible for reacquiring the `LCDUILock` for itself as described above.

In order to serialize callouts (Requirement 2), the code must hold `calloutLock` across any call that might end up in the application. These calls include the following:

- `Canvas.hideNotify`
- `Canvas.keyPressed`
- `Canvas.keyRepeated`
- `Canvas.keyReleased`
- `Canvas.paint`
- `Canvas.pointerDragged`
- `Canvas.pointerPressed`
- `Canvas.pointerReleased`
- `Canvas.showNotify`
- `Canvas.sizeChanged`
- `CommandListener.commandAction`
- `CustomItem.getMinContentHeight`
- `CustomItem.getMinContentWidth`
- `CustomItem.getPrefContentHeight`
- `CustomItem.getPrefContentWidth`
- `CustomItem.hideNotify`
- `CustomItem.keyPressed`
- `CustomItem.keyRepeated`
- `CustomItem.keyReleased`

- CustomItem.paint
- CustomItem.pointerDragged
- CustomItem.pointerPressed
- CustomItem.pointerReleased
- CustomItem.showNotify
- CustomItem.sizeChanged
- CustomItem.traverse
- CustomItem.traverseOut
- Displayable.sizeChanged
- ItemCommandListener.commandAction
- ItemStateListener.itemStateChanged
- Runnable.run resulting from a call to Display.callSerially

Because calloutLock has a higher priority than LCDUILock, the code must release LCDUILock prior to taking calloutLock and calling into the application. An example of this occurs in CustomItem.java where the application's ItemStateListener is called:

```
void callKeyPressed(int keyCode) {
    ItemCommandListener cl = null;
    Command defaultCmd = null;

    synchronized (Display.LCDUILock) {
        cl = commandListener;
        defaultCmd = defaultCommand;
    } // synchronized

    // SYNC NOTE: The call to the listener must occur outside
    // of the lock
    try {
        // SYNC NOTE: We lock on calloutLock around any calls
        // into application code
        synchronized (Display.calloutLock) {
            if ((cl != null) ... ) {
                cl.commandAction(defaultCmd, this);
            } else {
                this.keyPressed(keyCode);
            }
        }
    } catch (Throwable thr) {
        Display.handleThrowable(thr);
    }
}
```

(Some error checking has been omitted for brevity.)

This code uses the *open call* technique of loading information into local variables while the lock is held, and ensuring that unlocked code uses only these local variables. This is necessary in case the value of the itemStateListener instance variable or the items array is changed between the set and the actual call.

A consequence of this structure is that the call to the application must occur at the same level in the calling structure as the locking of `LCDUILock`, because `LCDUILock` must be released before `calloutLock` is taken and the call made into the application. This in turn implies that the information about the decision of whether to make the `itemStateChanged` call must be returned from the `Item` that made the decision. The call cannot be made directly from the code in the `Item` that handles the event.

5.3.6 Graphics Conventions

Since the calls to a `Canvas` object's `paint` method are serialized, the `Graphics` object passed to it need not be synchronized at all. This relies on the underlying graphics library to be thread-safe. It also assumes that native methods implicitly provide exclusion, as is the case in KVM.

However, a `Graphics` object whose destination is an `Image` will need to be synchronized, since multiple threads may attempt to use the `Graphics` object to paint simultaneously. The locking occurs on the `Graphics` object itself, not on `LCDUILock`, because the effect of any graphics call affects only the state of that `Graphics` object. This locking is applied in `ImageGraphics`, a private implementation subclass of `Graphics`, leaving the main `Graphics` class without locking:

```
class Graphics {
    public void clipRect(int x, int y, int width, int height) {
        ...
    }
    ...

class ImageGraphics extends Graphics {
    public void clipRect(int x, int y, int w, int h) {
        synchronized (this) {
            super.clipRect(x, y, w, h);
        }
    }
    ...
}
```

Locking is necessary for certain methods, such as `clipRect`, because it manipulates the `Graphics` object's state using code written in the Java programming language. However, the `drawLine` method is simply a direct call to a native method, which the MIDP Reference Implementation assumes runs single-threaded, and so no locking is necessary. The subclass can add no value by taking a lock around the native call, so it simply inherits the native implementation.

Note that it is not necessary to lock the destination image of any graphics call nor the source image of the `drawImage` call. The assumption is that the only operations that have access to actual image data are native, and so no exclusion is necessary.

Graphics operations from different threads might be interleaved, but locking within the `Graphics` class will not prevent that from occurring; that is the application's responsibility.

5.4 The `serviceRepaints` Method

The *MIDP 2.0 Specification* requires that, in general, callouts be serialized. That is, a callout may not begin until the previous callout has returned. (This rule is covered by Requirement 2.) However, the specification for `serviceRepaints` states that it must not return until the paint method is called and has returned. This is true even if `serviceRepaints` is called from within an application callout. This is the only case where two callouts are allowed to be in progress simultaneously.

This arrangement gives rise to the possibility of deadlock. Suppose that the event thread has taken `calloutLock` and is calling an application's event method, which attempts to acquire one of the application's locks. Suppose further that an application thread already holds this application lock and now calls `serviceRepaints` in order to force a pending repaint to be processed. The `serviceRepaints` code attempts to acquire `calloutLock` in order to call `paint`, and the system is now deadlocked.

The *MIDP 2.0 Specification* prevents deadlock in this case by imposing a special rule, which is that applications are prohibited from holding any of their own locks during a call to `serviceRepaints`. This rule is reflected in the lock ordering policy described in "Design Approach" on page 22. Since `serviceRepaints` may call `paint`, which requires holding `calloutLock`, the rule about holding no locks still applies. This is the only case where the lock ordering rules are exposed to the application. The application is allowed to hold its locks during a call to any other LCDUI method, in accordance with Requirement 3. The `serviceRepaints` method is the sole exception to this rule.

A multi-threaded application will generally need to hold locks on its own data structures in order to protect them from concurrent access. The problem is that the application is *prohibited* from holding these locks when it calls `serviceRepaints`. This makes `serviceRepaints` hard to use correctly. In practice, application code must use the open call technique in order to use `serviceRepaints` safely.

CHAPTER **6**

LCDUI Graphical User Interface

This chapter covers some issues in porting the graphical user-interface (GUI) portion of the MIDP Reference Implementation. The GUI portion of MIDP is implemented in the `javax.microedition.lcdui` package and the `javax.microedition.lcdui.games` package. The GUI portion of MIDP is often called *LCDUI*.

Most of LCDUI is written in the Java™ programming language, although some of the code is written in C. The native methods are written using portable code whenever possible, but some will require additional porting. Native methods that require additional porting have an `LCDUI` prefix. Their signatures are listed in *midpInstallDir*`/src/share/native/defaultLCDUI.h`.

This chapter discusses issues in porting both the `javax.microedition.lcdui` and the `javax.microedition.lcdui.games` packages. It contains the sections:

- Overview
- Issues in Porting Low Level API Functionality
- Using Native Widgets for MIDP Screens
- Porting the Game Package

6.1 Overview

The `javax.microedition.lcdui` package contains APIs for creating both *structured screens* and *unstructured screens*. (MIDP 2.0 also has a hybrid class, `CustomItem`, which is an unstructured item for a form.) The parent class for both structured and unstructured screens is `Displayable`. It contains the common capabilities, such as a title, a ticker (an instance of the `Ticker` class), and the ability to have associated abstract commands (instances of the `Command` class.) You will probably extend `Displayable` if you replace RI widgets with native ones. (See Section 6.3.2 "Example: Replacing Part of `DateField`" on page 43 for an example.)

Structured screens are portable but do not give the application access to low-level input mechanisms or control of the screen. Lists are an example of a structured screen. You create a structured screen with the LCDUI's *high-level* APIs. The high-level APIs are subclasses of the `Screen` class: `Alert`, `Form`, `List`, and `TextBox`. A form is a screen that contains items. These items are subclasses of the `Item` class: `ChoiceGroup`, `CustomItem`, `DateField`, `Gauge`, `ImageItem`, `Spacer`, `StringItem`, and `TextField`. An alert can have an associated `AlertType` so that its behavior can be customized for different purposes.

Unstructured screens provide access to low-level I/O, but can be less portable. A screen that shows the push puzzle game board is an example of an unstructured screen. You create unstructured screens with the LCDUI's *low-level* APIs. The low-level APIs are provided by the classes `Canvas`, `GameCanvas`, and `Graphics`.

The `javax.microedition.lcdui.games` package is series of classes for creating rich gaming content for wireless devices.

6.2 Issues in Porting Low Level API Functionality

This chapter discusses some issues in porting low-level graphics classes, such as `Canvas` and `Graphics`. It covers the topics:

- Drawing Graphics Primitives
- Porting PNG Transparency

6.2.1 Drawing Graphics Primitives

The `Graphics` class provides drawing primitives for text, images, lines, rectangles, and arcs. This section describe the graphics coordinate system, and how that affects the drawing of lines, rectangles, and filled rectangles. It also explains the requirements of drawing arcs. It contains the sections:

- Overview of the Coordinate System
- Drawing Lines
- Drawing and Filling Rectangles
- Drawing and Filling Arcs

6.2.1.1 Overview of the Coordinate System

A `Graphics` object has a coordinate system with its origin at the upper left-hand corner of the destination. The X-axis direction is positive towards the right, and the Y-axis direction is positive downwards. The coordinate system and graphics implementation in the MIDP Reference Implementation assume that the device has square pixels. If your device does not have square pixels, you will have to modify the graphics subsystem to take the different pixel shape into account. An application must be able to rely on equal horizontal and vertical distances being equal on the device display.

The coordinate system represents locations between pixels, not the pixels themselves. The first pixel in the upper left corner of the display lies in the square bounded by coordinates `(0,0)`, `(1,0)`, `(0,1)`, and `(1,1)`. This is shown in the following figure.

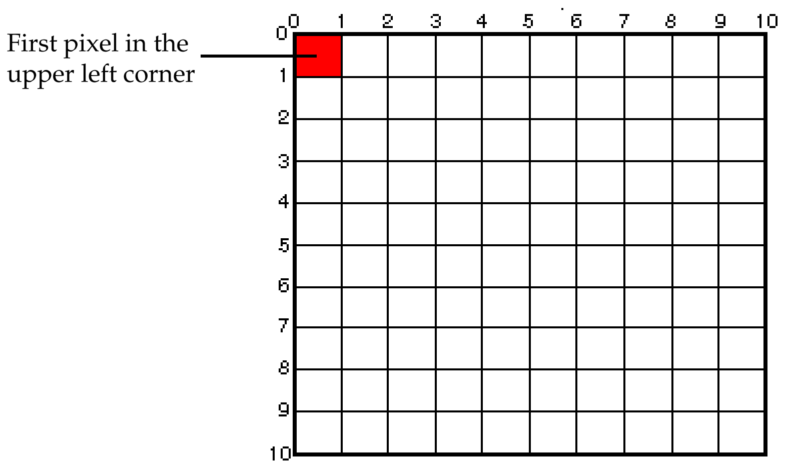

FIGURE 3 Coordinate System of the Graphics Class

6.2.1.2 Drawing Lines

A `Graphics` object must draw a line between the coordinates `(x1,y1)` and `(x2,y2)`. In the coordinate system, that means that the line begins below and to the left of the `(x1,y1)`, and ends below and to the left of `(x2,y2)`. This is shown in the following picture, which shows a line drawn from `(2,2)` to `(6,6)`.

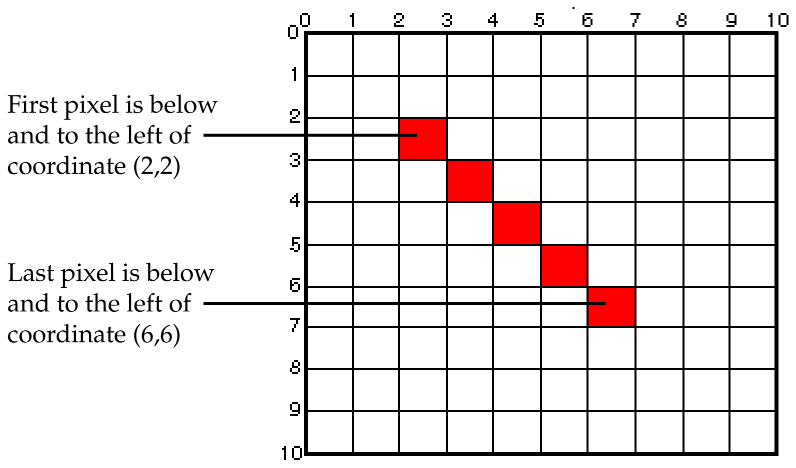
First pixel is below and to the left of coordinate (2,2)
Last pixel is below and to the left of coordinate (6,6)
0 1 2 3 4 5 6 7 8 9 10
0 1 2 3 4 5 6 7 8 9 10

FIGURE 4 Drawing a Line

6.2.1.3 Drawing and Filling Rectangles

A `Graphics` object draws and fills rectangles such that a drawn rectangle will be one pixel larger than a filled rectangle of the same requested height and width. That is, the *MIDP 2.0 Specification* says that the `drawRect` method, “Draws the outline of the specified rectangle...The resulting rectangle will cover an area `(width + 1)` pixels wide by `(height + 1)` pixels tall.” In contrast, the `fillRect` method, “Fills the specified rectangle with the current color.”

The following figure shows two rectangles. The one on the left shows the `drawRect` method drawing a rectangle starting at `(0,0)` with a width of `8` and height of `6`. The one on the right shows the `fillRect` method's rectangle that starts at `(0,0)` and has a width of `8` and height of `6`. That is, they correspond to the calls:

```
drawRect(0, 0, 8, 6);
fillRect(0, 0, 8, 6);
```

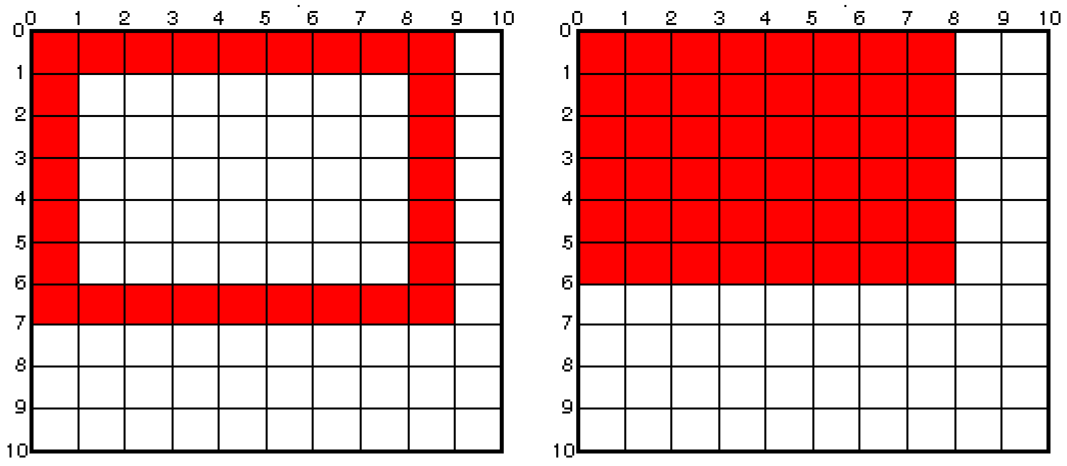

FIGURE 5 Drawing Versus Filling a Rectangle

6.2.1.4 Drawing and Filling Arcs

It can be difficult to understanding the contract that the `drawArc` method must fulfill. This section explains its requirements. Note that the methods that draw and fill arcs have the same size difference as that described in Section 6.2.1.3 "Drawing and Filling Rectangles."

To draw an arc means to draw all or part of an ellipse. The mathematical definition of an ellipse is:

$x^2/a^2 + y^2/b^2 = 1$

The value of the variable *a* controls the width of the ellipse, and the value of the variable *b* controls the height. If a `= 2` and b `= 1`, the ellipse is twice as wide as it is high. It looks like this:

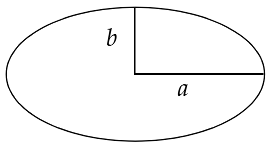

FIGURE 6 Ellipse

If *a* and *b* have the same value, the equation reduces to a circle:

$(x^2 + y^2)/r^2 = 1$

And it looks like this:

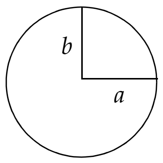

FIGURE 7 Circle

The `drawArc` method specification says:

> Angles are interpreted such that 0 degrees is at the 3 o'clock position. A positive value indicates a counter-clockwise rotation while a negative value indicates a clockwise rotation.

The specification uses the common convention that 0 degrees lies where the circle crosses the positive x axis—sometimes referred to as 3:00 like in the specification—and that positive degrees lie with the positive values on the y axis (they go counter-clockwise) and negative degrees lie with the negative values on the y axis (they go clockwise). The following figure shows the angles at 0, 90, and -90:

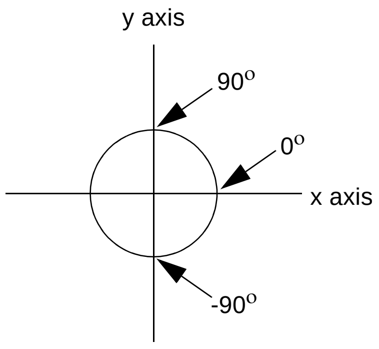

FIGURE 8 0, 90, and -90 degrees on a circle

The specification then says:

> The center of the arc is the center of the rectangle whose origin is (*x*, *y*) and whose size is specified by the width and height arguments.

The `drawArc` method has x, y, width, and height as four of its arguments. Remember that the coordinate system of a canvas has its origin (0, 0) at the upper left corner, the numeric values of the x-coordinates monotonically increase from left to right, and the numeric values of the y-coordinates monotonically increase from top to bottom.

To draw an ellipse, the value of the variable *a* in the ellipse equation must be *w*/2 and the variable *b* must be *h*/2. The center of the ellipse must be at (*x*+(*w*/2), *y* + (*h*/2)). Using these values creates an ellipse that is centered within the rectangle, as specified by the method description. The ellipse is also tangent to the rectangle at four points, which are the minimum and maximum radii of the ellipse; this shows that the size of the ellipse is specified by the rectangle's width and height arguments, as specified by the method description.

The following figure shows a canvas with an arc that meets these requirements; the arc is a full ellipse. The figure shows the ellipse within the rectangle mentioned in the specification. The rectangle is shown with a dashed line because it would not actually be drawn on the canvas. The rectangle's width represented by the variable *w*, and its height by the variable *h*.

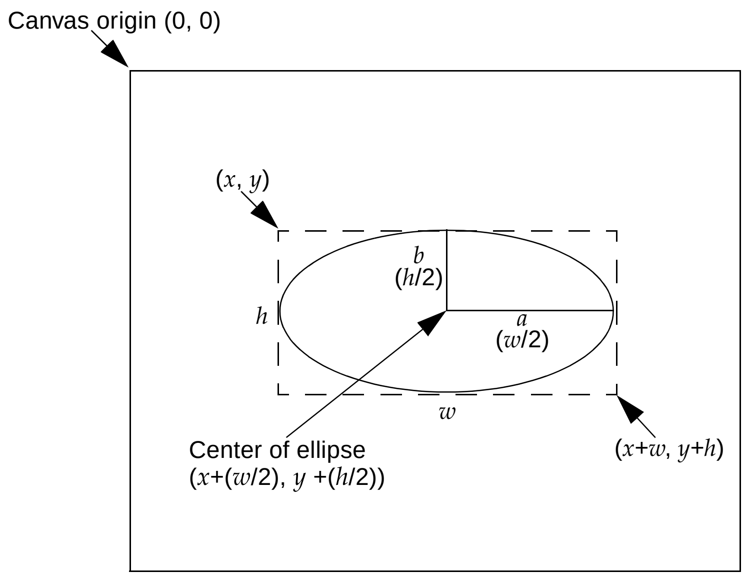

FIGURE 9 Boxed Ellipse

The final part of the specification covers what happens when the method is asked to draw only part of the ellipse. It says:

> The angles are specified relative to the non-square extents of the bounding rectangle such that 45 degrees always falls on the line from the center of the ellipse to the upper right corner of the bounding rectangle. As a result, if the bounding rectangle is noticeably longer in one axis than the other, the angles to the start and end of the arc segment will be skewed farther along the longer axis of the bounds.

In a circle, there is no skewing of the angles, as shown in the 45 and -120 degree angles in the following figure:

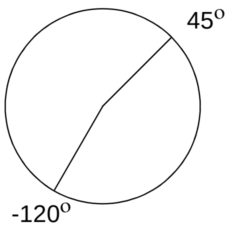

FIGURE 10 45 and -120 degrees on a circle

Skewing appears when the circle is stretched into an ellipse, as shown in the following figure. The point that was 45 degrees on the circle gets shifted, so the angle of the line from the center to that point is no longer 45 degrees. Technically, there is 45 “degrees of arc” between the x axis and the labeled spot, but on paper the angle of the line from the center to that point is not really 45 degrees.

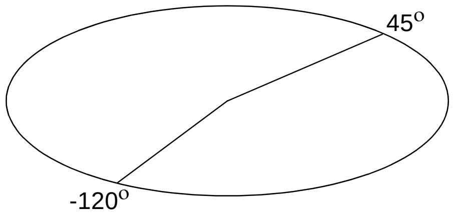

FIGURE 11 45 and -120 degrees on an ellipse

A port of this method must work the same way: it must use the start and end points of a partial arc as though the ellipse were a circle. To make sure that the skewing happens correctly, the *MIDP 2.0 Specification* requires that 45 degrees always falls on the line from the center of the ellipse to the upper right corner of the bounding rectangle, as shown by the circle and ellipse in the following figure. (The figure again shows the bounding box with a dashed line. The figure also extends the 45 degree line so that it touches the box.)

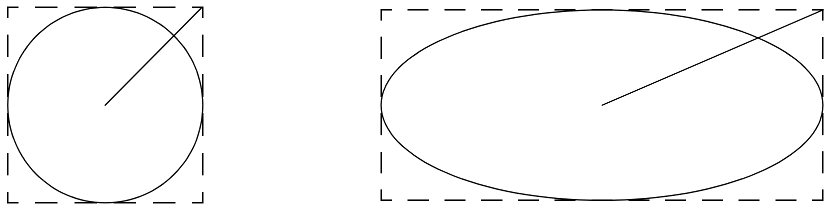

FIGURE 12 Boxed circle and ellipse, both showing the 45 degree angle

The (*x*,*y*) coordinates of a point on a circle are calculated using sine and cosine. The (*x*,*y*) coordinates of the point at *d* degrees on a circle of radius *r* is:

`(x = cos(d)*r, y = sin(d)*r)`

For example, on a circle where *r* = 1 the (*x*,*y*) coordinates for the point that is at 30 degrees are:

```
(cos(30), sin(30))
```

The equation for the corresponding point at *d* degrees on an ellipse with axes *a* and *b* is:

`(x = cos(d)*a, y = sin(d)*b)`

6.2.1.5 Drawing and Filling Rectangles With Rounded Corners

Drawing and filling rectangles with rounded corners combines the previous two topics, drawing rectangles and drawing arcs. (The difference between drawing and filling a rectangle with rounded corners is one pixel in height and width, as discussed in Section 6.2.1.3 “Drawing and Filling Rectangles” on page 34.) A MIDlet draws a rectangle with rounded corners by specifying not only the starting pixel, the height, and the width of the rectangle, but also the width and height of the arc to use to round the corners.

The following figure shows a rectangle drawn with round corners using the call `drawRoundRect(0, 0, 30, 40, 4, 3)`. That is, the rectangle starts at the upper left corner, and is 30 (+1) pixels wide, 40 (+1) pixels high, with corners rounded with an arc that is four pixels wide and three pixels high.

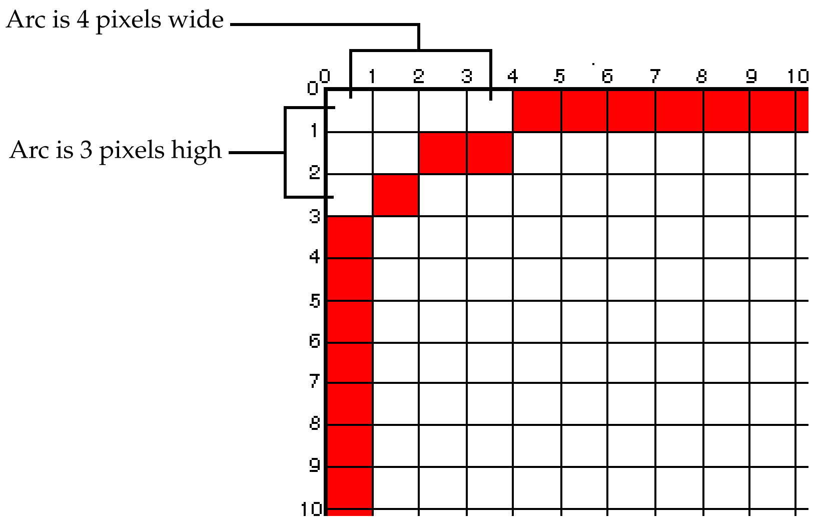

FIGURE 13 Drawing a Rectangle With Rounded Corners

6.2.2 Porting PNG Transparency

The *MIDP 2.0 Specification* supports transparency in Portable Network Graphics (PNG) images. (See `http://www.w3.org/Graphics/PNG/` and `http://www.libpng.org/pub/png` for more information on Portable Network Graphics.) Transparency is useful for creating non-square images. That is, a graphic is a square or rectangular image. If part of that image is opaque and part is transparent, the image can appear to be non-square.

Transparency may exist only in immutable, off-screen images created from PNG images or created from arrays of ARGB (alpha, red, green, blue) data. Moving an image that contains transparency data into a mutable `Image` instance, (*rendering* it) changes the data into a native, opaque format and the transparency information is lost.

One way that an image can encode transparency is by providing image data that specifies the color of each pixel, and a chunk that specifies a color (a *tRNS chunk*). All bits of that color in the image are considered transparent.

Another way that an image can encode transparency is by including an alpha value with each pixel. This is ARGB (alpha, red, green, blue) data. For a black and white image, eight bits encode a pixel's alpha value (`00000000` is transparent and `11111111` is opaque), and eight bits specify its color (`00000000` is black and `11111111` is white). For a color image, some number of bits encode a pixel's alpha value (all ones is fully opaque, all zeros is fully transparent, and values in between is semi-transparent) and the same number of bits specify the red, blue, and green elements of each pixel. The number of bits is usually eight or sixteen; the MIDP Reference Implementation uses sixteen.

An image can also use a *palette alpha* encoding technique, which provides image data that specifies the color of each pixel, and a chunk that specifies alpha values for corresponding RGB values (a *PLTE chunk*).

A compliant MIDP 2.0 implementation must be able to render full transparency. That is, it must treat pixels with an alpha value of zero (or the color in a tRNS chunk) as fully transparent, and must treat pixels with alpha values greater than zero as non-transparent. The MIDP implementation may choose to treat non-transparent pixels as opaque or to implement *alpha-blending*, a technique that renders non-transparent pixels in a way that shows some of whatever is behind them.

A MIDP implementation that supports alpha-blending could use either of two techniques to interpret non-transparent pixels:

- It could composite the non-transparent bits against the screen's background color. This provides the best looking image, but takes a lot of processing and could be impractical on a small device. The MIDP Reference Implementation does not use this technique.
- It could composite the non-transparent bits against some color, such as white. This provides a good looking image, and less processing is required. The MIDP Reference Implementation uses this technique.

However you support transparency in your port, follow this general rule: process transparency data at decode time. There are two reasons for this rule:

- You have all the information at that time, so it's easier to make the picture look as its creator intended it.
- You might be able to combine processing steps to improve the look of the image while decreasing processing time.

6.3 Using Native Widgets for MIDP Screens

The MIDP graphical user-interface should be well integrated with the device, so that a MIDlet looks and behaves as much as possible like a native application. Commonly, ports of the MIDP Reference Implementation replace the implementations of the `Screen` classes with implementations that use the native widgets on a device.

As you replace the MIDP Reference Implementation widgets, keep in mind that `TextBox` and `List` are implemented using `Form`, and all subclasses of `Item` are implemented in the same way as `CustomItem`. The exception to the `Item` subclass implementation is a pop-up choice group. The pop-up choice group is implemented in native code.

This section covers topics that help you replace the Reference Implementation's classes with native widgets:

- General Instructions
- Example: Replacing Part of `DateField`
- Pop-Up Choice Groups

6.3.1 General Instructions

To replace a `Screen` subclass with a native widget:

1. **Remove most of the code for the screen that you want to replace, leaving only empty copies of the public methods.**
2. **Add package-private native methods for `showNotify` and `hideNotify`.**

   ```
   native void showNotify();
   native void hideNotify();
   ```

   Depending on your system, you may also want to handle the `paint` method
3. **Add private native methods for passing data back and forth between your native GUI and the public API**
4. **Fill in the public methods so that they call your natives appropriately.**
5. **Implement the native methods to communicate with your native GUI.**

6.3.2 Example: Replacing Part of `DateField`

This section is an example of how to replace part of the MIDP Reference Implementation's LCDUI code with a native widget. It replaces the time editor for the `DateField` class. The `DateField` class is defined in the *MIDP 2.0 Specification*, which says that the time value, date value, or both can be editable.

This section has the topics:

- Native Time Editor's Interface
- Changing `DateField` and Its Helper Classes

6.3.2.1 Native Time Editor's Interface

For this example, assume that there is a built-in editor for time values that has the following C programming interface:

```
/* set the time.  timeVal is in seconds since midnight */
extern void TimeEditor_setTime(int timeVal);

/* get the time value (in seconds since midnight) */
extern int TimeEditor_getTime();

/* make the time editor visible.  This gives it control
 * over the screen, and causes it to get all user input
 * events (except that the keys used for abstract commands
 * are still passed through to the application).
 */
extern void TimeEditor_setVisible(int shown);

/* repaint the indicated portion of the screen. */
extern void TimeEditor_repaint(int x, int y, int width, int height);
```

6.3.2.2 Changing `DateField` and Its Helper Classes

This section walks through the steps for substituting the native time editor for the one in the MIDP Reference Implementation. The section follows the steps listed in Section 6.3.1 "General Instructions." It has the topics:

- Removing MIDP Reference Implementation Code
- Adding Package Private Native Methods
- Adding Private Native Methods
- Filling in the Public Methods
- Implementing the Native Methods

Removing MIDP Reference Implementation Code

The first step is said to be removing most of the code for the screen that you want to replace. This example is a little more complex, because the MIDP Reference Implementation combines date and time editing into a helper class called `EditScreen`. The example, though, assumes that the device has only a time editor. As a result, the first task is to remove the time-editing code from the `EditScreen` class. You can then create a new helper class that uses the native time editor. The example calls the new class `TimeEditor`. (This example uses only the new class; it does not show the `EditScreen` class with the time editing code removed.)

The following code example shows the empty `TimeEditor` class. Notice that it extends `Displayable`. As mentioned in the introduction, `Displayable` is the highest level class that can be shown on the device screen. It can have a title and abstract commands; these will be added later.

```
class TimeEditor extends Displayable implements CommandListener {

    public TimeEditor(Screen returnScreen, DateField df) {
    }

    public void setDateTime(Date currentValue) {
    }

    public void commandAction(Command cmd, Displayable s) {
    }
}
```

Adding Package Private Native Methods

The second step is to add package-private native methods for `showNotify`, `hideNotify`, and `paint`. The class then adds these lines:

```
class TimeEditor extends Displayable implements CommandListener {
    ...
    public void commandAction(Command cmd, Displayable s) {
    }

    native void paint0(int x, int y, int w, int h);
    native void showNotify();
    native void hideNotify();
}
```

Adding Private Native Methods

The native methods for passing data back and forth between your native GUI and the public API are `getTime` and `setTime`. The following code example shows them in the `TimeEditor` class:

```
class TimeEditor extends Displayable implements CommandListener {
    ...
    private native void paint0(int x, int y, int w, int h);
    native void showNotify();
    native void hideNotify();
    native void setTime(int timeVal);
    native int getTime();
}
```

Filling in the Public Methods

Step four, filling in the public methods, is where you add the code for the constructor and for handling the abstract commands. The following code example shows the whole `TimeEditor` class. The code uses native methods to get and set the time, to show and hide the screen, and to refresh a portion of the display.

The `TimeEditor` also has two abstract commands. Their labels are Done and Back. If the native component handled these commands itself (that is, if it already had a way for the user to leave the editor with and without saving the new value), then you would have to find a way to get those commands back to MIDP.

Finally, note the synchronization done in the `commandAction` method. As noted in Section 5.3.2 "Public Method Conventions" on page 24, `LCDUILock` must be held whenever there is access to shared data. The method, therefore, takes the `LCDUILock` when it sets a value in the date field. Section 5.3.5 "Application Callout

Conventions" on page 27 says that the LCDUILock must not be held during any callout, because doing so may give rise to deadlock. The call to the application's ItemStateChangedListener, therefore, is done after dropping the LCDUILock.

```
class TimeEditor extends Displayable implements CommandListener {
    DateField field;
    Screen returnScreen;

    Command Back = new Command("Back", Command.BACK, 0);
    Command OK   = new Command("Done", Command.OK, 1);

    public TimeEditor(Screen returnScreen, DateField df) {
        field = df;

        addCommand(OK);
        addCommand(Back);
        setCommandListener(this);
        this.returnScreen = returnScreen;
    }

    public void setDateTime(Date currentValue) {
        System.err.println("setDateTime :
                           currentValue = " + currentValue);
        calendar.setTime(currentValue);
        int timeVal = calendar.get(Calendar.HOUR_OF_DAY)*60
            + calendar.get(Calendar.MINUTE);

        setTime(timeVal);
    }
    public void commandAction(Command cmd, Displayable s) {
        Form form = null;
        Item item = null;

        synchronized (Display.LCDUILock) {
            if (cmd == OK) {
                field.saveDate(calendar.getTime());
                item = field;
                form = (Form)item.getOwner();
            }
            currentDisplay.setCurrent(returnScreen);
        } // synchronized

        // SYNC NOTE: Move the call to the application's
        // ItemStateChangedListener outside the lock
        if (form != null) {
            form.itemStateChanged(item);
        }
    }

    void callPaint(Graphics g, Object target) {
        super.callPaint(g, target);
        paint0(g.getClipX() + g.getTranslateX(),
               g.getClipY() + g.getTranslateY(),
               g.getClipWidth(),
               g.getClipHeight());
    }

    private native void paint0(int x, int y, int w, int h);
    native void setTime(int timeVal);
    native int getTime();
    native void showNotify();
    native void hideNotify();

    private Calendar calendar =
        Calendar.getInstance(TimeZone.getDefault());
}
```

Because the helper classes have been changed, the `DateField` class itself must also be changed. The `EditScreen` class now holds only date editing code. Instead of using only calls to `EditScreen`, the `DateField` class must call methods of `TimeEditor` to enable the user to edit the time. Further, because there is no longer a single editor for changing the date and the time, the code in the `DateField` class that provided this capability through the `EditScreen` class must also be changed. The following code examples will show these changes in the `DateField` class's callKeyPressed method. The first code example shows the original MIDP Reference Implementation code. Note that the editor is provided by EditScreen, and there are cases for DATE editing, TIME editing, and DATE_TIME editing:

```
/**
 * Called by the system to signal a key press
 *
 * @param keyCode the key code of the key that has been pressed
 */
void callKeyPressed(int keyCode) {
    if (keyCode != Display.KEYCODE_SELECT) {
        return;
    }

    Screen returnScreen = getOwner();

    if (editor == null) {
        editor = new EditScreen(returnScreen, this);
    }

    switch (mode) {
        case DATE:
            ...
            editor.setDateTime(currentDate.getTime(), DATE);
            break;

        case TIME:
            editor.setDateTime(initialized ?
                               currentDate.getTime() : EPOCH,
                                TIME);
            break;

        case DATE_TIME:
            editor.setDateTime(currentDate.getTime(),
                               (highlight < 1) ? TIME : DATE);
        }

        returnScreen.resetToTop = false;
        returnScreen.currentDisplay.setCurrent(editor);
}
```

The following code example shows the new code. Note that although `EditScreen` still provides a date editor, `TimeEditor` provides the time editor. Also note that there is no longer a `DATE_TIME` editing option.

```
/**
 * Called by the system to signal a key press
 *
 * @param keyCode the key code of the key that has been pressed
 */
void callKeyPressed(int keyCode) {
    if (keyCode != Display.KEYCODE_SELECT) {
        return;
    }

    Screen returnScreen = getOwner();

    int editMode = this.mode;

    if (editMode == DATE_TIME) {
        editMode = (highlight < 1) ? TIME : DATE;
    }

    switch (editMode) {
     case DATE:
        ...
        if ((editor == null) || !(editor instanceof EditScreen)) {
            editor = new EditScreen(returnScreen, this);
        }
        ((EditScreen)editor).setDateTime
            (currentDate.getTime(), DATE);
        break;

     case TIME:
        if ((editor == null) || !(editor instanceof TimeEditor)) {
            editor = new TimeEditor(returnScreen, this);
        }
        ((TimeEditor)editor).setDateTime(initialized ?
            currentDate.getTime() : EPOCH);
    }

    returnScreen.resetToTop = false;
    returnScreen.currentDisplay.setCurrent(editor);
}
```

Implementing the Native Methods

The final step in the task list for substituting a native widget is implementing the native methods that you declared in the "Adding Package Private Native Methods" and "Adding Private Native Methods" sections. The following code example shows the C language functions that satisfy the declarations. The functions use the the native time-editor's' interface. (See Section 6.3.2.1 "Native Time Editor's Interface" on page 43 for its API.) They also use calls to KNI for getting parameters off of and onto the stack. (See the documentation that came with CLDC for more information on KNI.)

```
KNIEXPORT KNI_RETURNTYPE_VOID
   Java_javax_microedition_lcdui_TimeEditor_paint0() {
    int clipHeight = KNI_GetParameterAsInt(4);
    int clipWidth = KNI_GetParameterAsInt(3);
    int clipY = KNI_GetParameterAsInt(2);
    int clipX = KNI_GetParameterAsInt(1);

    TimeEditor_repaint(clipX, clipY, clipWidth, clipHeight);
    KNI_ReturnVoid();
}

KNIEXPORT KNI_RETURNTYPE_VOID
   Java_javax_microedition_lcdui_TimeEditor_setTime() {
    int timeVal = KNI_GetParameterAsInt(1);

    TimeEditor_setTime(timeVal);
    KNI_ReturnVoid();
}

KNIEXPORT KNI_RETURNTYPE_INT
   Java_javax_microedition_lcdui_TimeEditor_getTime() {
    KNI_ReturnInt(TimeEditor_getTime());
}

KNIEXPORT KNI_RETURNTYPE_VOID
   Java_javax_microedition_lcdui_TimeEditor_showNotify() {
    TimeEditor_setVisible(TRUE);
    KNI_ReturnVoid();
}

KNIEXPORT KNI_RETURNTYPE_VOID
   Java_javax_microedition_lcdui_TimeEditor_hideNotify() {
    TimeEditor_setVisible(FALSE);
    KNI_ReturnVoid();
}
```

6.3.3 Pop-Up Choice Groups

In the MIDP Reference Implementation, popup choice groups are implemented differently from other LCDUI items: they are written in native code instead of the Java programming language. (In general, the technique of moving to native code when faced with a problem in the Java programming language layer can be useful.) This section explains why native code is used, so that you can better understand the MIDP Reference Implementation and how to port it.

The reasons for using native code for popup choice group stem from the fact that, for maximum code reuse, the MIDP Reference Implementation gives subclasses of `Item` the same capabilities as those given to the `CustomItem` class by the *MIDP 2.0 Specification*.

The *MIDP 2.0 Specification* allows a custom item to request repaints within its bounds (the space allotted to it on its form). A MIDP implementation must ensure that the item does not repaint other parts of the screen. A popup choice group, as currently designed, needs to draw its popup window of elements beyond its granted item bounds. Because a custom item cannot do this, the current MIDP Reference Implementation architecture makes it impossible for a pop-up choice group written in the Java programming language to draw its popup window of elements.

The *MIDP 2.0 Specification* does not give a custom item access to its position on the screen. (A custom item can know its height and width on its form, but there is no API that gives it its location on the screen.) A popup choice group needs access to its location so that it locates its popup correctly. For example, if the choice group is at the bottom of the viewport, the popup should display up from the bottom; if it is at the top of the viewport, the popup should display down from the top. Again, the current MIDP Reference Implementation architecture makes it impossible to write the popup choice group in the Java programming language.

The native functions that implement the popup choice group are `LCDUIupdatePopupElement` and `LCDUIinitPopupMenu`. These methods are called by the methods `updatePopupElements` and `getPopupSelection` in the `ChoiceGroup` class. If you want to port the native functions to use a menu on your device that allows the user to choose only one element from its list but does not have popup behavior, you may. Your port will still be compliant with the *MIDP 2.0 Specification*.

6.4 Porting the Game Package

The `javax.microedition.lcdui.game` package is designed so that it can be implemented using only the functionality provided by the `javax.microedition.lcdui` APIs. For example, methods were added to the `Graphics` and `Image` classes to help develop games. The `Image.getRBG` method is one such addition. Because of this design, the classes in the `game` package need not call native methods directly. If you have ported the LCDUI functionality to your device, you could port the `game` package without changes. (You might want to make changes, however, for better performance.)

The MIDP Reference Implementation implements the `game` package in the Java programming language using the LCDUI APIs. It relies heavily on the `javax.microedition.lcdui.Graphics` class; the methods `clipRect`, `setClip`, and `drawRegion` are extensively used. It requires LCDUI's transparency support for the `Layer` class. The `GameCanvas` class extends the `javax.microedition.lcdui.Canvas` class, adding key latching and an offscreen buffer. The `Tile` and `Sprite` classes are implemented as a predefined series of LCDUI images.

Because perceived performance is very important to games, you might want to enhance your MIDP port by using any available hardware capabilities to speed up portions of the game API. Hardware capabilities that are meant to accelerate gaming content may include:

- Hardware sprites
- Collision detection (bounding box and pixel-level)
- Tiling
- Image compositing

In addition to the hardware capabilities aimed at games developers, your MIDP port might get performance improvements from accelerating the underlying graphics APIs. For example, if your device can do a fast bitblit, you could use this functionality to improvement the speed of rendering `Sprite` and `TiledLayer` objects. In addition, if your device does not have collision detection in hardware, you might want to move the pixel-level collision detection to native code.

CHAPTER 7

Security

This chapter describes the security functionality of the MIDP Reference Implementation. It is implemented in the Java™ programming language, and should therefore run without changes on a new device.

The document *Security for MIDP Applications*, which is part of the *MIDP 2.0 Specification*, gives MIDP implementations freedom in implementing the security functionality. (The document is available from `http://jcp.org/jsr/detail/118.jsp`.) The MIDP Reference Implementation demonstrates just one way. You may decide, given the requirements of a collaborator such as a service provider or network operator, to use a different approach and replace this implementation.

This chapter contains the sections:

- Overview
- Permissions
- Key Storage

7.1 Overview

The *MIDP 2.0 Specification*, available at `http://jcp.org/jsr/detail/118.jsp`, defines a security model that requires MIDlets to have permission to use security-sensitive APIs. This is different from the MIDP 1.0 security model, which had all MIDlet suites operate in a sandbox that prevented access to sensitive APIs or functions of the device.

The highlights of the MIDP Reference Implementation's implementation of the security policy are:

- A protection domain defines a set of permissions that can be granted to its MIDlet suites. The domains used by the MIDP Reference Implementation are defined in the *midpInstallDir*`\appdb\_policy.txt` file, where *midpInstallDir* is the directory that holds your MIDP installation. This file is editable.
- A domain is associated with each public key in the ME keystore, *midpInstallDir*`\appdb\_main.ks`. When the `MEKeyTool` utility imports the key, the domain is either provided or the default domain, untrusted, is associated with the key.
- A MIDlet suite's JAR file can be signed, and the digital signature stored in the JAD file. The `JadTool` utility creates a digital signature for the JAR file with a private key specified by the MIDlet implementor. The utility uses the EMSA-PKCS1-v1_5 encoding method of PKCS #1, version 2.0. (See RFC 2437 at `http://www.ietf.org/rfc/rfc2437.txt`.)
- A MIDlet suite is assigned to a domain when it is installed. The assignment is done one of two ways:
  - Manually, by users installing unsigned MIDlet suites from the command line. (This capability is not available from within the device emulator)
  - Automatically by the MIDP Reference Implementation when users install signed MIDlet suites. The MIDP Reference Implementation assigns the MIDlet suite to the domain associated with the first public key in the ME keystore that it needs to check digital signature. (If the MIDP Reference Implementation does not find the public keys of a signer, it does not install the MIDlet suite.)

See *Using MIDP* for more information on the MIDP Reference Implementation security policy and tools.

7.2 Permissions

MIDlet-suite developers using the MIDP Reference Implementation use the `JadTool` utility to sign a MIDlet suite. (See *Using MIDP* for more information on the `JadTool` utility.) Signing a MIDlet suite involves adding certificates to the JAD file, creating the digital signature of the JAR file, and adding the digital signature to the JAD file. The `JadTool` utility is in the `com.sun.midp.jadtool` package, which is in the *midpInstallDir*`\tools` directory.

When a user attempts to install a MIDlet suite, the MIDP Reference Implementation must determine whether the MIDlet suite can be trusted. The `com.sun.midp.midletsuite.Installer` class makes this determination by looking for and verifying a digital signature of the JAR file and certificates supporting the signature. If the JAR file is signed, the MIDP Reference Implementation expects the signature to be based on X.509 Public Key Infrastructure so that it can verify the signer and trust the MIDlet suite. The functionality supporting the PKI-secured permissions on signed jar files is the same functionality used in HTTPS certificate authentication (This is the `com.sun.midp.ssl` package; its reference documentation is reproduced in Appendix C.)

If a signed MIDlet suite can be trusted, it is installed and assigned a domain. If a signed MIDlet suite cannot be trusted, then it is not installed. (Note that an unsigned MIDlet suite can be installed and, if installed on the command-line, manually assigned a domain, as described in the previous section. See *Using MIDP* for more information.)

The MIDP Reference Implementation associates permissions with domains by reading and parsing the *midpInstallDir*`\appdb\_policy.txt` file. This is done in the classes:

- `com.sun.midp.security.PermissionProperties`
- `com.sun.midp.security.Permissions`

As a MIDlet suite runs, the MIDP Reference Implementation must be able to determine which permissions it has. Protected APIs get the name of the currently running MIDlet from the `com.sun.midp.midlet.Scheduler` class when a MIDlet calls them, and uses the `com.sun.midp.security.SecurityToken` class to see whether the MIDlet has permission to run the method. When you port the MIDP Reference Implementation, ensure that all paths through security-sensitive APIs perform permission checks.

At times, the user must explicitly give or deny a MIDlet suite permission to access a protected API. The classes that query the user for permission are:

- `com.sun.midp.security.PermissionDialog`
- `com.sun.midp.security.RadioButtonSet`

In addition to granting MIDlet suites the permissions to which they are entitled (and keeping them from using protected APIs for which they do not have permission) the MIDP Reference Implementation must also enable internal classes to perform their duties, including calling security-sensitive APIs when necessary. The security tokens for the internal classes are provided by the `com.sun.midp.security.ImplicitlyTrustedClass` interface. The class `com.sun.midp.Main`, being the first class running in the virtual machine (VM), uses the `ImplicitlyTrustedClass` interface to create and issue `SecurityToken` objects to the internal classes that need them. When you port the MIDP Reference Implementation, you must ensure that these security tokens are not available to applications.

7.3 Key Storage

To change the way public keys are stored, replace the `com.sun.midp.publickeystore` package. The following classes define a simple serialization mechanism (that is, a simple form of the mechanism defined by the `java.io.Serializable` interface in the Java 2 Standard Edition (J2SE™) platform) for storing and retrieving the public keys in the keystore:

- `com.sun.midp.publickeystore.Storage`
- `com.sun.midp.publickeystore.InputStorage`
- `com.sun.midp.publickeystore.OutputStorage`

The `com.sun.midp.publickeystore.PublicKeyInfo` class contains the information serialized for a public key.

The `com.sun.midp.publickeystore.PublicKeyStore` class defines a read-only ME keystore. It uses the `PublicKeyInfo` and `InputStorage`, above.

The following classes extend `PublicKeyStore` to create a read/write ME keystore so that keys can be managed:

- `com.sun.midp.publickeystore.PublicKeyStoreBuilder`
- `com.sun.midp.publickeystore.PublicKeyStoreBuilderBase`
- `com.sun.midp.publickeystore.WebPublicKeyStore`

If you change the key storage classes, you might also need to change the utility, `MEKeyTool`, that manages the keystore. The classes that comprise the `MEKeyTool` utility are in the `com.sun.midp.mekeytool` package. The classes are `MEKeyTool` and its inner classes:

- `TLV` — A type, length, value trio.
- `UsageException` — Error made by the user when calling the utility.

The `com.sun.midp.mekeytool` package is in the *midpInstallDir*`\tools` directory.

CHAPTER 8

Networking

This chapter describes porting and customizing the networking protocols of the MIDP Reference Implementation: HTTP, HTTPS, comm, Socket, Server Socket and UDP Datagram. The protocols are built on the Generic Connection Framework of CLDC, defined in the specification from the “J2ME™ Connected, Limited Device Configuration” (JSR-000030). For more information on CLDC, see `http://jcp.org/jsr/detail/30.jsp`.

The networking protocols implementation has both a native layer and a Java™ programming language layer (Java layer). This chapter covers the layers in the sections:

- Porting the Native Layer
- Customizing the Java Layer

Many files make up the networking implementation; they are listed in the sections below. Note that the MIDP 2.0 Specification requires support for HTTP and HTTPS, but it only recommends support for datagram connections, server socket stream connections, socket stream connections, and secure socket stream connections. If your port does not support one or more of the recommended protocols, you can remove its associated files to improve the footprint of your implementation. There is one exception: do not remove the `com.sun.midp.ssl` package. Deleting it will break the implementation of PKI secured permissions on signed JAR files.

In addition to porting the protocols in the MIDP Reference Implementation you can add other network protocols. For example, if you add support for the “Wireless Messaging API” (JSR-000120) to your MIDP port, you would add SMS (Short Message Service) and CBS (cell broadcast message) protocols.

8.1 Porting the Native Layer

Native code provides IP networking support. The socket and datagram implementations for the Windows 2000 operating system and the Solaris™ Operating Environment (OE) are very similar. They share the following files, which assume a Posix layer for the networking calls to `socket`, `connect`, `bind`, and so on. You will need to reimplement the functionality in these files if the networking on your device is not Posix based.

- `share/native/socketProtocol.c`
- `share/native/datagramProtocol.c`
- `share/native/commProtocol.c`

The MIDP Reference Implementation also has a set of native files that mediate between the Java layer and the shared files in the native layer. The code in the files handles arguments and checks parameters. The platform-specific files are:

- For the Solaris OE:
  - `solaris/native/commProtocol_md.c`
  - `solaris/native/socketProtocol_md.c`
  - `solaris/native/datagramProtocol_md.c`
- For Windows 2000:
  - `win32/native/dirent.c`
  - `win32/native/commProtocol_md.c`
  - `win32/native/socketProtocol_md.c`
  - `win32/native/datagramProtocol_md.c`

8.2 Customizing the Java Layer

This section provides a high-level description of the Java layer of the networking code. It points out places where you could get improved performance or footprint by re-implementing the functionality in native code, and places where such replacement would probably not provide much improvement. It has the sections:

- Generic Connections
- Comm
- IP Support
- Secure Connections
- HTTP 1.1
- HTTPS
- Internal Utilities

8.2.1 Generic Connections

The *CLDC 1.0 Specification* establishes the basic architecture for all stream based IO connections. The architecture is called the Generic Connection Framework (GCF). The architecture is both compact and extensible. A single class, `Connector`, is a factory for specific protocol handlers.

The MIDP reference implementation adds classes that handle the fundamental capabilities of generic connections.

- `com.sun.midp.io.ConnectionBaseAdapter` — Base class for some of the common functionality needed to implement a CLDC Generic Connection.
- `com.sun.midp.io.NetworkConnectionBase` — Base class for network connection protocols; if necessary, it enables network initialization

These are Java programming language implementations of high level stream handling code. Although you could replace some of this functionality with native code, there might not be a significant savings in footprint or performance.

8.2.2 Comm

The Comm protocol provides a way to access external devices using a local serial port as a stream connection. The MIDP Reference Implementation implements the comm protocol in the following classes:

- `javax.microedition.io.CommConnection`
- `com.sun.midp.io.j2me.comm.Protocol`

These classes should not need to be changed when you port the MIDP Reference Implementation to a new device.

8.2.3 IP Support

The MIDP Reference Implementation uses the following classes to provide IP networking support:

- Socket:
  - `javax.microedition.io.SocketConnection`
  - `com.sun.midp.io.j2me.socket.Protocol`
- Server Socket:
  - `javax.microedition.io.ServerSocketConnection`
  - `com.sun.midp.io.j2me.serversocket.Socket`

- Datagram:
  - `javax.microedition.io.UDPDatagramConnection`
  - `com.sun.midp.io.j2me.datagram.Protocol`
  - `com.sun.midp.io.j2me.datagram.DatagramObject`

These classes should not need to be changed when you port the MIDP Reference Implementation to a new device.

8.2.4 Secure Connections

The following interfaces in the *MIDP 2.0 Specification* define the secure connection functionality:

- `javax.microedition.io.SecureConnection`
- `javax.microedition.io.SecurityInfo`
- `javax.microedition.pki.Certificate`

The definition also includes the class `javax.microedition.pki.CertificateException`

According to the *MIDP 2.0 Specification*, a secure connection must implement one or more of the following specifications:

- **Transport Layer Security (TLS)** — TLS Protocol Version 1.0 as specified in RFC 2246. (See `http://www.ietf.org/rfc/rfc2246.txt`.)
- **Wireless Application Protocol (WAP) TLS** — *WAP TLS Profile and Tunneling Specification*, *WAP-219-TLS-20010411-a* at `http://www.wapforum.com/what/technical.htm`.
- **Secure Socket Layer** — SSL V3 as specified in *The SSL Protocol Version 3.0* at `http://wp.netscape.com/eng/ssl3/draft302.txt`.

The MIDP Reference Implementation uses the following classes to provide Secure Socket Layer (SSL) functionality:

- `com.sun.midp.io.j2me.ssl.Protocol`
- `com.sun.midp.ssl.SSLStreamConnection`
- `com.sun.midp.ssl.Record`
- `com.sun.midp.ssl.In`
- `com.sun.midp.ssl.Out`
- `com.sun.midp.ssl.X509Certificate`
- `com.sun.midp.ssl.Cipher`
- `com.sun.midp.ssl.Handshake`
- `com.sun.midp.ssl.Session`
- `com.sun.midp.ssl.Utils`
- `com.sun.midp.ssl.Alg2`
- `com.sun.midp.ssl.RSASig`
- `com.sun.midp.ssl.Signature`
- `com.sun.midp.ssl.SSLSecurityInfo`
- `com.sun.midp.ssl.Alg3`
- `com.sun.midp.ssl.CertStore`

- `com.sun.midp.ssl.MessageDigest`
- `com.sun.midp.ssl.SecretKey`
- `com.sun.midp.ssl.Key`
- `com.sun.midp.ssl.RSAPublicKey`
- `com.sun.midp.ssl.PublicKey`
- `com.sun.midp.ssl.CryptoException`
- `com.sun.midp.ssl.RandomData`
- `com.sun.midp.ssl.RSAKey`
- `com.sun.midp.ssl.RSAPrivateKey`
- `com.sun.midp.ssl.PrivateKey`
- `com.sun.midp.ssl.KeyBuilder`
- `com.sun.midp.ssl.MD5`
- `com.sun.midp.ssl.PRand`
- `com.sun.midp.ssl.MD2`
- `com.sun.midp.ssl.SHA`

The interfaces and classes above support the HTTPS protocol and network connections that have URIs beginning with `ssl://host:port`. (For more information on HTTPS, see Section 8.2.6 "HTTPS" on page 63.) The SSL functionality also supports the PKI secured permissions on signed JAR files. (See Chapter 7, "Security" for more information on signed JAR files and other security topics.)

If your device will provide a secure connection using a different specification, do not remove the `com.sun.midp.ssl` package. Deleting it will break the implementation of PKI secured permissions on signed JAR files.

8.2.5 HTTP 1.1

The MIDP Reference Implementation implements the HTTP 1.1 protocol atop its socket URL support. (See Section 8.2.3 "IP Support" on page 59 for more information.) The HTTP 1.1 implementation is in the following Java classes:

- `javax.microedition.io.HttpConnection`
- `com.sun.midp.io.j2me.http.Protocol`
- `com.sun.midp.io.j2me.http.StreamConnectionElement`
- `com.sun.midp.io.j2me.http.StreamConnectionPool`

This organization provides for maximum portability but might not provide the most optimum performance or size. If your device has a native implementation of HTTP, you might gain efficiency if you change the classes to take advantage of your device's native functionality.

8.2.5.1 HTTP Requests Using Proxies

A high level feature in the `com.sun.midp.io.j2me.http.Protocol` class is making requests through a proxy server. The implementation requires that HTTP proxy servers support the generic tunneling mechanism for TCP based protocols through Web proxy servers. See the following documents for more information on HTTP tunneling:

- `http://www.globecom.net/ietf/draft/draft-luotonen-web-proxy-tunneling-01.html`
- `http://www.ietf.org/rfc/rfc2817.txt`

If you modify the HTTP implementation to be layered on top of a WAP Gateway, the WAP Gateway would replace the proxy access. Replace the references to a generic HTTP proxy with calls to a native WSP stack.

(To have the device emulator use a proxy server, set the `com.sun.midp.io.http.proxy` configuration parameter. See *Using MIDP* for more information.)

8.2.5.2 HTTP1.1 Persistent Connections

The *MIDP 2.0 Specification* supports HTTP 1.1 persistent connections. Do *not* change the implementation to include only HTTP 1.0 connection behavior. Persistent connections are important because they:

- Open and close fewer TCP connections. This saves CPU time and memory throughout the network.
- Reduce latency on subsequent requests, since there is no time spent in TCP's connection-opening handshake.
- Allow HTTP requests and responses to be pipelined. (That is, it allows requests to be made without first getting responses to preceding requests.) This improves the efficiency because not as much time is spent waiting on the network connection.
- Allow HTTP exchanges to degrade gracefully, since errors can be reported without closing the TCP connection. A client using a future version of HTTP might try a new feature and, if the server were older and reported an error, could use the same TCP connection to retry the request using old HTTP semantics.

Persistent connections have many advantages, but on small devices with limited resources they can be a problem if they are not closed properly. The MIDP Reference Implementation uses the value of the configuration property `com.sun.midp.io.http.persistent_connection_linger_time`, set in `com/sun/midp/io/j2me/http/Protocol.java`, as the time that a connection can remain open and unused. Its default is 60000 ms. (60 seconds). Once the time limit is exceeded, the MIDP Reference Implementation closes the connection and removes it from the connection pool. (See *Using MIDP* for the location of the property in the `midp` executable's configuration files.)

8.2.6 HTTPS

The *MIDP 2.0 Specification* requires a secure HTTP (HTTPS) connection be compliant one or more of the following specifications:

- **HTTP over TLS** — See HTTP Over TLS as specified in RFC 2818 a `http://www.ietf.org/rfc/rfc2818.txt` and TLS Protocol Version 1.0 as specified in RFC 2246 at `http://www.ietf.org/rfc/rfc2246.txt`.
- **SSL V3** — See *The SSL Protocol Version 3.0* at `http://wp.netscape.com/eng/ssl3/draft302.txt`.
- **WTLS** — See the Wireless Transport Layer Security document WAP-199 at the WAP Forum Specifications June 2000 (WAP 1.2.1) conformance release at `http://www.wapforum.org/what/technical_1_2_1.htm`.
- **Wireless Application Protocol (WAP) TLS** — *WAP TLS Profile and Tunneling Specification*, *WAP-219-TLS-20010411-a* at `http://www.wapforum.com/what/technical.htm`.

The HTTPS implementation in the MIDP Reference Implementation is built on top of HTTP. (See "HTTP 1.1" on page 61 for more information.) It uses the secure socket layer (SSL) implementation to make secure connections, and uses the implementation of the `SecureConnection` for all certificate handling and data encryption. (See Section 8.2.4 "Secure Connections" on page 60 for more information on SSL and on `SecureConnection`.)

The MIDP Reference Implementation implements HTTPS with the following Java classes:

- `javax.microedition.io.HttpsConnection`
- `com.sun.midp.io.j2me.https.Protocol`

The implementation of HTTPS does not expose an API to control the handshake listener. Exposing such an API would allow certain certificate errors to be over-ridden by higher level applications. It should not be possible for an end-user to override policy decisions about expired certificates. (Not exporting this API is compliant with the *MIDP 2.0 Specification*.)

The organization of the Java layer for HTTPS provides for maximum portability but might not provide the most optimum performance or size. If your device has a native implementation of HTTPS, you might gain efficiency if you change the classes to take advantage of your device's native functionality. To replace the entire HTTPS implementation, replace the `com.sun.midp.io.j2me.https` package.

8.2.7 Internal Utilities

The MIDP Reference Implementation implements the following utilities in the Java programming language:

- `com.sun.midp.io.ConnectionBaseAdapter`
- `com.sun.midp.io.Base64`
- `com.sun.midp.io.HttpUrl`
- `com.sun.midp.io.Properties`
- `com.sun.midp.io.NetworkConnectionBase`
- `com.sun.midp.io.BufferedConnectionAdapter`
- `com.sun.midp.io.ResourceInputStream`
- `com.sun.midp.io.SystemOutputStream`
- `com.sun.midp.io.Util`
- `com.sun.midp.io.BaseInputStream`
- `com.sun.midp.io.BaseOutputStream`
- `com.sun.midp.io.InternalConnector`

They should not need to be changed when the MIDP Reference Implementation is ported to a new device.

CHAPTER 9

The Application Management System

This chapter describes how to port and customize the application management system (AMS), which manages MIDlet suites on a device or device emulator. AMS loads, installs, lists, updates, and removes MIDlet suites. On development platforms it performs these functions either from a command line or from a graphical user interface.

The AMS implementation has both a native layer and a Java™ programming language layer (Java layer). The files for the native layer are in the `src/share/native/kvm` directory and include the files `main.c`, `midpServices.c`, `JarReader.c`, and `ResourceInputStream.c`. The Java layer includes the packages `com.sun.midp.dev`, and `com.sun.midp.midletsuite`.

In addition to its own files, the AMS implementation requires persistent storage. This chapter assumes that you have ported it as described in Chapter 4, "Persistent Storage." The AMS implementation also depends the modules `com.sun.midp.security`, `com.sun.midp.publickeystore`, and `com.sun.midp.midlet`.

This chapter contains the sections:

- Porting `main.c`
- Customizing AMS

9.1 Porting `main.c`

The start up and initialization of the AMS runtime environment is implemented in the `src/share/native/kvm/main.c` file. The functions in the file parse the command line and initialize the command state structure that holds internal information. It sets up a command loop for the Java program `com.sun.midp.Main`, and then starts and stops the virtual machine (VM). (The VM is restarted between AMS commands.) The functions in `main.c` also clean up and reset the classpath between VM invocations.

The command state structure that `main.c` uses for its initialization is updated within `Main.java`, in order to save state between successive launches of the virtual machine. (For example, `Main.java` saves information for the push functionality. See Section 10.2 "Porting the Native Layer" on page 72 for more information on push.)

As `main.c` carries out its tasks, its functions use the native storage interface. See Chapter 4, "Persistent Storage" if you have not yet ported the storage module. If you have changed the native calls for persistent storage, you must update `main.c` to match it.

If your implementation has a different way of specifying command line parameters, change `main.c` so that it can parse the new command line syntax.

Other changes, such as adding or removing commands or command options are part of customizing the AMS. Customization is covered in the next section.

9.2 Customizing AMS

There are many ways to customize AMS. For example you could:

- Remove one or more commands or command options from the `midp` command
  - To make the smallest number of code changes, remove the commands or command options from `main.c`.
  - To make the MIDP executable as small as possible, remove the commands or command options from both `main.c` and `Main.java`.

- Change the way the `midp` executable processes commands (for example, to enable users of the emulator to do part of a MIDlet installation on the command line and part using the graphical user interface on the emulator)

  To do this, replace the `com.sun.midp.main.CommandProcessor` class. In addition, you might also change parts of the `com.sun.midp.midletsuite` package, such as the `Installer` class, or the `com.sun.midp.dev` package, such as the Graphical`Installer` class.

- Incorporate native functionality into the user experience for over-the-air provisioning (for example, to use a native browser for fetching documents using HTTP connections, checking for appropriate MIME types of returned resources, and so on)

  To do this, rewrite the MIDlet that provides the functionality. The MIDlet is implemented in the `com.sun.midp.dev.GraphicalInstaller.java` class. (You might do this work as part of changing the way that MIDP processes commands, described previously.)

- Change the network protocol used to download MIDlets onto the device from HTTP to some other protocol

  To do this, modify the `com.sun.midp.midletsuite.Installer` class. It implements the MIDlet suite interface the scheduler uses.

- Incorporate native functionality into the user experience for AMS. (for example, to have MIDP provide data on the installed MIDlets to the device operating system, which could then display the MIDlets with its native applications; in this scenario, if the user launched a MIDlet, MIDP could accept the storage name from the device operating system and use it to launch the MIDlet)

  To do this, change `main.c` and `Main.java`.

- Replace one or more graphical application management screens (such as the one shown in *Using MIDP* that lists the MIDlets installed on the device)

  To do this, change the MIDlet that provides the functionality. The MIDlet is implemented in the `com.sun.midp.dev.Manager` class.

- Replace AMS with a new Java layer to provide a different user experience with portable code that can be used in multiple devices

  To do this, consider providing a new `com.sun.midp.midlet.Selector` class, which provides the user experience for selecting a MIDlet or MIDlet suite to run when using the GUI, or a new `com.sun.midp.midlet.GraphicalInstaller` class, which provides the user experience for downloading and installing a MIDlet suite when using the GUI.
- Replace AMS with native code (for example for a resource constrained device, as suggested in Chapter 4, "Persistent Storage.")

  To do this, replace the following packages with native code:
  - `com.sun.midp.midlet`
  - `com.sun.midp.dev`
  - `com.sun.midp.midletsuite`
- Change the user experience when the push functionality launches a MIDlet.

  Update the `com.sun.midp.io.j2me.PushRegistryImpl` class, or, to update the messages shown to the user, update the `com.sun.midp.main.Main` class.

- Change the text or presentation of the security questions that users might be asked when they install a MIDlet suite.

  To update the text, change the strings in the `com.sun.midp.security.Permissions` class. To change the layout of the messages to the user, or the text of the answers to the security questions, update the `com.midp.security.SecurityToken` class.
- Completely remove the Java Application Manager (JAM)

  Removing JAM means removing the command-line interface, the graphical user interface code, the installer, and MIDlet suite storage, to replace it with device-specific functionality. In other words, it means removing all of the application management system except the scheduler. To do this you would remove:
  - `main.c`
  - `midpStartup.c`
  - `commandState.*`
  - `com.sun.midp.main.*`
  - `com.sun.midp.dev.*`
  - `com.sun.midp.midletsuite.*`
  - `com.sun.midp.midlet.Selector`

  The following classes will have code that is no longer used. You should remove the unused code but not the entire class.
  - `com.sun.midp.security.SecurityToken`
  - `com.su.midp.security.Permissions`
  - `com.sun.midp.io.j2me.push.PushRegistryImpl`

  The following classes rely on being initialized by the first class started in the virtual machine (`com.sun.midp.main.Main`). You should make sure to that your new implementation also initializes them.
  - `com.sun.midp.rms.RecordStoreFile`
  - `com.sun.midp.publickeystore.WebPublicKeyStore`
  - `com.sun.midp.io.j2me.http.Protocol`
  - `com.sun.midp.io.j2me.https.Protocol`
  - `com.sun.midp.io.j2me.ssl.Protocol`
  - `com.sun.midp.io.j2me.datagram.Protocol`
  - `com.sun.midp.lcdui.DisplayManagerFactory`
  - `com.sun.midp.midlet.MIDletState`

Whatever customizations you make, you should also remove code you are no longer using. Concentrate your search in the following packages:

- `com.sun.midp.midletsuite`
- `com.sun.midp.midlet`
- `com.sun.midp.dev`

CHAPTER 10

Push Functionality

Push starts a MIDlet in response to receiving an inbound connection for that MIDlet. For a MIDlet to use this functionality, it must add an entry in the Push Registry for the port that it will use and, optionally, the entities from which it will accept a request. (If a MIDlet will use more than one port, it must put multiple entries into the Push Registry.) There are two ways that a MIDlet can add the entry for its inbound connections. One is with an attribute-value pair in the application descriptor file; the other is by adding the entry at runtime.

The push implementation has a native layer and a Java™ programming language layer (Java layer). This chapter discusses how to port and customize the push implementation. Because you could implement push in a number of ways, the chapter begins by covering some design decisions that will affect your port.

The push functionality depends on persistent storage, AMS, security, and networking. The porting of these components was described in earlier chapters.

This chapter has the sections:

10.1 Design Considerations

There are many ways that a device could provide the push functionality. Issues that will affect your design include the protocols that you will accept, how MIDP will listen for messages, buffering, whether the device supports running multiple MIDlets concurrently, and how to handle user interaction. This section covers those decisions.

10.1.1 Protocols

From the protocols that you have supported in your networking port, you can choose which will be available for push functionality. You can have a protocol that accepts messages in your MIDP implementation (for example, a MIDlet might use server sockets if they are supported on your device) but you do not have to use that protocol as a reason to launch a MIDlet. You can also support additional protocols, such as the Bluetooth protocols specified in "Java™ APIs for Bluetooth" (JSR 000082), the short message service (SMS) and cell broadcast service (CBS) specified in "Wireless Messaging API" (JSR 000120), and so on. (For more information on these Java specification requests, see `http://jcp.org`

The MIDP Reference Implementation supports datagram and server socket connections for push functionality.

10.1.2 Listening for Incoming Data

MIDP must listen for inbound connection notifications, so that it can launch a MIDlet to handle the incoming message. It can listen using native callbacks or a polling mechanism. Using native callbacks to implement asynchronous I/O is a better technical solution if the device has multithreading or interrupt callback capability. Comparatively, polling uses more resources because it periodically checks for new input, and it is not as timely if the polling cycle is too large.

The MIDP Reference Implementation uses polling.

10.1.3 Message Buffering

The requirements for message buffering are protocol-specific. A MIDP implementation must support buffering for some protocols, but it is not required for all protocols. What is required, however, is that if your MIDP implementation buffers messages, it must provide them to the MIDlet when the MIDlet is launched and opens the push connection.

When your MIDP implementation supports datagram connections for push, it must buffer at least the first message. MIDP must be able to give the launched MIDlet at least the datagram that caused it to start when the MIDlet opens the `UDPDatagramConnection`.

When your MIDP implementation supports socket connections for push, it does not have to buffer any messages. MIDP must only ensure that the launched MIDlet can open the `ServerSocketConnection` and accept the connection, if the connection hasn't timed out.

If your device has limited space resources, you might decide to limit message buffering for some protocols. For example, if you support SMS messaging, you might be able to cache only a certain number of messages.

The MIDP Reference Implementation buffers the first message and the inbound connection sent to a MIDlet using the datagram protocol. It also accepts the inbound connection for a server socket. MIDP uses the data from these operations to check whether the MIDlet registered to receive data from the message's source. It does this check before disturbing any running application.

10.1.4 User Interaction

When a message arrives for a MIDlet, the push functionality cannot launch the MIDlet without the user's acknowledgement. Users need to be in control so that applications don't randomly disrupt what they are doing. You must decide how to present the request to interrupt the currently running MIDlet for the push. That is, will your users have to allow the interruption each time, or will you give them the ability to allow the MIDlet to be launched on future occasions? Requiring users to agree to launching the MIDlet each time burdens them with more frequent interruptions to which they must respond. On the other hand, if you give them the ability to agree to future launching, you must develop and support a way to let them change their mind at a later date.

The MIDP Reference Implementation gives users the ability to allow future launches of the MIDlet. It also has a preference that users can set so that they can again be asked each time before they are interrupted (or to allow future launches if they had chosen to be asked each time).

10.1.5 MIDlet Concurrency

Some devices can run multiple MIDlets concurrently, while others can run only one MIDlet at a time. Which type of device you have might effect the behavior of your push functionality.

When MIDP starts a MIDlet because of a push message, it must do something with whatever MIDlet is currently running. If your device can run only one MIDlet at a time, MIDP will have to exit the currently running MIDlet so that it can launch the MIDlet receiving a message. If your device can run multiple MIDlets concurrently, you can choose whether MIDP will pause or exit the currently running MIDlet.

Choosing to pause the currently running MIDlet may be easier on the user: when the MIDlet is resumed, the user might be able to pick up from the point where the MIDlet was paused. In addition, pausing a well-behaved MIDlet should give the launched MIDlet access to the full resources of the device because a paused MIDlet should have released its shared resources (such as its network connections, cached images, open record stores, and so on). If you choose to exit the currently running MIDlet, the user will have to start over with it after the launched MIDlet exits.

Exiting the MIDlet, though, has the advantage of forcing the MIDlet to release its resources. Your users will not have failures due to resource shortages caused by a MIDlet that did not release its resources when it was paused.

The MIDP Reference Implementation exits a currently running MIDlet before launching the MIDlet receiving a message.

10.2 Porting the Native Layer

The push mechanism listens for inbound connections on all ports entered into the push registry, and must continue to listen even across restarts of the virtual machine (VM). To listen continuously, the low level implementation must interact with the device to intercept requests to open and close network connections. It must also be able to access and manage the push registry.

To enable this persistence, the native code called JAM (Java application manager) in CLDC that is run prior to launching the VM must open and read the push registry, and open all the inbound connections. This happens in the general start up of the system, as the following pseudo-code for `main.c` shows. The file descriptors can then remain open even when the VM is stopped and restarted.

```
...
pushopen()

do {
  Launch VM
} until done
```

Native code checks the open connections for new messages and manages the push registry. The native code is in the `src/share/native/pushregistry.c` file. (When the VM is running, the class `PushRegistryImpl` uses the functions in the `pushregistry.c` file to check for new connections, and when it adds, deletes, and lists entries in the Push Registry.)

In addition to its own native code the push implementation uses native code for socket and datagram. When the push implementation opens a connection to receive an inbound message, the native code for the message's protocol checks out an open file descriptor. The push mechanism passes the open file descriptor to the MIDlet when it launches the MIDlet. When the MIDlet exits, the native code for the message's protocol checks in the open file descriptor. (For more information on the native code for the network protocols, see Chapter 8, "Networking.")

The MIDP Reference Implementation can easily pass a file descriptor to a MIDlet because CLDC, the push functionality, and the launched MIDlet are in a single process. If your device uses separate processes for the MIDlets and the push functionality the file descriptors may not be as easy to share. You may need to use some form of interprocess communication to achieve the check-in/check-out behavior.

10.3 Customizing the Java Layer

The MIDP Reference Implementation implements the push registry in the Java programming language. It uses the following interface and class:

- `javax.microedition.io.PushRegistry`
- `com.sun.midp.io.j2me.push.PushRegistryImpl`

The `PushRegistryImpl` class handles:

- Polling for inbound connection notifications
- Launching the appropriate MIDlet when an inbound connection is received
- Providing the methods required by the *MIDP 2.0 Specification* for registry maintenance.

The number of changes that you need to make to the `PushRegistryImpl` class to customize the push functionality depends on the decisions you made on the issues in section Section 10.1 "Design Considerations. If your platform is very different (for example, it supports concurrently running MIDlets and you have decided not to exit the currently running MIDlet), you will have to replace some or all of the code.

If you do make major changes to the MIDP Reference Implementation's push code, make sure that your implementation has at most once semantics for launching a MIDlet to receive a message. Never notify the application twice for the same message.

CHAPTER 11

Porting the Audio Building Block

This chapter discusses porting the *Audio Building Block* (ABB), a proper subset of the Mobile Media API (MMAPI) that provides only audio functionality. MMAPI is specified in the "Mobile Media API" (JSR-000135) to support multimedia applications for the Java™ 2 Platform, Micro Edition (J2ME™). See `http://jcp.org/jsr/detail/135.jsp` for more information.

ABB in the MIDP Reference Implementation provides the following functionality:

- Single tones
- Monotonic tone-sequences
- Playback of audio (`.wav`) files

If a device implements the ABB, it must support both single tones and monotonic tone sequences. The ability to playback audio files (also called *sampled audio*) is optional.

This chapter contains the sections:

11.1 Overview

This section provides an overview of the audio building block, and how to implement the one interface that is required for every type of audio support. It covers the topics

- Architecture
- Implementing the Player Interface

11.1.1 Architecture

The ABB has a high-level object, a `Player` that controls media playback. A factory mechanism, the `Manager` (`javax.microedition.media.Manager`), can create a `Player` from either a URL or an `InputStream` object. The manager also provides a call to produce single tones. To get input, the ABB uses the security functionality of the MIDP Reference Implementation. (See Chapter 7, "Security" for more information on the MIDP security model.)

These classes, and others in the ABB, are implemented with both the Java programming language and calls to native code. The following table describes the high-level breakdown between the two for each media type:

TABLE 2 Java Programming Language versus Native Implementation

Feature	Code in the Java Programming Language	Native Code
Single tone	API wrapper: `playTone`	Low-level MIDI tone synthesis
Tone Sequence	API wrapper: `Player` Tone format parsing Tone sequencing	Low-level MIDI tone synthesis
Wave audio playback (Sampled audio)	API wrapper I/O handling Wave file parsing Data flow management	Native audio output device

The division that you use depends on your device. Take as much advantage as you can of the device's native capabilities.

11.1.2 Implementing the Player Interface

The `Player` interface defines the rendering of time-based media data. Its methods manage the Player's life cycle, controls playback progress, and gets the presentation components.

The basic `Player` operations that are common to all media types are the `Player` states, events, controls, and looping. (They are the top-level player API calls.) These operations are usually not CPU-intensive and can be implemented in the Java programming language without sacrificing much performance. The implementation of these basic `Player` operations is referred to as the *player wrapper*.

The CPU-intensive operations that require parsing, decoding, and rendering of the media data are described in the later sections that discuss the media types. The implementation of these data processing operations is referred to as the *playback engine*.

This section discusses how to write classes that implement the `Player` interface, such as the classes in the `com.sun.mmedia` package. This section has the topics:

- General Player States
- Events
- Looping

11.1.2.1 General Player States

A `Player` object can be in one of five states:

- `UNREALIZED` – Has been instantiated but not have the information needed to aquire resources; this is the starting state.
- `REALIZED` – Has the information needed to acquire the media resources; achieving this state can be resource and time consuming
- `PREFETCHED` – Gets scarce or exclusive resources, fills buffers with media data, or performs other start-up processing; the actions of this state reduce startup latency.
- `STARTED` – Runs and processes data; this is the state in which sound played.
- `CLOSED` – Releases most of its resources and must not be used again.

The `Player` class defines six state transition methods: `realize`, `prefetch`, `start`, `stop`, `deallocate`, and `close`. These are all synchronous methods—the methods will not return until the state transition is completed or a `MediaException` occurs. For example, if `prefetch` is not able to acquire an audio resource, it will throw a `MediaException` and the `Player` will remain in the `REALIZED` state.

Typically, a few operations will be carried out in each of these methods. The `Player`'s state will be updated and the method will return.

Some of these operations can be executed in native code. For example, opening the audio device in `prefetch` can be executed in native code. If the operation takes a long time to complete, the implementation should make sure that the native code will not block the operations of the virtual machine (VM). Blocking can occur when the native platform or VM does not support multi-threading in native code, such as in the MIDP Reference Implementation.

In these cases, the implementation should attempt to call the native code in a non-blocking manner and poll the status of the native call until it is finished. The pseudo-code in the following example illustrates this.

```
native void nonBlockingNativeCall();
    native boolean isNativeCallDone();
    :
    nonblockingNativeCall();
    while (!isNativeCallDone()) {
        try {
            wait(100);    // Poll at every 100ms as an example.
        } catch (Exception e) {}
    }
```

11.1.2.2 Events

Events are delivered asynchronously from the `Player` to applications by using the `PlayerListener` interface, which is in the `javax.microedition.media` package. It is recommended that you use a separate thread to deliver the events to the `PlayerListeners`. By doing so, the `Player` operations will not be blocked in case the application is blocked at handling the events.

The implementation of the `Player`'s event delivery mechanism is in the `com.sun.mmedia.BasicPlayer`.

11.1.2.3 Looping

The `setLoopCount` method requests the `Player` to loop for a specified number of times. Looping can generally be implemented in two ways:

- The `Player` wrapper can use Java programming language code to track when the media is finished playing by listening for the `END_OF_MEDIA` event. It can then restart the playback.
- If the underlying playback engine supports the equivalent looping operation, the operation can be performed directly at the engine level without the intervention of the `Player` wrapper. This may result in smoother loopback transitions.

The ABB uses the first technique. The class `com.sun.mmedia.BasicPlayer` implements the looping mechanism.

11.2 Porting Synthetic Tones

The phrase synthetic tone refers to the generation and playback of a simple tone or a monotonic tone sequence. It can provide amusing multimedia experiences, such as playing different ring tones.

A simple tone is defined by a note, a duration, and a volume. A monotonic tone sequence is defined as a list of <*note*, *duration*> pairs and user-defined blocks. A block is a unit which consists of <*note*, *duration*> pairs and can be referenced as a whole from any place in the rest of the tone sequence. For more information about the format and contents of a tone sequence, see the reference documentation for the `javax.microedition.media.control.ToneControl` class.

This section covers porting the generation of synthetic tones. It has the topics:

- Architectural Considerations
- Generating Single Tones
- Generating Tone Sequences

11.2.1 Architectural Considerations

The ABB includes modules written in both the Java programming language and native code to implement synthetic tones. The Java programming language modules are described in TABLE 3. The native modules are described in TABLE 4.

TABLE 3 ABB Java Programming Language Modules for Synthetic Tones

Module	Description
`com/sun/mmedia/TonePlayer.java`	Implements the `javax.microedition.media.Player` interface for tone sequences.

TABLE 4 ABB Native Modules for Synthetic Tones

Module	Description
`src/win32/native/mmaevt.c` and `src/solaris/native/mmaevt.c`	Module to deliver end of media (`EOM`) events from the native layer to the Java layer.
`src/win32/native/mmatone.c` and `src/solaris/native/mmatone.c`	Implementation of synthetic tones and tone sequences.

When porting synthetic tones to a target device platform, you must consider whether it has a native tone generator, and whether the tone generator is available in software or hardware. If the platform does not have a native tone generator, you can implement your own in software, but you must make certain implementation decisions that will affect the quality of the tone. If the tone generator has no mixing support, you can also implement tone and tone sequence mixing in software. Finally, you must decide whether the tone generator will run on a thread in the Java platform or a native thread. This has an impact on how tones and tone sequences are played.

This section guides you in making these decisions. It covers the topics:

- Using a Provided Native Tone Generator
- Determining the Quality of Software-Generated Tones
- Thread Considerations
- Issues in Playing a Tone Sequence
- Mixing Tones and Tone Sequences

11.2.1.1 Using a Provided Native Tone Generator

When porting synthetic tones to a specific device platform, take advantage of any native tone generator that might exist on the device. This is the most efficient and cost-effective way to port synthetic tones. A hardware monotonic tone generator is an example of a generator that might be included on a device platform. If the platform provides a native tone generator, it should also provide native access APIs to start and stop the tone.

11.2.1.2 Determining the Quality of Software-Generated Tones

If a native tone generator is not available on the device platform to which you are porting, you must generate the tone in software. You must also make decisions that will affect the playback quality of the tone. For example, you must decide which wave form to use for the tone, whether to perform extra processing on the tone if the device supports floating point, and so on.

For example, in the ABB on the Solaris™ Operating Environment (OE) and Linux operating system, the decision was made to generate the triangle wave tone in the format of 8Khz/16bit/linear/mono.

The following sections describe some of the factors which you must consider when generating a synthetic tone.

Using Floating Point on the Device Platform

Floating point can be used for a number of tasks with synthetic tones. For example, it can be used for calculating and interpolating the amplitude and frequency of the wave form. It can also be used in data encryption and decryption. In some wireless devices, floating-point support is provided in hardware. A hardware floating-point unit allows calculations to be performed faster and with greater accuracy. If you are porting the MMA to a device that has such a unit, you are encouraged to take advantage of it.

However, most wireless devices do not have a hardware floating-point unit. In these devices, floating-point computation is emulated in software. This can be extremely slow and expensive in terms of memory.

For example, if the platform to which you are porting does not have floating-point support, you can work around it by scaling values by a large factor (such as 100) before the calculation and scaling it back after the calculation. You could also use a lookup table as much as possible: for each data sample, find the closest element in the table. Of course, these methods will sacrifice the accuracy and increase the memory usage to certain extent.

The ABB in the Solaris OE and Linux operating system does not use floating point. It saves the wave-length of each note in an integer lookup table. Whenever interpolation is needed, the ABB scales up the value by 1000, then scales it down after the calculation.

CODE EXAMPLE 1 displays how interpolation is performed in the ABB.

CODE EXAMPLE 1 Interpolation with a Scale Factor of 1000

```
          ...
          ...
          slope = (amplitude) * 1000 /(K[note]/4);
          slopeXFade = (amplitude - *yInterrupt) * 1000 /(K[note]/
   4);
          ...
          ...
            /* triangle wave: 4 discrete fcn's based on phase */
            if (t <= K[note]/4) {
                if (firstPeriod == 1)
                  tonedata = (int)(slopeXFade * t / 1000 +
   *yInterrupt);
                else
                  tonedata = (int) (slope * t / 1000);
            }    else if (t <= K[note]/2) {
                  tonedata = (int)(amplitude - (slope * (t -
   (K[note]/4))/1000));
            } else if (t <= ((3 * K[note])/4)) {
                  tonedata  = (int) ((-1 * slope) * (t - (K[note]/
   2))/ 1000);
            } else {
                tonedata = (int) ((-1 * amplitude) + (slope * (t -
   (3 * (K[note]/4))) / 1000));
            }
```

Choosing the Wave Form of the Tone

A synthetic tone can be generated by various different wave forms, such as sine waves, triangle waves, saw waves, square waves, and so on. Each wave form sounds different when it is played back. Sine waves sound purest, but they are also the most expensive to generate in terms of processing and memory. Square waves sound the most compound and are the least expensive to generate. The quality of triangle waves and saw waves are in between sine and square waves.

The ABB on the Solaris OE and Linux operating system uses the triangle wave. CODE EXAMPLE 1 displays a sample implementation of a triangle wave from the ABB.

Choosing 8-bit versus 16-bit for Audio Format

Tone can be generated in either 8-bit or 16-bit format. Normally, 16-bit format has better sound quality, but it requires more memory. A trade-off must be made between sound quality and memory usage.

Another consideration is whether the device to which you are porting supports 16-bit format audio data. Some audio devices support both 16-bit and 8-bit format, while others support 8-bit only.

The ABB on the Solaris OE and Linux operating system uses 16-bit, because the audio driver is more stable with 16-bit data on the Solaris OE.

The following example shows a sample implementation of 16-bit audio support from the ABB.

```
...
...
    int len =  8000 * 16 * 1 / 8 / 32 & ~3;
...
...
#ifdef BIG_ENDIAN
    data[2*(x-written)] = (char)((tonedata >> 8) & 0xff);
    data[2*(x-written)+1] = (char)(tonedata & 0xff);
#else  /* LITTLE_ENDIAN */
    data[2*(x-written)] = (char)(tonedata & 0xff);
    data[2*(x-written)+1] = (char)((tonedata >> 8) & 0xff);
#endif
```

Choosing a Chunk Size for Data Generation

When generating sampled data for a tone, you should decide how much data is generated per cycle; that is, you should choose a good chunk size. Choosing larger chunk sizes means that more CPU time is required per cycle. If the chunk size is too big, then the first chunk might finish playing before the second chunk is fully generated. The result is a tone that sounds choppy. If the chunk size is too small, the playback by audio device might not be that smooth.

The ABB on the Solaris OE and Linux operating system sets the chunk size to be 32 milliseconds of sampled data. This is an acceptable size for these operating systems.

The following example displays how the chunk size is set to 32 milliseconds of sampled data in the ABB.

```
...
    int len =  8000 * 16 * 1 / 8 / 32 & ~3;
...
```

11.2.1.3 Thread Considerations

Tone generation needs a separate thread to either:

- Periodically generate the sampled data and push it into the audio device, or
- Sleep for the duration of the tone, then wake up to stop the tone

You could use either a Java platform thread or a native thread to perform this task. The following sections describe some of the issues that are involved in deciding which thread to use.

Using Java Platform Threads versus Native Threads

Since the Java programming language has built-in multi-thread support, using those threads is always an option. However, in some cases, Java platform threads could be extremely inefficient. For example, consider the KVM implementation by Sun Microsystems. The entire KVM runs on a single native thread. All of the Java platform threads are green thread: they basically share the time slices within that single native thread. Whenever a Java platform thread invokes a native method, all other Java platform threads are blocked before that native method returns. Therefore, if multiple tones are generated and rendered simultaneously, then there will probably be breakups.

Some device platforms support multiple native threads, while others support only one thread.

If the device platform supports multiple native threads, you should consider using native threads, especially when running on the CLDC/MIDP stack.

For example, consider the function:

```
KNI_RETURNTYPE_INT
    Java_javax_microedition_media_Manager_nPlayTone()
```

in `mmatone.c` in the ABB on the Solaris OE and Linux operating system. This function launches a native thread to generate and render the tone whenever a new tone arrives.

Using the Timer Interrupt on the Device Platform

You should consider using the timer interrupt if it is provided by the device platform. The timer interrupt is usually more accurate than `sleep`, especially the Java platform's `Thread.sleep` method, and it avoids the overhead of creating a separate thread. This is extremely convenient if you use it in conjunction with the native tone generator.

11.2.1.4 Issues in Playing a Tone Sequence

Once you resolve how you are going to generate the single tone, playing a tone sequence seems trivial: simply play the tones one by one. However, there are some issues you must consider: whether to cache the sequence in the Java platform layer (Java layer) or the native layer, and how to parse the tone sequence. This section discusses those issues.

Caching the Sequence in the Java Layer or the Native Layer

Base your decision on where to cache the sequence on the type of single tone generator and the type of thread you use.

- If you generate the single tone by using a Java platform thread, then caching the tone sequence in the Java layer is most straight-forward approach.
- If you use a native tone generator and native thread, then caching the tone sequence in the native layer is the most straight-forward approach. By using the native layer, you avoid either calling back to the Java layer to request the next tone or creating a polling Java platform thread to periodically check the status in the native layer.

For example, the file `mmatone.c` in the ABB on Windows 2000 contains code to cache the entire tone sequence in the native layer and create a periodic multimedia timer (similar to a native thread). This timer periodically wakes up to start or stop a tone, move to the next tone, and so on.

Parsing the Tone Sequence

ABB defines its own tone sequence format as a byte stream. In addition to the <*note*, *duration*> pairs, it also defines the structure component "block" and some control pairs, such as tempo setting, resolution setting, volume setting, long note, and so on. For more information on the tone sequence format, see the refernce documentation for the `ToneControl` class.

Since the format of the tone sequence is not that simple, parsing it is not trivial. You could choose a one-pass or a two-pass parsing approach.

One Pass Parsing:

These are the tasks that you would have to complete for one-pass parsing:

- Scan the sequence and verify the syntax.
- Play the sequence.
- Unroll the blocks on demand.

This approach uses less memory, but may not be efficient if the sequence is played more than once.

Two Pass Parsing:

Two-pass parsing uses more memory than one-pass parsing. These are the tasks that you would have to complete for two-pass parsing:

- First pass:
  - Scan the sequence and verify the syntax.
  - Build a block start index table.
- Second pass:
  - Calculate the flat sequence length.
  - Unroll all the blocks.
  - Scan the sequence and convert the blocks into an internal flat format. The flat format is more convenient for playing the audio data.

11.2.1.5 Mixing Tones and Tone Sequences

Mixing tones and tone sequences can be a very desirable feature in applications, such as games. For example, a background sequence can play while foreground tones are triggered by some event.

You should decide whether to support mixing and how to do it. For example, ABB on the Solaris 8 OE takes advantage of the underlying audio driver's mixing functionality. In the Solaris 7 OE and Linux operating system, there is no mixing support.

You could implement your own tone or tone sequence mixing. In this case, it is recommended that you create only one thread to handle all the tones or tone sequences, instead of creating a new thread for each one.

11.2.2 Generating Single Tones

In the ABB on Windows 2000, single tone generation uses the Win32 MIDI API and multimedia timer to generate the tone.

On the Solaris OE and Linux operating system, the ABB generates single tone in 8khz/16bit/mono/linear format, and renders the generated tone data by using `/dev/audio` and the OSS driver. For each tone or tone sequence, it creates a native thread to generate and render the tone.

11.2.3 Generating Tone Sequences

The `com.sun.mmedia.TonePlayer` class, and the C-language files `mmatone.c` and `mmaevt.c`, are the implementation modules which play back tone sequences. A `TonePlayer` instance can be created by invoking either of these methods:

- `Manager.createPlayer(Manager.TONE_DEVICE_LOCATOR)`, or
- `Manager.createPlayer(InputStream, tone-mime-type)`

The created tone player provides a special type of control, `ToneControl`, to allow the programming of a tone sequence on the fly.

`TonePlayer`'s `setSequence` method converts the original sequence to an integer array consisting of <*note*, *duration*> pairs. Then it calls a native method to pass this array to the native layer.

On the Windows 2000 operating system, the `mmatone.c` module contains a native data structure `TONESEQ` which holds the integer array. There is periodic timer which periodically wakes up to send `Note-On` and `Note-Off` messages to the MIDI synthesizer and advance from one tone to the next one. For more information, see the definition of the `timeTSProc` function in the `mmatone.c` module.

In the Solaris OE and Linux operating system, the `mmatone.c` module contains a native data structure `TONEDATA` which holds the integer array of sequence data. A native thread generates sampled tone data and writes it to the audio device, tone by tone. For more information, see the definition of the `tonemain` function in the `mmatone.c` module.

11.2.3.1 Working with the END_OF_MEDIA Event

In the ABB running on Windows 2000, Solaris OE, and Linux operating system, the native timer or thread reaches the end of the sequence, it must deliver an `END_OF_MEDIA` (`EOM`) event to `TonePlayer` in the Java layer. Since CLDC/MIDP does not support callbacks to Java programming language smethods in the native layer, the ABB uses the MIDP event queue mechanism to deliver this `EOM`. This not only improves the performance but is more responsive than other techniques such as creating a polling Java platform thread to periodically check whether an `EOM` occurs in the native layer.

11.2.3.2 Working with Event Queues

The MIDP event queue is based on KVM. However, enqueue/dequeue operations in the KVM event queue are not thread safe. This is because KVM assumes that there is only one native thread. To work around this problem, the ABB employs the MIDP `read` event function. The ABB posts the `EOM` message to the system event queue (as defined in `mmatevt.c`), then allows the read event function to pick up the `EOM` event from the system event queue.

For more information on how the ABB handles event queues, see the code in the `com.sun.midp.lcdui.DefaultEventHandler` class and the `nativeGUI.c` implementation module.

11.2.3.3 Controlling Playback Volume

The `TonePlayer` in the ABB provides the `VolumeControl` and `ToneControl` classes.

On the Windows 2000 operating system, the ABB uses the MIDI's channel volume to implement `VolumeControl`, and the velocity parameter in the MIDI event message to implement the gain set by the `SET_VOLUME` directive. This avoids changes in one `TonePlayer`'s volume affecting the volume of another `TonePlayer`, if they are playing simultaneously.

In the Solaris OE and Linux operating system, `VolumeControl` is implemented by adjusting the audio device's volume. The `SET_VOLUME` directive is implemented by adjusting the amplitude of the wave form.

11.3 Porting Sampled Audio

Sampled audio consists of successive digital snapshots of an analog audio signal. Each snapshot is called a sample. The accuracy of the digital approximation depends on its resolution in time and its quantization or resolution in amplitude. Resolution in time is defined as the sampling rate; resolution in amplitude is defined as the number of bits used to represent each sample.

As a point of reference, the audio data digitized for storage on compact discs is sampled 44100 times per second and represented with 16-bits per sample.

Sampled audio data can be compressed by removing the redundancy among samples. A number of compression algorithms do this, such as MP3, ADPCM, GSM, and ULAW. Also, sampled audio can be saved in various file formats, such as WAV, AIFF, and AU.

Sampled audio support port is an optional part of the ABB. If you do support sampled audio in your port, you must be able to handle WAV audio files with 8bit, 8K, mono PCM data. You may also support additional formats, but it is not required.

This section covers porting the generation of synthetic tones. It has the topics:

- Architectural Considerations
- Implementing the Playback of Sampled Audio

11.3.1 Architectural Considerations

The ABB implementation of sampled audio has both a Java layer and native modules. The Java layer is described in TABLE 5. The native modules are described in TABLE 6.

TABLE 5 ABB Java Programming Language Modules for Sampled Audio

Module	Description
`com/sun/mmedia/WavPlayer.java`	Implements a `Player` to play back uncompressed WAVE files.

TABLE 6 ABB Native Modules for Sampled Audio

Module	Description
`src/share/native/audiornd.c`	Defines a platform-independent audio renderer wrapper.
`src/win32/native/waveout.c`	Implements an audio renderer using the Win32 `waveout` API.
`src/solaris/native/waveout.c`	Implements an audio renderer using `/dev/audio` on the Solaris OE, and OSS on the Linux operating system.

The basic requirements for a device to support sampled audio playback are:

- Audio output device/hardware to generate the sound
- Basic native APIs to access the audio devices
- Enough I/O performance to sustain audio playback in real time.

  The amount of I/O bandwidth required, in bytes per second, can be calculated by the following formula:

  ```
  ( sample_size_in_bytes * sample_rate * channels ) / 8
  ```

  This assumes the audio data is not compressed (in PCM). Compressed data will have less I/O bandwidth.
- Enough CPU performance to sustain audio playback in real-time.

  Compressed audio typically places more demands on CPU performance than uncompressed audio.

If your device meets the basic requirements, then there are further considerations: which audio formats the device will support, and whether there will be support for mixing audio streams.

11.3.1.1 Supported Audio Formats

The following parameters must be specified for the audio format:

- Sample size – 8-bit or 16-bit per sample
- Sampling rate – typical rates are 8000, 11025, 22050, 44100, 16000, 24000, and 48000
- Number of channels – mono or stereo
- Endianess for 16-bit – little endian or big endian
- Sign—signed or unsigned

An audio device might only be able to render certain formats of sampled audio. For example, the audio driver in the Solaris OE renders big endian data and at the sampling rates of 8000, 11025, 22050, 44100 only. In contrast, the `WaveOut` audio driver on Windows 2000 can render only 8-bit/unsigned data or 16-bit/signed/

little endian data, but it does support a wider range of the sampling rates. It is also possible that the audio driver on a particular wireless device can render 8000 sampling rate/8-bit/unsigned data only.

When you implement the audio player, you must determine:

- The kind of audio format that is supported on the device platform, and
- The format of the audio data that is contained in the audio file

If there is a mismatch between the two, a software format converter must be provided. For example, the `WavPlayer`'s implementation on the Solaris OE in the ABB converts little endian audio data to big endian.

11.3.1.2 Support for Mixing Audio Streams

Mixing allows you to mix multiple audio streams and render them simultaneously. Not all audio drivers support mixing. Some examples of drivers that do not support mixing are the OSS driver on Linux, the `WaveOut` driver in Windows 95, and the `/dev/audio` driver in the Solaris 7 OE.

In some applications, mixing is a highly desirable feature. For example, in a game, a background audio track could be playing all the time; over the background track there could be various sound effects.

Therefore, when you implement an audio player on a particular device platform, you must decide whether to support mixing. If you choose to support mixing and the underlying audio driver does not support it, then you must implement mixing in software.

11.3.2 Implementing the Playback of Sampled Audio

The code that enables audio playback can be broken up into different modules, each having a specific task. In addition to these modules, you might want to implement buffering to create a smoother playback. Finally, if a device plays back sampled audio, it should have a volume control.

This section covers these implementation issues in the topics:

- Modules for Audio Playback
- Buffering

11.3.2.1 Modules for Audio Playback

The data for playing sampled audio goes through a typical set of transformations. FIGURE 14 illustrates the typical data flow that guided the composition of the audio playback modules.

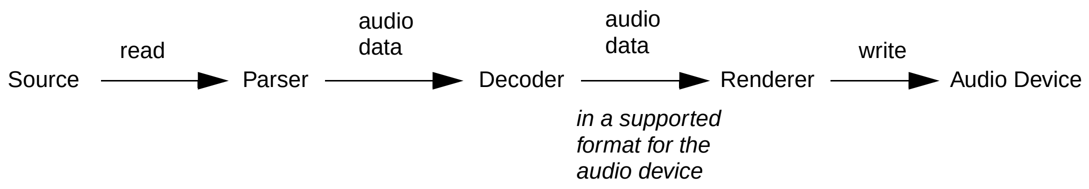

FIGURE 14 Data Flow to Play Back Sampled Audio

The data flow in the illustration is described in the following steps:

1. Data is read from a source.
2. The parser parses the data. It obtains the audio format information and, if necessary, extracts the audio data from the control information. (See "The Parser Module.")
3. The decoder converts the audio data in the format suitable for the audio device. (See The Decoder Module.")
4. The renderer writes the audio data to the audio device driver. (See "The Renderer Module.")

In the Solaris OE, the ABB's decoder is merged with the renderer, since it only does the simple format conversion from 8-bit/unsigned to 16-bit/signed/big endian or from 16-bit/little endian to 16-bit/big endian. For more information on the data processing performed by the converter, see the implementation of the `fmtcvrt` function in `waveout.c`.

The Parser Module

The parser verifies the validity of the WAVE file and parses the file header to obtain the audio format information. This information includes the sample rate, sample size, number of channels, endianess, sign, and the duration of the WAVE file. Typically, as in the `WavPlayer` implementation, parsing is performed in the `Player`'s `realize` method.

In a more complicated file format, the audio data might be interleaved with some control information. In this case, the parser must correctly interpret the control information and extract the actual audio data.

The Decoder Module

The decoder converts the audio data to a format that can be rendered by the audio device. If the audio data in the WAVE file is in a compressed format, MP3 or GSM for example, then the decoder should decompress the data into its uncompressed format.

The Solaris OE version of the ABB uses the decoder to convert audio data to the audio format supported by the Solaris OE. The decoder uses the `fmtcvrt` function in the native file `waveout.c` to convert the data from 8-bit/unsigned to 16-bit/signed/big endian, and from 16-bit/little endian to 16-bit/big endian.

The Renderer Module

The renderer is basically a wrapper for the audio device driver. It provides the function that writes the audio data to the device driver. It also provides other functionalities to control the audio device.

The ABB's `WavPlayer` class uses the renderer module to get control information about the playback data flow to the audio device. The following sections describe how the ABB implements these methods.

Prefetching and Deallocating Device Resources

Typically, the `prefetch` method opens the audio device and initializes it to accept the audio data in a certain audio format. Typically, the `deallocate` method closes the audio device and releases all of the related resources.

Starting and Stopping the Device

The `start` method starts the playback thread and makes the audio device start playing the audio data. The `stop` method pauses the playback thread and pauses the audio device.

Setting the Media Time

The `setMediaTime` method fast-forwards and rewinds the `Player`. It first stops the `Player` if it is playing, then it flushes the audio device to remove any audio data buffered in it. The method positions the media input, such as an `InputStream` instance, at the target location, then calls `start` to start the `Player` if it was playing when `setMediaTime` was called.

Getting the Current Media Time

The `getMediaTime` method reports the current media time. The current media time is defined as the number of microseconds measured from the beginning of the media.

Draining and Flushing the Device

Conceptually, `drain` is a blocking call: it does not return until the audio device consumes all of the audio data sent to it. The `drain` call is useful for delivering an `END-OF-MEDIA` event, because when the audio device finishes playing all of the audio data, it means that it has reached the end of the media.

The implementation of `drain` in the MMAPI RI on the KVM/MIDP platform handles the call differently. Here, the RI transforms the blocking `drain` to a non-blocking `drain`, and periodically polls whether the audio device has finished playing. This implementation of `drain` was chosen because the KVM runs on a single native thread. If a blocking call was made from the Java layer to the native layer, then the entire KVM would be blocked.

The `flush` call simply discards all of the audio data buffered in the audio device.

11.3.2.2 Buffering

Another factor that you must consider when porting the sampled audio player is the buffer size for the audio device. The larger the buffer, the smoother the playback. However, this has the disadvantage of causing a longer latency.

CHAPTER 12

Building Your Port

The MIDP build environment is a set of makefiles, the Java™ platform tools, and the free tools from the CygWin project such as gcc and make. For more information on CygWin tools, see:

`http://sources.redhat.com/cygwin`

The tools and makefiles work together to create a specific *target*. A target is a name in a makefile that is associated with a file or set of related files on which a set of commands will be run. The commands will typically act on the target's files to create, update, or delete them.

The build environment provides *configuration options* to enable you to customize the target. For example, configuration options control whether to include debugging symbols. These configuration options are macros in the makefiles.

Gnumake accepts targets and configuration options on the command line using the general syntax:

```
make [target ...] [config_option=value ...]
```

The `make` command has more options, however. See your make documentation for more information. Instead of being a make tutorial, this section shows you how to use the tool in the context of the MIDP build environment.

This chapter contains advice to help you build different versions of the `midp` executable command. It contains the sections:

12.1 The Build Process

This section covers the makefiles that the `make` command uses, how to run a build, and how the full build works. It contains the sections:

- Makefiles
- Build Instructions
- Example Commands
- Build Steps for the Target `all`

12.1.1 Makefiles

The *Release Contents* contains a list of all files in the build. The table in this section shows only the makefiles, along with a brief description of their purposes, in roughly the order that they are used in the build. In the table, *midpInstallDir* is the directory that holds your MIDP installation. In addition, when a file is in a shared directory, the table shows the directory with Windows-style separators (\).

TABLE 7 Makefiles

Makefile	Directory or directories	Description
`GNUmakefile`	This file is in multiple, device-specific directories: • *midpInstallDir*`\build\win32\kvm\` • *midpInstallDir*`/build/linux/kvm/` • *midpInstallDir*`/build/solaris/`	Top-level makefile; the build starts here
`Platform.gmk`	This file is in multiple, device-specific directories: • *midpInstallDir*`\build\win32\kvm` • *midpInstallDir*`/build/linux/kvm` • *midpInstallDir*`/build/solaris`	Definitions of the target platform and CPU
`Defs.gmk`	*midpInstallDir*`\build\share\makefiles\`	Global definitions (such as common Java programming language and native source files); no targets
`Defs.gmk`	*midpInstallDir*`\build\share\makefiles\kvm\`	Global virtual machine (VM) specific definitions (such as Java programming language and native VM source files); no targets
`Defs-pre.gmk`	This file is in multiple, device-specific directories: • *midpInstallDir*`\build\win32\kvm\makefiles\` • *midpInstallDir*`/build/linux/kvm/makefiles/` • *midpInstallDir*`/build/solaris/makefiles/`	Platform-specific definitions (such as the suffix for executable files); no targets

TABLE 7 Makefiles *(Continued)*

Makefile	Directory or directories	Description
`Defs-post.gmk`	This file is in multiple, device-specific directories: • *midpInstallDir*`\build\win32\kvm\makefiles\` • *midpInstallDir*`/build/linux/kvm/makefiles/` • *midpInstallDir*`/build/solaris/makefiles/`	Platform-specific definitions (such as include files); no targets
`MIDP.gmk`	*midpInstallDir*`\build\share\makefiles\`	MIDP directives and build targets
`CancelImplicits.gmk`	*midpInstallDir*`\build\share\makefiles\`	Directives that cancel the GNUmake implicit rules that MIDP doesn't use; this is for faster builds
`Tools.gmk`	*midpInstallDir*`\build\share\makefiles\`	Definitions and targets for building MIDP tools, such as `MEKeyTool`
`VM.gmk`	*midpInstallDir*`\build\share\makefiles\kvm\`	Definitions and targets for building the VM and its tools (such as the preverifier)
`Example.gmk`	*midpInstallDir*`\build\share\makefiles\`	Definitions and targets for building the samples
`Docs.gmk`	Files with this name are in multiple directories: • *midpInstallDir*`\build\share\makefiles\` • *midpInstallDir*`\build\share\makefiles\kvm\`	Definitions, directives, and targets for using the Javadoc™ tool to build the reference documentation
`Release.gmk`	*midpInstallDir*`\build\share\makefiles\`	Definitions and targets for building release bundles
`Testbeans.gmk`	*midpInstallDir*`\build\share\makefiles\`	Definitions and rules for compiling and installing TestBeans unit test programs into an existing MIDP TestBeans runtime workspace

For more information on the targets in these makefiles, see Appendix A, "Build Targets." For more information on the definitions, see Appendix B, "Configuration Options."

12.1.2 Build Instructions

To build the MIDP executable or documentation bundle:

1. **Change directories to the location of the platform-dependent makefiles.**

   On a Microsoft Windows platform this is the directory *midpInstallDir*`\build\win32\kvm`. For example:

   ```
   c:\> cd midp2.0fcs\build\win32\kvm
   ```

   On a UNIX® platform, the directory is *midpInstallDir*`/build/linux/kvm` or *midpInstallDir*`/build/solaris`.

2. **Choose one or more targets and configuration options.**

   You can choose to use the default target and configuration options, or choose other targets and options.

   The default build target, `all`, creates a MIDP executable and builds the examples. The default configuration options direct the build environment to create an executable that supports the full CLDC reference implementation, supports internationalization, and does not include debugging symbols.

   See Appendix A, "Build Targets" for a list of targets and common build options.

3. **Run the make tool with any targets and configuration options.**

   Providing no arguments uses the default build target and configuration-options.

   For example, to clean up from any previous builds, then build a MIDP executable you would run the command:

   ```
   c:\midp2.0fcs\build\win32\kvm> make clean all
   ```

If the target that you use creates a MIDP executable, it will be placed in the following directory:

- *midpInstallDir*`\build\win32\kvm\bin` on a Microsoft Windows 2000 platform
- *midpInstallDir*`/build/linux/kvm/bin` on a Linux platform
- *midpInstallDir*`/build/solaris/bin` in the Solaris™ Operating Environment.

12.1.3 Example Commands

The following list describes common functionality requirements and the combinations of targets and configuration-options that provide them.

- **Getting better status text in the command window as the VM runs —** The following command builds a MIDP executable that writes more text to the command window, such as stack traces when a method throws an exception:

  ```
  c:\midp2.0fcs\build\win32\kvm> make INCLUDEDEBUGCODE=true
  ```

  The command uses the default build target, `all` and creates the MIDP executable `midp`.

- **Debugging capabilities for native code —** The following command builds a MIDP executable with debugging capabilities. It enables you to build an executable that you can attach to with the C debugger that corresponds to your compiler (for example, `gdb` if you compiled with `gcc`).

  ```
  c:\midp2.0fcs\build\win32\kvm> make DEBUG=true
  ```

  The command uses the default build target, `all` and creates the MIDP executable `midp_g`.

- **Debugging capabilities for Java class files —** The following command compiles the Java class files with information that enables source-code debugging. It enables you to build an executable that you can attach through the KVM debug proxy.

  ```
  c:\midp2.0fcs\build\win32\kvm> make ENABLE_DEBUGGER=true
  ```

  The command uses the default build target, `all` and creates the MIDP executable `midp`.

- **Creating a MIDP Executable but no examples —** The following command provides a faster build time because it creates the MIDP executable but does not build the examples:

  ```
  c:\midp2.0fcs\build\win32\kvm> make midp
  ```

  The command creates the MIDP executable `midp`.

- **Checking that your Java programming language code (Java code) compiles —** The following command creates new, preverified Java class files, and recreates the `classes.zip` file:

  ```
  c:\midp2.0fcs\build\win32\kvm> make classes.zip
  ```

  The command does not create the MIDP executable.

12.1.4 Build Steps for the Target `all`

When you build the MIDP Reference Implementation with the target all, GNUmake performs the following steps:

1. Builds the preverifier
2. Creates the ROMizer
3. Compiles and preverifies Java code
4. Creates the `classes.zip` file
5. Reads `classes.zip` and generates `ROMjava`*platform*`.c`
6. Builds the `extractOffsets` tool
7. Uses the `extractOffsets` tool to generate header files with offsets into the classes of the Java programming language
8. Compiles CLDC and MIDP C files
9. Links the CLDC and MIDP C file
10. Creates the `MEKeyTool` and `JadTool` utilities
11. Creates the *MidpInstallDir*`/build/appdb` directory
12. Compiles, preverifies, and packages the examples (except the HelloMIDlet, which is left for the reader of *Creating MIDlet Suites* to do).

12.2 Adding Files to the Build

The MIDP build environment allows you to add new class files and object files to the build that are not part of the source distribution. To add the files, you must update the contents of the `Defs-pre.gmk` file. (See TABLE 7 on page 96 for more information on this file.)

This section shows you how to update the file in the topics:

- Adding Java Programming Language Files
- Adding C Files

12.2.1 Adding Java Programming Language Files

You can add two types of classes written in the Java programming language to the build: platform-specific and platform-generic.

If you add platform-specific classes, they must be in `PLATFORM_CLASS_DIR`, which has the default value *midpInstallDir*`/src/`*platform*`/classes`. There is no required location for platform-generic classes.

The two types of classes are held in different variables in the `Defs-pre.gmk` file:

- `PLATFORM_INCLUDE_CLASSES` — Holds the file names of platform-specific classes. The filenames that are the value of this variable have the form

  `$(PLATFORM_CLASS_DIR)/`*packageName*`/`*className*`.java`

  Note that these file names use a slash ("`/`") delimiter, even on Windows 2000.
- `MIDP_INCLUDE_CLASSES` — Holds the file names of platform-generic classes.

To add a file to one of these variables, add a line that has the following syntax to the `Defs-pre.gmk` file:

VARIABLE_NAME `+=` *fileName*

For example, to add the platform-specific class `com.MyCompany.productY.Foo`, you would update the `PLATFORM_INCLUDE_CLASSES` variable like this:

```
PLATFORM_INCLUDE_CLASSES += classes/com/MyCompany/productY/foo.java
```

Similarly, to add the platform-generic class `com.MyCompany.midp.bar`, you would update the `MIDP_INCLUDE_CLASSES` variable like this:

```
MIDP_INCLUDE_CLASSES += classes/com/MyCompany/midp/bar.java
```

The classes will be included in the build the next time you use the default target, `all`.

12.2.2 Adding C Files

You can add two types of C language files to the build: platform-specific and platform-generic. Different variables in `Defs-pre.gmk` hold these two kinds of files:

- `PLATFORM_INCLUDE_SRC` — Holds the names of platform-specific C files
- `MIDP_INCLUDE_SRC` — Holds the names of platform-generic C files

To add a file to one of these variables, add a line that has the following syntax to the `Defs-pre.gmk` file:

VARIABLE_NAME `+=` *fileName*

For example, to add the platform-specific file, `productY_natives.c`, you would update the `PLATFORM_INCLUDE_SRC` variable like this:

```
PLATFORM_INCLUDE_SRC += productY_natives.c
```

Similarly, to add the platform-generic file, `common_natives.c`, you would update the `MIDP_INCLUDE_SRC` variable like this:

```
MIDP_INCLUDE_SRC += common_natives.c
```

This is the normal way to add native method implementations to MIDP. The files will be included in the build the next time you use the default target, `all`.

12.3 Removing Unused Code

Removing unused code can improve the footprint of your port. This section covers two areas where you might find code to remove: configuration code and code made obsolete by your customization.

The MIDP Reference Implementation uses a configuration mechanism to select among alternate behaviors at runtime. In commercial ports of the RI, many of these design choices will have been made. (For example, whether the device has a color or grey scale screen.) You can eliminate the branches created by the configuration options that your device does not use. To locate these branches, look for calls to the `Configuration.getProperty` method. The list of classes that include optional behavior based on calls to access configuration parameters is:

- `com.sun.midp.Main`
- `com.sun.midp.io.j2me.http.Protocol`
- `com.sun.midp.io.j2me.https.Protocol`
- `com.sun.midp.io.j2me.socket.Protocol`
- `com.sun.midp.io.j2me.comm.Protocol`
- `com.sun.midp.lcdui.InputMethodHandler`
- `com.sun.midp.lcdui.Resource`
- `com.sun.midp.midlet.Scheduler`
- `com.sun.midp.midletsuite.Installer`
- `javax.microedition.lcdui.Display`

In addition, if you have customized the MIDP Reference Implementation (for example, by re-implementing one or more components in native code), there could be Java programming language or C code that is no longer used. Some examples have already appeared in the text, such as in Section 6.3 “Using Native Widgets for MIDP Screens” on page 42. You can use the examples in this guide as a starting point for a more thorough search.

APPENDIX **A**

Build Targets

This appendix lists some targets available for building the MIDP Reference Implementation. Many targets that operate on the source files are in the top-level makefile, `GNUmakefile`. The file is in the directory *midpInstallDir*`\build`\win32\kvm on Windows 2000 and *midpInstallDir*`/build/linux` or *midpInstallDir*`/build/solaris` on a UNIX® platform.

The targets that create API documentation are in either the makefile *midpInstallDir*`\build\share\makefiles\Docs.gmk` or the makefile *midpInstallDir*`\build\share\makefiles\Release.gmk`. The build environment uses the Javadoc™ software to create documentation targets.

This chapter divides the targets into the following topics:

- Basic Build Targets
- Advanced Build Targets

A.1 Basic Build Targets

The following table describes the main build targets for the MIDP Reference Implementation build environment.

TABLE 8 Basic Build Targets

Target	Purpose	Description
`all`	Release bundle	Creates a MIDP Reference Implementation executable and builds the examples. This is the default target.
auction audiodemo demos fonts games manyballs photoalbum pushpuzzle stock	Release bundle	Builds the named example.
`example`	Release bundle	Builds all of the examples.
`classes.zip`	Release bundle	Compiles and preverifies Java programming language files.
`clean`	Preparation for creating a new build.	Removes all the class files, object files, and other temporary build files from the MIDP Reference Implementation build environment. It does not remove virtual machine (VM) tools or cryptographic object files.
`reallyclean`	Preparation for creating a new build.	Removes everything in the `clean` target, plus the contents of the *midpInstallDir*`\build\appdb` storage directory and all VM tools. It does not remove VM tools or cryptographic object files.
`insanelyclean`	Preparation for creating a new build.	Removes everything in the `reallyclean` target and all crytographic object files. **Note**: Do not use the `insanelyclean` target unless you have the SSL source files supplied under special agreement with Sun Microsystems. If you use this target but do not have access to the cryptographic source, you will not be able to rebuild the MIDP Reference Implementation executable without reinstalling MIDP Reference Implementation.
`docs_html`	Documentation	Generates the API reference documentation in HTML for the MIDP Reference Implementation and CLDC classes.

A.2 Advanced Build Targets

The following table describes some special-purpose targets for creating release documentation and bundles for external use. These build targets are not typically used. For example, the targets that create release bundles are intended to meet the needs of the expert group deploying the specification and the initial reference implementation.

TABLE 9 Advanced Build Targets

Target	Target Purpose	Description
`midp-src`	Release bundle	Creates a source bundle that includes the MIDP Reference Implementation reference implementation source code.
`midp-docs`	Documentation	Creates a documentation bundle that includes the combined CLDC and MIDP Reference Implementation API reference documentation.
`midp-dist`	Release bundle	Creates a release bundle that includes a MIDP Reference Implementation executable and reference implementation.
`docs_all`	Documentation	Generates the API reference documentation in HTML for the MIDP Reference Implementation and CLDC classes, including their private fields and methods.
`docs_mid`	Documentation	Generates the API reference documentation in HTML for the MIDP Reference Implementation classes. Because this target does not create API documentation for CLDC, it does not include links to CLDC classes and methods.

APPENDIX **B**

Configuration Options

Configuration options provide a way to balance memory consumption and functionality by producing targets with different levels of functionality. You can change one or more configuration options, or accept the default configuration.

This appendix covers configuration options in the sections:

The sections have tables of configuration options. The tables list the options in alphabetical order.

B.1 Build Tools and Their Options

The following tables present configuration options that control the command names used by the build environment and their common command options. The tables do not contain lists of files and directory names that might be used by the commands (unless they are common and used by only one command). The configuration options that contain lists of files and directory names are in Section B.2 "File Lists and Locations" on page 113.

TABLE 10 C Compiler Command and Common Command Options

Option	Type	Default Value on Windows 2000	Description
`CC`	`String`	`cl`	Compiler command
`CC_OUTPUT`	`String`	`-Fo`	Argument used with the C compiler to generate the compiled (`.o`) file, so that it appears in the right directory
`CMDLINE_CFLAGS`	`String`	`""`	Compiler arguments that are defined on the makefile command-line **Note:** Do not set or change this configuration option within the makefiles.
`EXTRA_CFLAGS`	`String`	`-DWIN32 -nologo -D_MT -MT -O2 -DUSE_KVM=1 -DGENERIC_IO_WAIT_TIME=10 -DROMIZING=1 -DMAXIMUMHEAPSIZE=500000 -DENABLE_JAVA_DEBUGGER=0 -DINCLUDE_I18N -DUSE_DEBUG_COLLECTOR=0 -DENABLEPROFILING=0 -DRELEASE=\dist" -DIMPL_VERSION="2.0" -DFULL_VERSION=": 10.23.02-09:58" -DPLATFORMNAME="j2me"`	Additional arguments for the C compiler

TABLE 10 C Compiler Command and Common Command Options *(Continued)*

Option	Type	Default Value on Windows 2000	Description
`EXTRA_INCLUDES`	`String`	`-I$(MIDP_DIR)/src/share/native/i18n -I$(MIDP_DIR)/src/<platform>/native/i18n -I$(MIDP_DIR)/src/<platform>/native -I$(MIDP_DIR)/src/<platform>/native/kvm -I$(MIDP_DIR)/src/share/native -I$(MIDP_DIR)/src/share/native/kvm -I$(MIDP_DIR)/src/share/native/crypto -I$(MIDP_DIR)/src/<platform>/native/xpm/include -I$(MIDP_DIR)/src/<platform>/native/png/include`	Additional directories that the C compiler should search for header files
`GCC`	`boolean`	`false`	Whether to use the `gcc` compiler to compile native methods If this value is `false`, the compiler specified by the `CC` option is used.
`MAXIMUMHEAPSIZE`	`String`	`500000`	Amount of memory to be set aside for the heap, in bytes (such as 500000), kilobytes (such as 500k) or megabytes (such as 1M).
`OBJ_DIR`	`String`	`obj$(g)$(ARCH_DIR)`	Directory to hold object files
`OBJ_SUFFIX`	`String`	`obj`	Suffix of the object files generated by the C compiler

TABLE 11 Debugging Command (Extract Offsets) and Common Command Options

Option	Type	Default Value on Windows 2000	Description
`EXTRACT_OFFSETS_CMD`	`String`	`$(BINDIR)/extractOffsets $(g)$(EXE)`	Command that generates header files with offsets into the classes of the Java programming language
`EXTRACT_OFFSETS_INCDIR`	`String`	`$(BUILD_DIR)/include`	Directory to search for header files used to build the extract offsets tool
`EXTRACT_OBJ_DIR`	`String`	`extract_obj$(g) $(ARCH_DIR)`	Directory to hold object files
`g`	`String`	`""`	Suffix of debug-enabled object directory

TABLE 12 Commands and Common Command Options Related to the Java Programming Language

Option	Type	Default Value on Windows 2000	Description
`JAR`	`String`	`$(BOOTDIR)/bin/jar$(EXE)`	Command that runs the `jar` utility
`JAVA`	`String`	`$(BOOTDIR)/bin/java$(EXE)`	Command that runs the Java programming language interpreter
`JAVAC`	`String`	`$(BOOTDIR)/bin/javac$(EXE)`	Command that runs the compiler for the Java programming language
`JAVADOC`	`String`	`$(BOOTDIR)/bin/javadoc$(EXE)`	Command that runs the Javadoc™ tool
`JCC_CMD`	`String`	`$(JCC_DIR)/JavaCodeCompact.class`	Command that runs a compression utility on compiled Java application files
`JCC_DIR`	`String`	`$(BUILD_DIR)/jcc_classes`	Directory that holds the output from running the `JCC_CMD`
`PATHSEP`	`String`	`;`	Path separator for the `-classpath` argument

TABLE 13 KVM and MIDP Related Commands and Common Command Options

Option	Type	Default Value on Windows 2000	Description
`CA_KEYSTORE`	`String`	`$(STORAGEDIR)/_main.ks`	Keystore that MIDP uses when it does secure communications
`EXE`	`String`	`.exe`	Suffix to place at the end of executable programs
`EX_SERVER_URL`	`String`	`http://localhost/`	URL prefix to use for MIDlet-JAR-URL tag when creating the example MIDlets The name of the JAR file will be appended to this string.
`KDP_CMD`	`String`	`$(BINDIR)/kdp.jar`	Command for the KVM Debug Proxy
`MIDP_CMD`	`String`	`midp`	Command to start the MIDP executable
`PREVERIFY_CMD`	`String`	`$(BINDIR)/ preverify$(EXE)`	Command to preverify class files
`ROMJAVA`	`String`	`ROMjavaWin`	Name of the file that holds the results of converting the class files in `classes.zip` to an object-file format for linking with the KVM
`VM`	`String`	`kvm`	The virtual machine used with this release of MIDP
`VM_INCLUDES`	`String`	`-I$(KVM_DIR)/cldc/1.0.4/ kvm/VmCommon/h -I$(KVM_DIR)/cldc/1.0.4/ kvm/VmExtra/h -I$(EXTRACT_OFFSETS_ INCDIR) -I$(KVM_DIR)/cldc/1.0.4/ kvm/VmWin/h`	Directories to search for header files needed to build the virtual machine
`VERIFYDIR`	`String`	`$(BUILD_DIR)/tmpclasses`	The temporary directory that holds the compiled files that will be processed by the preverifier

TABLE 14 Linking Command and Command Options

Option	Type	Default Value on Windows 2000	Description
`LD`	`String`	`link`	Linker command
`LIBS`	`String`	`user32.lib  gdi32.lib kernel32.lib winmm.lib wsock32.lib imm32.lib`	Libraries to append at link stage
`LINKER_OUTPUT`	`String`	`-out:`	Argument used with the linker to generate an executable so that it appears in the right directory

TABLE 15 Command to Create Zip Files

Option	Type	Default Value on Windows 2000	Description
`ZIP`	`String`	`zip`	Command used to create zip files The command must be able to create ZIP files using the following command options: `-qr` If your zip command uses different command options, you must edit the makefiles where the `ZIP` option appears.
`UNZIP`	`String`	`unzip`	Command used to extract files from ZIP files The command must be able to unpack zip files using the following options: `-q -u` If your zip command uses different command options, you must edit the makefiles where the `UNZIP` option appears.

B.2 File Lists and Locations

The following tables present configuration options that set directory names and lists of files:

TABLE 16 Lists of Files

Option	Type	Default Value on Windows 2000	Description
`EXAMPLE_EXCLUDE_CLASSES`	`String`	`SCCS|HelloMIDlet|i18n`	Any class files to exclude from examples
`EXAMPLE_INCLUDE_CLASSES`	`String`	“”	Any `.class` files to include from the examples
`EXAMPLE_SOURCE_FILES`	`String`	Any `.java` files found in the `$(MIDP_DIR)/src/example/` directory, excluding files that match any of the patterns listed in `$(EXAMPLE_EXLUDE_ CLASSES)`.	Example files to compile
`EXTRA_CLASS_FILES`	`String`	“”	Any `.class` files to include from some other tree If you give this option a value, you must also define the proper target(s) to build the classes in this list.
`SSL_SOURCE_FILES`	`String`	any `.java` files found in the `$(MIDP_DIR)/src/share/ classes/com/sun/midp/ssl` directory, if the directory exists	The `.java` files that define the secure socket layer code.

TABLE 16 Lists of Files *(Continued)*

Option	Type	Default Value on Windows 2000	Description
`KVM_EXCLUDE_CLASSES`	`String`	`SCCS|j2se|io/palm| io/j2me| io/connections| io/ConnectionBase.java| io/ NetworkConnectionBase.java| io/DateParser.java| java/lang/System.java| java/lang/Runtime.java| java/lang/Class.java`	Any class files to exclude from the KVM tree
`KVM_DEF_SRC`	`String`	`cache.c class.c fields.c frame.c garbage.c global.c interpret.c loader.c native.c pool.c thread.c nativeCore.c loaderFile.c runtime_md.c events.c hashtable.c profiling.c StartJVM.c verifier.c verifierUtil.c log.c stackmap.c execute.c inflate.c jar.c kni.c async.c`	Default source files that make up the virtual machine
`KVM_EXCLUDE_SRC`	`String`	`SCCS`	Any `.c` files to exclude from KVM tree
`KVM_INCLUDE_CLASSES`	`String`	“”	Any `.class` files to include from KVM tree
`KVM_INCLUDE_SRC`	`String`	“”	Any `.c` files to include from the KVM tree

TABLE 16 Lists of Files *(Continued)*

Option	Type	Default Value on Windows 2000	Description
`MIDP_DEF_SRC`	`String`	`property.c configuration.c imageDecode.c pngDecode.c defaultLCDUI.c menus.c popup.c nativeGUI.c images.c text.c graphics.c commandState.c exitInternal.c main.c midpStartup.c SystemOutputStream.c ResourceInputStream.c JarReader.c storage.c RandomAccessStream.c storageFile.c socketProtocol.c socketProtocol_md.c datagramProtocol.c datagramProtocol_md.c commProtocol.c commProtocol_md.c phonesim.c staticGraphics.c midpEvents.c pushregistry.c midlet.c midlet_md.c audiornd.c mmatone.c mmaevt.c waveout.c vibrate.c pvibrate.c`	Default MIDP source files Some options, such as `ENABLE_SCREEN_CAPTURE`, add to this list.
`MIDP_EXCLUDE_CLASSES`	`String`	`SCCS|perfmon|KSImpl.java|j2se`	Any class files to exclude from the MIDP tree
`MIDP_EXCLUDE_SRC`	`String`	“”	Any `.c` files to exclude from the MIDP tree
`MIDP_INCLUDE_CLASSES`	`String`	“”	Any `.class` files to include from MIDP tree
`MIDP_INCLUDE_SRC`	`String`	“”	Any `.c` files to include from the MIDP tree
`MIDP_SOURCE_FILES`	`String`	Any `.java` files found in the `$(MIDP_DIR)/src/` directory, excluding files that match any of the patterns listed in `$(MIDP_EXLUDE_CLASSES)`.	The `.java` files that are part of the `MIDP Reference Implementation`.

TABLE 16 Lists of Files *(Continued)*

Option	Type	Default Value on Windows 2000	Description
PLATFORM_DEF_SRC	String	dirent.c	Default source files specific to the platform
PLATFORM_EXCLUDE_CLASSES	String	SCCS	Any class files to exclude from the platform tree
PLATFORM_EXCLUDE_SRC	String	“”	Any platform-specific .c files to exclude when building an executable bundle
PLATFORM_INCLUDE_CLASSES	String	“”	Any platform-specific .class files to include when building an executable bundle
PLATFORM_INCLUDE_SRC	String	“”	Any platform-specific .c files to include when building an executable bundle
TEST_DEF_SOURCE	String	any .java files found in the $(MIDP_DIR)/tests directory, if the directory exists.	Default source files for testing
TEST_EXCLUDE_CLASSES	String	SCCS	Any class files to exclude from testing
TEST_INCLUDE_CLASSES	String	“”	Any .class files to include from the tests

TABLE 17 Directory Names

Option	Type	Default Value on Windows 2000	Description
`ALT_BOOTDIR`	`String`	Microsoft Windows 2000: `C:/JDK1.3`	Top-level directory of an alternate JDK to use to build the target You should set this environment variable when you installed MIDP. See *Installing MIDP* for more information.
`ARCH_DIR`	`String`	“”	Additional sub-directory to store object files This configuration option is useful when you build binaries for the same platform but two different CPU architectures, such as Solaris™/X86 and Solaris/SPARC. **Note:** This variable must start with a path separator.
`BINDIR`	`String`	`$(BUILD_DIR)/bin$(ARCH_DIR)`	Directory to store executable files, such as `MIDP_CMD`
`BUILD_DIR`	`String`	“.”	The directory from which the build was started
`CLASSBINDIR`	`String`	`$(BUILD_DIR)/classes`	
`KVM_DIR`	`String`	`$(MIDP_DIR)/../kvm`	Top-level directory of the KVM to use to build the target
`LIBDIR`	`String`	`$(BUILD_DIR)/lib`	
`MIDP_DIR`	`String`	*midpInstallDir*	Top-level directory of MIDP
`PLATFORM_CLASS_DIR`	`String`	`$(MIDP_DIR)/src/win32/classes`	Directory containing platform-specific classes This value must *not* have a trailing path separator, and its last directory *must* be called `classes`.
`PLATFORM_SRC_DIR`	`String`	`$(MIDP_DIR)/src/win32`	Directory containing platform-specific `.c` files
`SHARE_SRC_DIR`	`String`	`$(MIDP_DIR)/src/share`	
`STORAGEDIR`	`String`	`$(BUILD_DIR)/appdb`	The directory of the keystore that MIDP uses for secure communications

B.3 Target Functionality

The following table contains the configuration options that affect the functionality present in the build target. (This table does not include any options for debugging. Those are listed in the next section, "Debugging.")

TABLE 18 Target Capabilities

Option	Type	Default Value	Description
`ENABLE_SCREEN_CAPTURE`	`boolean`	`false`	Whether to include the code that enables capturing of the contents of the emulator's screen
`INCLUDE_I18N`	`boolean`	`true`	Whether the implementation supports internationalization **Note**: Setting this option to false reduces the build size by about 35K on Microsoft Windows 2000, and about 90K on UNIX®.
`PLATFORM`	`String`		Platform on which the executable will run
`RELEASE_HTTPS`	`boolean`	`true`	Whether to include HTTPS functionality in the target for release
`ROMIZING`	`boolean`	`true`	Whether to convert the class files in `classes.zip` to an object-file format for linking with the KVM
`SOUND_SUPPORTED`	`boolean`	`false`	Whether to include calls to augment button pushes with sound cues

B.4 Debugging

The following table contains configuration options that affect the amount of debugging information included in the build target.

TABLE 19 Debugging Capabilities

Option	Type	Default Value on Windows 2000	Description
`DEBUG`	`boolean`	`false`	Whether to create an object file with debugging symbols When this option is set to `true`, the MIDP executable is renamed to `midp_g`, and you can attach to the application with an appropriate C debugger. For example, you can use the `gdb` debugger if you have compiled with `gcc`.
`DEBUG_CFLAGS`	`string`	`-Zi -Od`	Additional flags for the C compiler if `DEBUG` is set to `true`
`DEBUG_COLLECTOR`	`boolean`	`false`	Whether to use a version of the garbage collector with debugging hooks (The value determines whether the build compiles the KVM file `collector.c` or `collectorDebug.c`.)
`ENABLE_DEBUGGER`	`boolean`	`false`	Whether to compile all Java class files with debugging information to allow source-level debugging When this option is set to `true`, you can connect to the application through the KVM debug proxy.
`ENABLEPROFILING`	`boolean`	`false`	Whether to have the MIDP Reference Implementation executable print a message on exit giving VM statistics, such as memory usage **Note**: This option has no effect unless the option `INCLUDEDEBUGCODE` is also `true`.
`ENABLE_MALLOC_ TRACE`	`boolean`	`false`	Whether to trace memory usage and print warnings for memory leaks or corruption This option is valid only when `USE_MIDP_MALLOC` is `true`.

TABLE 19 Debugging Capabilities *(Continued)*

Option	Type	Default Value on Windows 2000	Description
`INCLUDEDEBUGCODE`	`boolean`	`false`	Whether to include extra code to allow tracing of the VM internals If this option is set to `true`, the MIDP executable writes extra status text to the command window, such as stack traces when a method throws an exception. **Note**: This slows the VM and slightly increases its size.
`SRCPROFILING`	`boolean`	`false`	Whether to compile MIDP so that it is profiler enabled This option is valid only when `GCC` is `true`.
`USE_MIDP_MALLOC`	`boolean`	`false`	Use internal memory management for native heap allocations

APPENDIX C

Reference Documentation for `com.sun.midp.ssl`

This appendix reproduces the reference documentation for the `com.sun.ssl` package that is generated by the Javadoc™ tool.

Package com.sun.midp.ssl

Description

Provides a reference implementation of SSLv3.0 for MIDP. The code is modified version of KSSL (`http://sunlabs.eng/~vgupta/ssl/kssl/index.html`) from Sun Labs.

Supports abbreviated SSL handshakes.

Does not support SSL client authentication. This is because purpose of HTTPS for MIDP is to provide the wireless device with end-to-end confidentiality. The authentication must be done at the application level, which is what nearly all consumer applications do.

Only supports the SSL_RSA_WITH_RC4_128_SHA, SSL_RSA_WITH_RC4_128_MD5, and SSL_RSA_EXPORT_WITH_RC4_40_MD5 cipher suites. All known HTTPS servers support these suites.

Only Supports RSA X.509 certificates, these are the defacto standard. KSSL supports version 1, 2, and 3 X.509 certificates signed using RSA with either SHA-1, MD5 and MD2. For security reasons only version 3 certificates are allowed to be chained. While the key usage extension is supported, the name constraints extension is not.

Related Information

SSL for Java™ 2 Micro-Edition
(`http://sunlabs.eng/~vgupta/ssl/kssl/index.html`)

A Good Introduction to SSL
(`http://docs.iplanet.com/docs/manuals/security/sslin/index.htm`)

A Good Introduction to Public Key Cryptography
(`http://docs.iplanet.com/docs/manuals/security/pkin/index.htm`)

RFC 2246, The TLS Protocol Version 1.0
(`http://www.ietf.org/rfc/rfc2246.txt`)

SSL Protocol Version 3.0
(`http://home.netscape.com/eng/ssl3/draft302.txt`)

RFC 2459, Internet X.509 Public Key Infrastructure Certificate and CRL Profile
(`http://www.ietf.org/rfc/rfc2459.txt`)

RSA's PKCS (Public Key Cryptography Standards) document set.
(`http://www.rsasecurity.com/rsalabs/pkcs/`)

Since: MIDP 2.0

Class Summary

Interfaces

`CertStore`	This *interface* supports storage of certificates (not private keys or symmetric keys).
`Key`	Implements an abstract class that represents all keys (both symmetric and asymmetric).
`PrivateKey`	The PrivateKey abstract class is the base class for all private keys used in asymmetric algorithms.
`PublicKey`	The PublicKey abstract class is the base class for all public keys used in asymmetric algorithms.
`RSAPrivateKey`	Specifies the RSA private key interface.
`RSAPublicKey`	Specifies the RSA public key interface.

Class Summary

Classes

`KeyBuilder`	The KeyBuilder class is used to generate key objects for use in cryptographic operations.
`MessageDigest`	Implements an abstract class that generalizes all message digests.
`RandomData`	Implements an abstract class that generalizes random number generators.
`SecretKey`	Implements the base interface for keys used in symmetric algorithms.
`Signature`	Implements an abstract class that generalizes all signature algorithms.
`SSLStreamConnection`	The SSLStreamConnection class implements the StreamConnection interface.
`X509Certificate`	This class implements methods for creating X.509 certificates and accessing their attributes such as subject/issuer names, public keys and validity information.

Exceptions

`CryptoException`	Implements CryptoException which is thrown when a cryptographic method encounters an error.

com.sun.midp.ssl

CertStore

Declaration

`public interface` **`CertStore`**

Description

This *interface* supports storage of certificates (not private keys or symmetric keys).

Member Summary

Methods	
`X509Certificate[]`	`getCertificates(java.lang.String subjectName)` Returns the certificate(s) corresponding to a subject name string.

Methods

getCertificates(String)

```
public com.sun.midp.ssl.X509Certificate[]
              getCertificates(java.lang.String subjectName)
```

Returns the certificate(s) corresponding to a subject name string.

Parameters:

`subjectName` - subject name of the certificate in printable form.

Returns: corresponding certificates or null (if not found)

com.sun.midp.ssl

CryptoException

Declaration

```
public class CryptoException extends java.lang.Exception
```

```
java.lang.Object
  |
  +--java.lang.Throwable
        |
        +--java.lang.Exception
              |
              +--com.sun.midp.ssl.CryptoException
```

All Implemented Interfaces: `java.io.Serializable`

Description

Implements CryptoException which is thrown when a cryptographic method encounters an error.

Member Summary

Fields

`static short`	`ILLEGAL_ENCODING` Indicates that encrypted message was illegal encoded.
`static short`	`ILLEGAL_USE` Indicates illegal use of a method.
`static short`	`ILLEGAL_VALUE` Indicates a method was passed illegal parameter values.
`static short`	`INVALID_INIT` Indicates an object has not been properly initialized for the requested operation.
`static short`	`NO_SUCH_ALGORITHM` Indicates that a requested algorithm or key type is not supported.
`static short`	`UNINITIALIZED_KEY` Indicates that is key object has not been properly initialized.

Member Summary

Constructors

	`CryptoException(short reason)` Constructs a CryptoException with the specified reason code.

Methods

`static void`	`throwIt(short reason)` Throws a CryptoException with the specified reason code.
`java.lang.String`	`toString()` Returns a human readable string describing the CryptoException.

Inherited Member Summary

Methods inherited from class `Object`

```
equals(Object), getClass(), hashCode(), notify(), notifyAll(),
  wait(), wait(), wait()
```

Methods inherited from class `Throwable`

```
fillInStackTrace(), getLocalizedMessage(), getMessage(),
  printStackTrace(PrintWriter), printStackTrace(PrintWriter),
  printStackTrace(PrintWriter)
```

Fields

ILLEGAL_ENCODING

`public static final short` **ILLEGAL_ENCODING**

Indicates that encrypted message was illegal encoded.

ILLEGAL_USE

`public static final short` **ILLEGAL_USE**

Indicates illegal use of a method.

ILLEGAL_VALUE

`public static final short` **ILLEGAL_VALUE**

Indicates a method was passed illegal parameter values.

INVALID_INIT

`public static final short` **INVALID_INIT**

Indicates an object has not been properly initialized for the requested operation.

NO_SUCH_ALGORITHM

`public static final short` **NO_SUCH_ALGORITHM**

Indicates that a requested algorithm or key type is not supported.

UNINITIALIZED_KEY

`public static final short` **UNINITIALIZED_KEY**

Indicates that is key object has not been properly initialized.

Constructors

CryptoException(short)

`public` **CryptoException**`(short reason)`

Constructs a CryptoException with the specified reason code.

Parameters:
`reason` - reason code

Methods

throwIt(short)

```
public static void throwIt(short reason)
            throws CryptoException
```

Throws a CryptoException with the specified reason code.

Parameters:
`reason` - reason code

Throws:
`CryptoException` - with the specified reason code

toString()

```
public java.lang.String toString()
```

Returns a human readable string describing the CryptoException.

Overrides: `toString` in class `Throwable`

Returns: string representation of the CryptoException

com.sun.midp.ssl

Key

Declaration

`public interface **Key**`

All Known Subinterfaces: `PrivateKey`, `PublicKey`, `RSAPrivateKey`, `RSAPublicKey`

All Known Implementing Classes: `SecretKey`

Description

Implements an abstract class that represents all keys (both symmetric and asymmetric).

Member Summary

Methods

`void`	`clearKey()` Clears the key and sets it to an uninitialized state.
`short`	`getSize()` Returns the key size in number of bits.
`byte`	`getType()` Returns the key type.
`boolean`	`isInitialized()` Returns true if and only if a key has been initialized.
`java.lang.String`	`toString()` Converts a key object to its human readable string representation.

Methods

clearKey()

`public void` **`clearKey`**`()`

Clears the key and sets it to an uninitialized state.

getSize()

`public short` **`getSize`**`()`

Returns the key size in number of bits.

Returns: the key size in number of bits

getType()

`public byte` **`getType`**`()`

Returns the key type.

Returns: the key type

isInitialized()

`public boolean` **`isInitialized`**`()`

Returns true if and only if a key has been initialized. A key is not initialized until all associated set methods have been called at least once since the last time the key was in an uninitialized state.

Returns: true if the key has been initialized, false otherwise

toString()

`public java.lang.String` **`toString`**`()`

Converts a key object to its human readable string representation.

Overrides: `toString` in class `Object`

Returns: a human readable string representation of the key

com.sun.midp.ssl

KeyBuilder

Declaration

```
public class KeyBuilder

java.lang.Object
  |
  +--com.sun.midp.ssl.KeyBuilder
```

Description

The KeyBuilder class is used to generate key objects for use in cryptographic operations. It is a key object factory. This version of the implementation only supports symmetric keys and RSA keys with modulus lengths up to and including 1024-bits. DSA keys are not supported.

See Also: `Key`

Member Summary

Fields

`static byte`	`TYPE_ARCFOUR` Indicates an ARCfour key type.
`static byte`	`TYPE_RSA_PRIVATE` Indicates an RSA private key type.
`static byte`	`TYPE_RSA_PUBLIC` Indicates an RSA public key type.

Constructors

	`KeyBuilder()`

Methods

`static Key`	`buildKey(byte kType, short kLen, boolean kEnc)` Creates cryptographic keys of the specified type and length.

Inherited Member Summary

Methods inherited from class `Object`

`equals(Object), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(), wait()`

Fields

TYPE_ARCFOUR

`public static final byte` **TYPE_ARCFOUR**

Indicates an ARCfour key type.

TYPE_RSA_PRIVATE

`public static final byte` **TYPE_RSA_PRIVATE**

Indicates an RSA private key type.

TYPE_RSA_PUBLIC

`public static final byte` **TYPE_RSA_PUBLIC**

Indicates an RSA public key type.

Constructors

KeyBuilder()

```
public KeyBuilder()
```

Methods

buildKey(byte, short, boolean)

```
public static com.sun.midp.ssl.Key buildKey(byte kType,
                short kLen, boolean kEnc)
                throws CryptoException
```

Creates cryptographic keys of the specified type and length. For now, the third argument is ignored —- it is included here only for compatibility with the javacard.security.KeyBuilder class. Note that the object returned by this method must be cast to their appropriate key type interface.

Parameters:

`kType` - type of key to be generated

`kLen` - the desired size of the key in bits

`kEnc` - this parameter is ignored

Returns: a key with the specified attributes

Throws:

`CryptoException` - with the NO_SUCH_ALGORITHM reason code if the requested key type is not supported.

com.sun.midp.ssl

MessageDigest

Declaration

```
public abstract class MessageDigest

java.lang.Object
  |
  +--com.sun.midp.ssl.MessageDigest
```

Description

Implements an abstract class that generalizes all message digests. It is modelled after javacard.security.MessageDigest. This version of the implementation only supports ALG_MD5 (which produces a 16-byte hash as described in RFC 2313) and ALG_SHA (which produces a 20-byte hash using NIST's SHA1 algorithm). MD2, a predecessor to MD5, is not supported.

Member Summary

Fields	
`static byte`	`ALG_MD2` Indicates the MD2 message digest algorithm.
`static byte`	`ALG_MD5` Indicates the MD5 message digest algorithm.
`static byte`	`ALG_SHA` Indicates the SHA message digest algorithm.
Methods	
`abstract java.lang.Object`	`clone()` Clones the MessageDigest object.
`abstract short`	`doFinal(byte[] inBuf, int inOff, int inLen, byte[] outBuf, int outOff)` Generates a hash of all/last input data.
`abstract byte`	`getAlgorithm()` Gets the message digest algorithm.
`static MessageDigest`	`getInstance(byte alg, boolean ext)` Creates a MessageDigest object instance of the specified algorithm.

Member Summary

`abstract byte`	`getLength()` Gets the length (in bytes) of the hash.
`abstract void`	`reset()` Resets the MessageDigest to the initial state for further use.
`abstract void`	`update(byte[] inBuf, int inOff, int inLen)` Accumulates a hash of the input data.

Inherited Member Summary

Methods inherited from class `Object`

`equals(Object), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(), wait()`

Fields

ALG_MD2

`public static final byte` **ALG_MD2**

Indicates the MD2 message digest algorithm.

ALG_MD5

`public static final byte` **ALG_MD5**

Indicates the MD5 message digest algorithm.

ALG_SHA

`public static final byte` **ALG_SHA**

Indicates the SHA message digest algorithm.

Methods

clone()

```
public abstract java.lang.Object clone()
```

Clones the MessageDigest object.

Returns: a clone of this object

doFinal(byte[], int, int, byte[], int)

```
public abstract short doFinal(byte[] inBuf, int inOff, int inLen,
            byte[] outBuf, int outOff)
```

Generates a hash of all/last input data. Completes and returns the hash computation after performing final operations such as padding. The MessageDigest object is reset after this call.

Parameters:

`inBuf` - input buffer of data to be hashed

`inOff` - offset within inBuf where input data begins

`inLen` - length (in bytes) of data to be hashed

`outBuf` - output buffer where the hash should be placed

`outOff` - offset within outBuf where the resulting hash begins

Returns: number of bytes of hash left in outBuf

getAlgorithm()

```
public abstract byte getAlgorithm()
```

Gets the message digest algorithm.

Returns: algorithm implemented by this MessageDigest object

getInstance(byte, boolean)

```
public static com.sun.midp.ssl.MessageDigest getInstance(byte alg,
            boolean ext)
            throws CryptoException
```

Creates a MessageDigest object instance of the specified algorithm. The second parameter is ignored for now —- it is here only for compatibility with the javacard.security.MessageDigest class.

Parameters:
`alg` - the desired message digest algorithm, e.g. ALG_MD5

`ext` - ignored

Returns: a MessageDigest object instance of the requested algorithm

Throws:
`CryptoException` - with NO_SUCH_ALGORITHM reason code if the requested algorithm is not supported.

getLength()

```
public abstract byte getLength()
```

Gets the length (in bytes) of the hash.

Returns: byte-length of the hash produced by this object

reset()

```
public abstract void reset()
```

Resets the MessageDigest to the initial state for further use.

update(byte[], int, int)

```
public abstract void update(byte[] inBuf, int inOff, int inLen)
```

Accumulates a hash of the input data. This method is useful when the input data to be hashed is not available in one byte array.

Parameters:
`inBuf` - input buffer of data to be hashed

`inOff` - offset within inBuf where input data begins

`inLen` - length (in bytes) of data to be hashed

See Also: `doFinal(byte[], int, int, byte[], int)`

com.sun.midp.ssl

PrivateKey

Declaration

```
public interface PrivateKey extends Key
```

All Superinterfaces: `Key`

All Known Subinterfaces: `RSAPrivateKey`

Description

The PrivateKey abstract class is the base class for all private keys used in asymmetric algorithms.

Inherited Member Summary

Methods inherited from interface `Key`

`clearKey(), getSize(), getType(), isInitialized(), toString()`

com.sun.midp.ssl

PublicKey

Declaration

`public interface` **`PublicKey extends`** `Key`

All Superinterfaces: `Key`

All Known Subinterfaces: `RSAPublicKey`

Description

The PublicKey abstract class is the base class for all public keys used in asymmetric algorithms.

Inherited Member Summary

Methods inherited from interface `Key`

`clearKey(), getSize(), getType(), isInitialized(), toString()`

com.sun.midp.ssl

RandomData

Declaration

```
public abstract class RandomData
```

```
java.lang.Object
  |
  +--com.sun.midp.ssl.RandomData
```

Description

Implements an abstract class that generalizes random number generators.

Member Summary

Fields

`static byte`	`ALG_PSEUDO_RANDOM` Identifies a utility pseudo random number generation algorithm.
`static byte`	`ALG_SECURE_RANDOM` Identifies a cryptographically secure random number generation algorithm.

Methods

`abstract void`	`generateData(byte[] buf, short off, short len)` Generates random data.
`static RandomData`	`getInstance(byte alg)` Creates a RandomData instance of the selected algorithm.
`abstract void`	`setSeed(byte[] buf, short off, short len)` Seeds the random number generator.

Inherited Member Summary

Methods inherited from class `Object`

`equals(Object), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(), wait()`

Fields

ALG_PSEUDO_RANDOM

`public static final byte` **ALG_PSEUDO_RANDOM**

Identifies a utility pseudo random number generation algorithm.

ALG_SECURE_RANDOM

`public static final byte` **ALG_SECURE_RANDOM**

Identifies a cryptographically secure random number generation algorithm.

Methods

generateData(byte[], short, short)

`public abstract void` **generateData**`(byte[] buf, short off, short len)`

Generates random data.

Parameters:

`buf` - output buffer in which the random data is to be placed

`off` - starting offset within buf for the random data

`len` - number of bytes of random data to be placed in buf

getInstance(byte)

```
public static com.sun.midp.ssl.RandomData getInstance(byte alg)
            throws CryptoException
```

Creates a RandomData instance of the selected algorithm.

Parameters:

`alg` - the desired random number generation algorithm, e.g. ALG_PSEUDO_RANDOM

Returns: a RandomData instance implementing the selected algorithm.

Throws:

`CryptoException` - with reason code set to NO_SUCH_ALGORITHM if an unsupported algorithm is requested.

WARNING: Requests for a secure random number generator are currently redirected to a class that implements a weakly unpredictable source of random data. Licensees of this reference implementation are strongly urged to link requests for ALG_SECURE_RANDOM to better generators that may be available on their specific platforms.

setSeed(byte[], short, short)

```
public abstract void setSeed(byte[] buf, short off, short len)
```

Seeds the random number generator.

Parameters:

`buf` - input buffer containing the seed

`off` - offset within buf where the seed starts

`len` - number of bytes of seed data in buf

com.sun.midp.ssl

RSAPrivateKey

Declaration

`public interface` **`RSAPrivateKey extends`** `PrivateKey`

All Superinterfaces: `Key`, `PrivateKey`

Description

Specifies the RSA private key interface. An RSA key is not ready for us until both the modulus and exponent have been set.

Member Summary

Methods

`boolean`	`equals(RSAPrivateKey k)` Checks if this RSAPrivateKey has the same exponent and modulus as the specified key.
`short`	`getExponent(byte[] buf, short offset)` Returns the private exponent value of the key in the specified buffer.
`short`	`getModulus(byte[] buf, short offset)` Returns the modulus value of the key in the specified buffer.
`void`	`setExponent(byte[] buf, short offset, short len)` Sets the private exponent value of the key.
`void`	`setModulus(byte[] buf, short offset, short len)` Sets the modulus value of the key.

Inherited Member Summary

Methods inherited from interface `Key`

`clearKey(), getSize(), getType(), isInitialized(), toString()`

Methods

equals(RSAPrivateKey)

`public boolean` **equals**`(com.sun.midp.ssl.RSAPrivateKey k)`

Checks if this RSAPrivateKey has the same exponent and modulus as the specified key.

Parameters:
`k` - an RSAPrivateKey against which this key is to be compared

Returns: true if the two keys have the same exponent and modulus, false otherwise

getExponent(byte[], short)

`public short` **getExponent**`(byte[] buf, short offset)`

Returns the private exponent value of the key in the specified buffer. The data format is big-endian and right-aligned (the least significant bit is the least significant bit of the last byte).

Parameters:
`buf` - the output buffer where the exponent is placed

`offset` - offset within buf where the exponent starts

Returns: byte length of the exponent copied into buf

See Also: `setExponent(byte[], short, short)`

getModulus(byte[], short)

`public short` **getModulus**`(byte[] buf, short offset)`

Returns the modulus value of the key in the specified buffer. The data format is big-endian and right-aligned (the least significant bit is the least significant bit of the last byte).

Parameters:
`buf` - the output buffer where the modulus is placed

`offset` - offset within buf where the modulus starts

Returns: byte length of the modulus copied into buf

See Also: `setModulus(byte[], short, short)`

setExponent(byte[], short, short)

```
public void setExponent(byte[] buf, short offset, short len)
            throws CryptoException
```

Sets the private exponent value of the key. The data format is big-endian and right-aligned (the least significant bit is the least significant bit of the last byte).

Parameters:

`buf` - the input buffer containing the exponent

`offset` - offset within buf where the exponent starts

`len` - exponent length in bytes

Throws:

`CryptoException` - with reason code set to ILLEGAL_VALUE if the specified exponent data is inconsistent with the key size.

See Also: `getExponent(byte[], short)`

setModulus(byte[], short, short)

```
public void setModulus(byte[] buf, short offset, short len)
            throws CryptoException
```

Sets the modulus value of the key. The data format is big-endian and right-aligned (the least significant bit is the least significant bit of the last byte).

Parameters:

`buf` - the input buffer containing the modulus

`offset` - offset within buf where the modulus starts

`len` - modulus length in bytes

Throws:

`CryptoException` - with reason code set to ILLEGAL_VALUE if the specified modulus data is inconsistent with the key size.

See Also: `getModulus(byte[], short)`

com.sun.midp.ssl

RSAPublicKey

Declaration

`public interface` **`RSAPublicKey extends`** `PublicKey`

All Superinterfaces: `Key`, `PublicKey`

Description

Specifies the RSA public key interface. An RSA key is not ready for us until both the modulus and exponent have been set.

Member Summary

Methods

`boolean`	`equals(RSAPublicKey k)` Checks if this RSAPublicKey has the same exponent and modulus as the specified key.
`short`	`getExponent(byte[] buf, short offset)` Returns the public exponent value of the key in the specified buffer.
`short`	`getModulus(byte[] buf, short offset)` Returns the modulus value of the key in the specified buffer.
`void`	`setExponent(byte[] buf, short offset, short len)` Sets the public exponent value of the key.
`void`	`setModulus(byte[] buf, short offset, short len)` Sets the modulus value of the key.

Inherited Member Summary

Methods inherited from interface `Key`

`clearKey(), getSize(), getType(), isInitialized(), toString()`

Methods

equals(RSAPublicKey)

`public boolean equals(com.sun.midp.ssl.RSAPublicKey k)`

Checks if this RSAPublicKey has the same exponent and modulus as the specified key.

Parameters:
`k` - an RSAPublicKey against which this key is to be compared

Returns: true if the two keys have the same exponent and modulus, false otherwise

getExponent(byte[], short)

`public short getExponent(byte[] buf, short offset)`

Returns the public exponent value of the key in the specified buffer. The data format is big-endian and right-aligned (the least significant bit is the least significant bit of the last byte).

Parameters:
`buf` - the output buffer where the exponent is placed

`offset` - offset within buf where the exponent starts

Returns: byte length of the exponent copied into buf

See Also: `setExponent(byte[], short, short)`

getModulus(byte[], short)

`public short getModulus(byte[] buf, short offset)`

Returns the modulus value of the key in the specified buffer. The data format is big-endian and right-aligned (the least significant bit is the least significant bit of the last byte).

Parameters:
`buf` - the output buffer where the modulus is placed

`offset` - offset within buf where the modulus starts

Returns: byte length of the modulus copied into buf

See Also: `setModulus(byte[], short, short)`

setExponent(byte[], short, short)

```
public void setExponent(byte[] buf, short offset, short len)
            throws CryptoException
```

Sets the public exponent value of the key. The data format is big-endian and right-aligned (the least significant bit is the least significant bit of the last byte).

Parameters:

`buf` - the input buffer containing the exponent

`offset` - offset within buf where the exponent starts

`len` - exponent length in bytes

Throws:

`CryptoException` - with reason code set to ILLEGAL_VALUE if the specified exponent data is inconsistent with the key size.

See Also: `getExponent(byte[], short)`

setModulus(byte[], short, short)

```
public void setModulus(byte[] buf, short offset, short len)
            throws CryptoException
```

Sets the modulus value of the key. The data format is big-endian and right-aligned (the least significant bit is the least significant bit of the last byte).

Parameters:

`buf` - the input buffer containing the modulus

`offset` - offset within buf where the modulus starts

`len` - modulus length in bytes

Throws:

`CryptoException` - with reason code set to ILLEGAL_VALUE if the specified modulus data is inconsistent with the key size.

See Also: `getModulus(byte[], short)`

com.sun.midp.ssl

SecretKey

Declaration

```
public final class SecretKey implements Key
```

```
java.lang.Object
  |
  +--com.sun.midp.ssl.SecretKey
```

All Implemented Interfaces: `Key`

Description

Implements the base interface for keys used in symmetric algorithms.

Member Summary

Methods

`void`	`clearKey()` Clears the key and sets it to uninitialized state.
`byte`	`getKey(byte[] buf, short off)` Gets the key data.
`short`	`getSize()` Gets the key size in bits.
`byte`	`getType()` Gets the key type.
`boolean`	`isInitialized()` Checks if the key is initialized.
`void`	`setKey(byte[] buf, short off)` Sets the key data.
`java.lang.String`	`toString()` Converts the key to its corresponding human readable string representation.

Inherited Member Summary

Methods inherited from class `Object`

`equals(Object), getClass(), hashCode(), notify(), notifyAll(),`
`wait(), wait(), wait()`

Methods

clearKey()

`public void` **clearKey**`()`

Clears the key and sets it to uninitialized state.

Specified By: `clearKey` in interface `Key`

getKey(byte[], short)

`public byte` **getKey**`(byte[] buf, short off)`

Gets the key data. The data format is big-endian and right-aligned (the least significant bit is the least significant bit of last byte).

Parameters:

`buf` - the output buffer in which key data is to be placed

`off` - starting offset within buf for the key data.

Returns: number of bytes of key data placed in buf.

See Also: `setKey(byte[], short)`

getSize()

`public short` **getSize**`()`

Gets the key size in bits.

Specified By: `getSize` in interface `Key`

Returns: the key size in bits

getType()

```
public byte getType()
```

Gets the key type.

Specified By: `getType` in interface `Key`

Returns: the key type

isInitialized()

```
public boolean isInitialized()
```

Checks if the key is initialized.

Specified By: `isInitialized` in interface `Key`

Returns: true if the key has been initialized, false otherwise

setKey(byte[], short)

```
public void setKey(byte[] buf, short off)
            throws CryptoException
```

Sets the key data. The data format is big-endian and right-aligned (the least significant bit is the least significant bit of last byte).

Parameters:

`buf` - the input buffer containing key data

`off` - starting offset within buf for the key data.

Throws:

`CryptoException` - with reason code set to ILLEGAL_VALUE if the specified secret data is inconsistent with the key size.

See Also: `getKey(byte[], short)`

toString()

```
public java.lang.String toString()
```

Converts the key to its corresponding human readable string representation.

Specified By: `toString` in interface `Key`

Overrides: `toString` in class `Object`

Returns: a string representation of the secret key

com.sun.midp.ssl

Signature

Declaration

```
public abstract class Signature

java.lang.Object
  |
  +--com.sun.midp.ssl.Signature
```

Description

Implements an abstract class that generalizes all signature algorithms. This version of the implementation only supports ALG_RSA_MD5_PKCS1 and ALG_RSA_SHA_PKCS1.

Member Summary

Fields

`static byte`	`ALG_RSA_MD5_PKCS1` Signature algorithm ALG_RSA_MD5_PKCS1 encrypts the 16-byte MD5 digest using RSA.
`static byte`	`ALG_RSA_SHA_PKCS1` Signature algorithm ALG_RSA_SHA_PKCS1 encrypts the 20-byte SHA digest using RSA.
`static byte`	`MODE_SIGN` Used in init() methods to indicate signature sign mode.
`static byte`	`MODE_VERIFY` Used in init() methods to indicate signature verify mode.

Methods

`abstract byte`	`getAlgorithm()` Gets the signature algorithm.
`static Signature`	`getInstance(byte alg, boolean ext)` Creates a `Signature` object instance of the selected algorithm.
`abstract short`	`getLength()` Gets the byte length of the signature data.
`abstract void`	`init(Key theKey, byte theMode)` Initializes the `Signature` object with the appropriate `Key` for signature creation or verification.

Member Summary

`abstract void`	`init(Key theKey, byte theMode, byte[] b, int off, int len)` Initializes the `Signature` object with the appropriate `Key` and algorithm specific parameters for signature creation or verification.
`abstract short`	`sign(byte[] inBuf, int inOff, int inLen, byte[] sigBuf, int sigOff)` Generates the signature of all/last input data.
`abstract void`	`update(byte[] inBuf, int inOff, int inLen)` Accumulates a signature of the input data.
`abstract boolean`	`verify(byte[] inBuf, int inOff, int inLen, byte[] sigBuf, int sigOff, short sigLen)` Verifies the signature of all/last input data against the passed in signature.

Inherited Member Summary

Methods inherited from class `Object`

`equals(Object), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(), wait()`

Fields

ALG_RSA_MD5_PKCS1

`public static final byte` **ALG_RSA_MD5_PKCS1**

Signature algorithm ALG_RSA_MD5_PKCS1 encrypts the 16-byte MD5 digest using RSA. The digest is padded according to the PKCS#1 (v1.5) scheme.

ALG_RSA_SHA_PKCS1

`public static final byte` **ALG_RSA_SHA_PKCS1**

Signature algorithm ALG_RSA_SHA_PKCS1 encrypts the 20-byte SHA digest using RSA. The digest is padded according to the PKCS#1 (v1.5) scheme.

MODE_SIGN

`public static final byte` **MODE_SIGN**

Used in init() methods to indicate signature sign mode.

MODE_VERIFY

`public static final byte` **MODE_VERIFY**

Used in init() methods to indicate signature verify mode.

Methods

getAlgorithm()

`public abstract byte` **getAlgorithm**`()`

Gets the signature algorithm.

Returns: the algorithm code defined above

getInstance(byte, boolean)

```
public static com.sun.midp.ssl.Signature getInstance(byte alg,
              boolean ext)
              throws CryptoException
```

Creates a `Signature` object instance of the selected algorithm.

Parameters:

`alg` - desired signature algorithm

`ext` - this parameter is here only for compatibility with `javacard.security.Signature`, it is ignored

Returns: a `Signature` object instance of the requested algorithm

Throws:

`CryptoException` - with reason code `NO_SUCH_ALGORITHM` if the requested algorithm is not supported

getLength()

```
public abstract short getLength()
```

Gets the byte length of the signature data.

Returns: the byte length of signature data

init(Key, byte)

```
public abstract void init(com.sun.midp.ssl.Key theKey,
             byte theMode)
             throws CryptoException
```

Initializes the `Signature` object with the appropriate `Key` for signature creation or verification.

Parameters:

`theKey` - the key object to use for signing or verification

`theMode` - one of `MODE_SIGN` or `MODE_VERIFY`

Throws:

`CryptoException` - with reason code `ILLEGAL_VALUE` if an invalid mode is specified or if the key type is inconsistent with the mode or signature implementation.

init(Key, byte, byte[], int, int)

```
public abstract void init(com.sun.midp.ssl.Key theKey,
             byte theMode, byte[] b, int off, int len)
             throws CryptoException
```

Initializes the `Signature` object with the appropriate `Key` and algorithm specific parameters for signature creation or verification.

Parameters:

`theKey` - the key object to use for signing or verification

`theMode` - one of `MODE_SIGN` or `MODE_VERIFY`

`b` - byte array containing algorithm specific parameters

`off` - starting offset of parameter data within the byte array

`len` - byte length of parameter data

Throws:

`CryptoException` - with reason code `ILLEGAL_VALUE` if an invalid mode is specified or if the key type is inconsistent with the mode or signature implementation or if this initialization mode is not supported by the signature algorithm

sign(byte[], int, int, byte[], int)

```
public abstract short sign(byte[] inBuf, int inOff, int inLen,
            byte[] sigBuf, int sigOff)
            throws CryptoException
```

Generates the signature of all/last input data. A call to this method also resets this signature object to the state it was in when previously initialized via a call to init(). That is, the object is reset and available to sign another message.

Parameters:

`inBuf` - the input buffer of data to be signed

`inOff` - starting offset within the input buffer for data to be signed

`inLen` - the byte length of data to be signed

`sigBuf` - the output buffer to store signature data

`sigOff` - starting offset within the output buffer at which to begin signature data

Returns: number of bytes of signature output in sigBuf

Throws:

`CryptoException` - with the following reason codes: (i) `UNINITIALIZED_KEY` if key is not initialized, (ii) `INVALID_INIT` if signature object wasn not properly initialized, for signing (iii) `ILLEGAL_USE` if the signature algorithm does not pad the message and the message is not block aligned

update(byte[], int, int)

```
public abstract void update(byte[] inBuf, int inOff, int inLen)
            throws CryptoException
```

Accumulates a signature of the input data. When this method is used, temporary storage of intermediate results is required. This method should only be used if all the input data required for the signature is not available in one byte array. The sign() or verify() method is recommended whenever possible.

Parameters:

`inBuf` - the input buffer of data to be signed

`inOff` - starting offset within the input buffer for data to be signed

`inLen` - the byte length of data to be signed

Throws:

`CryptoException` - with `UNINITIALIZED_KEY` or `INVALID_INIT` if the signature object is not properly initialized

See Also: `sign(byte[], int, int, byte[], int)`, `verify(byte[], int, int, byte[], int, short)`

verify(byte[], int, int, byte[], int, short)

```
public abstract boolean verify(byte[] inBuf, int inOff, int inLen,
            byte[] sigBuf, int sigOff, short sigLen)
            throws CryptoException
```

Verifies the signature of all/last input data against the passed in signature. A call to this method also resets this signature object to the state it was in when previously initialized via a call to init(). That is, the object is reset and available to verify another message.

Parameters:

`inBuf` - the input buffer of data to be verified

`inOff` - starting offset within the input buffer for data to be verified

`inLen` - the byte length of data to be verified

`sigBuf` - the input buffer containing signature data

`sigOff` - starting offset within the sigBuf where signature data begins

`sigLen` - byte length of signature data

Returns: true if signature verifies, false otherwise

Throws:

`CryptoException` - with the following reason codes: (i) `UNINITIALIZED_KEY` if key is not initialized, (ii) `INVALID_INIT` if signature object wasn not properly initialized, for verification (iii) `ILLEGAL_USE` if the signature algorithm does not pad the message and the message is not block aligned

com.sun.midp.ssl

SSLStreamConnection

Declaration

```
public class SSLStreamConnection implements
    javax.microedition.io.StreamConnection
```

```
java.lang.Object
  |
  +--com.sun.midp.ssl.SSLStreamConnection
```

All Implemented Interfaces: javax.microedition.io.Connection, javax.microedition.io.InputConnection, javax.microedition.io.OutputConnection, javax.microedition.io.StreamConnection

Description

The SSLStreamConnection class implements the StreamConnection interface. Data exchanged through a SSLStreamConnection is automatically protected by SSL. Currently, only SSL version 3.0 is supported and the list of cipher suites proposed by the client is hard coded to {SSL_RSA_WITH_RC4_128_MD5, SSL_RSA_EXPORT_WITH_RC4_40_MD5}. This version of the implementation does not support client authentication at the SSL layer —- a feature that is rarely used. Typical usage of this class by an application would be along the following lines:

```
// create a TCP connection
StreamConnection t =
    Connector.open(“socket://www.server.com:443”);
// Create an SSL connection
SSLStreamConnection s =
    new SSLStreamConnection(“www.server.com”, 443,
           t.openInputStream(), t.openOutputStream());
t.close();

// obtain the associated input/output streams
OutputStream sout = s.openOutputStream();
InputStream sin = s.openInputStream();
...
// send SSL-protected data by writing to sout and
// receive SSL-protected by reading from sin
...
sin.close();
sout.close();
s.close();   // close the SSL connection when done
```

Member Summary

Constructors

	`SSLStreamConnection(java.lang.String host, int port, java.io.InputStream in, java.io.OutputStream out)` Establish and SSL session over a reliable stream.

Methods

`void`	`close()` Closes the SSL connection.
`javax.microedition.io.SecurityInfo`	`getSecurityInfo()` Returns the security information associated with this connection.
`X509Certificate`	`getServerCertificate()` Returns the server certificate associated with this connection.
`static CertStore`	`getTrustedCertStore()` Gets the certificate store containing trusted root certificates used to verify SSL server certificate chains.
`static void`	`lockTrustedCertStore()` Locks the current trusted certificate store so it cannot be changed.
`java.io.DataInputStream`	`openDataInputStream()` Returns the DataInputStream associated with this SSLStreamConnection.
`java.io.DataOutputStream`	`openDataOutputStream()` Returns the DataOutputStream associated with this SSLStreamConnection.
`java.io.InputStream`	`openInputStream()` Returns the InputStream associated with this SSLStreamConnection.
`java.io.OutputStream`	`openOutputStream()` Returns the OutputStream associated with this SSLStreamConnection.
`static void`	`setTrustedCertStore(CertStore cs)` Sets the trusted certificate store which contains root certificates used to verify certificate chains sent by an SSL server.

Inherited Member Summary

Methods inherited from class `Object`

```
equals(Object), getClass(), hashCode(), notify(), notifyAll(),
  toString(), wait(), wait(), wait()
```

Constructors

SSLStreamConnection(String, int, InputStream, OutputStream)

```
public SSLStreamConnection(java.lang.String host, int port,
            java.io.InputStream in, java.io.OutputStream out)
            throws IOException
```

Establish and SSL session over a reliable stream. This connection will forward the input and output stream close methods to the given connection. If the caller wants to have the given connection closed with this connection, the caller can close given connection after constructing this connection, but leaving the closing of the streams to this connection.

Parameters:

`host` - hostname of the SSL server

`port` - port number of the SSL server

`in` - InputStream associated with the StreamConnection

`out` - OutputStream associated with the StreamConnection

Throws:

`java.io.IOException` - if there is a problem initializing the SSL data structures or the SSL handshake fails

Methods

close()

```
public void close()
            throws IOException
```

Closes the SSL connection. The underlying TCP socket, over which SSL is layered, is also closed unless the latter was opened by an external application and its input/output streams were passed as argument to the SSLStreamConnection constructor.

Specified By: `close` in interface `Connection`

Throws:
`java.io.IOException` - if the SSL connection could not be terminated cleanly

getSecurityInfo()

```
public javax.microedition.io.SecurityInfo getSecurityInfo()
            throws IOException
```

Returns the security information associated with this connection.

Returns: the security information associated with this open connection

Throws:
`java.io.IOException` - if the connection is closed

getServerCertificate()

```
public com.sun.midp.ssl.X509Certificate getServerCertificate()
```

Returns the server certificate associated with this connection.

Returns: the server certificate associated with this connection

getTrustedCertStore()

```
public static com.sun.midp.ssl.CertStore getTrustedCertStore()
```

Gets the certificate store containing trusted root certificates used to verify SSL server certificate chains.

Returns: certificate store containing trusted certificates

See Also: `setTrustedCertStore(CertStore)`

lockTrustedCertStore()

```
public static void lockTrustedCertStore()
```

Locks the current trusted certificate store so it cannot be changed. This method does nothing if the trusted certificate store is null.

openDataInputStream()

```
public java.io.DataInputStream openDataInputStream()
            throws IOException
```

Returns the DataInputStream associated with this SSLStreamConnection.

Specified By: `openDataInputStream` in interface `InputConnection`

Returns: a DataInputStream object

Throws:
`java.io.IOException` - if the connection is not open or the stream was already open

openDataOutputStream()

```
public java.io.DataOutputStream openDataOutputStream()
            throws IOException
```

Returns the DataOutputStream associated with this SSLStreamConnection.

Specified By: `openDataOutputStream` in interface `OutputConnection`

Returns: a DataOutputStream object

Throws:
`java.io.IOException` - if the connection is not open or the stream was already open

openInputStream()

```
public java.io.InputStream openInputStream()
            throws IOException
```

Returns the InputStream associated with this SSLStreamConnection.

Specified By: `openInputStream` in interface `InputConnection`

Returns: InputStream object from which SSL protected bytes can be read

Throws:
`java.io.IOException` - if the connection is not open or the stream was already open

openOutputStream()

```
public java.io.OutputStream openOutputStream()
             throws IOException
```

Returns the OutputStream associated with this SSLStreamConnection.

Specified By: `openOutputStream` in interface `OutputConnection`

Returns: OutputStream object such that bytes written to this stream are sent over an SSL secured channel

Throws:
`java.io.IOException` - if the connection is not open or the stream was already open

setTrustedCertStore(CertStore)

```
public static void setTrustedCertStore(com.sun.midp.ssl.CertStore
             cs)
```

Sets the trusted certificate store which contains root certificates used to verify certificate chains sent by an SSL server. This method does nothing if the certificate store is already locked.

Parameters:
`cs` - trusted certificate store to be used

See Also: `lockTrustedCertStore()`, `getTrustedCertStore()`

com.sun.midp.ssl

X509Certificate

Declaration

```
public class X509Certificate implements
   javax.microedition.pki.Certificate
```

```
java.lang.Object
  |
  +--com.sun.midp.ssl.X509Certificate
```

All Implemented Interfaces: `javax.microedition.pki.Certificate`

Description

This class implements methods for creating X.509 certificates and accessing their attributes such as subject/issuer names, public keys and validity information. Publicly visible methods are modeled after those in the X509Certificate classes from J2SE™ (Java 2 Platform, Standard Edition) but there are some differences and these are documented below.

NOTE: For now, only X.509 certificates containing RSA public keys and signed either using md5WithRSA or sha-1WithRSA are supported. This version of the implementation is unable to parse certificates containing DSA keys or signed using DSA. Certificates containing RSA keys but signed using an unsupported algorithm (e.g. RSA_MD2) can be parsed but cannot be verified. Not all version 3 extensions are supported (only subjectAltName, basicConstraints, keyUsage and extendedKeyUsage are recognized) but if an unrecognized extension is marked critical, an error notification is generated.

Member Summary

Fields

`static int`	`CERT_SIGN_KEY_USAGE` Bit mask for key certificate sign key usage.
`static int`	`CLIENT_AUTH_EXT_KEY_USAGE` Bit mask client auth for extended key usage.
`static int`	`CODE_SIGN_EXT_KEY_USAGE` Bit code signing mask for extended key usage.
`static int`	`CRL_SIGN_KEY_USAGE` Bit mask for CRL sign key usage.
`static int`	`DATA_ENCIPHER_KEY_USAGE` Bit mask for data encipherment key usage.

Member Summary

static int	DECIPHER_ONLY_KEY_USAGE Bit mask for decipher only key usage.
static int	DIGITAL_SIG_KEY_USAGE Bit mask for digital signature key usage.
static int	EMAIL_EXT_KEY_USAGE Bit email protection mask for extended key usage.
static int	ENCIPHER_ONLY_KEY_USAGE Bit mask for encipher only key usage.
static int	IPSEC_END_SYS_EXT_KEY_USAGE Bit IPSEC end system mask for extended key usage.
static int	IPSEC_TUNNEL_EXT_KEY_USAGE Bit IPSEC tunnel mask for extended key usage.
static int	IPSEC_USER_EXT_KEY_USAGE Bit IPSEC user mask for extended key usage.
static int	KEY_AGREEMENT_KEY_USAGE Bit mask for key agreement key usage.
static int	KEY_ENCIPHER_KEY_USAGE Bit mask for key encipherment key usage.
static int	MISSING_PATH_LENGTH_CONSTRAINT Indicates that no information is available on the pathLengthConstraint associated with this certificate (this could happen if the certifiate is a v1 or v2 cert or a v3 cert without basicConstraints or a non-CA v3 certificate).
static byte	NO_ERROR Indicates a no error condition.
static int	NON_REPUDIATION_KEY_USAGE Bit mask for non repudiation key usage.
static int	SERVER_AUTH_EXT_KEY_USAGE Bit mask server auth for extended key usage.
static int	TIME_STAMP_EXT_KEY_USAGE Bit time stamping mask for extended key usage.
static byte	TYPE_DNS_NAME DNS name alternative name type code.
static byte	TYPE_EMAIL_ADDRESS Email address (rfc 822) alternative name type code.
static byte	TYPE_URI URI alternative name type code.
static int	UNLIMITED_CERT_CHAIN_LENGTH Indicates there is no limit to the server certificate chain length.

Constructors

Member Summary

	`X509Certificate(byte ver, byte[] rawSerialNumber, java.lang.String sub, java.lang.String iss, long notBefore, long notAfter, byte[] mod, byte[] exp, byte[] chash, int pLen)` Creates an X.509 certificate with the specified attributes.
Methods	
`void`	`checkExtensions()` Checks if a certificate has any (version 3) extensions that were not properly processed and continued use of this certificate may be inconsistent with the issuer's intent.
`void`	`checkValidity()` Checks if the certificate is currently valid.
`void`	`checkValidity(long time)` Checks if the certificate is valid on the specified time.
`static X509Certificate`	`generateCertificate(byte[] buf, int off, int len)` Creates a certificate by parsing the ASN.1 DER X.509 certificate encoding in the specified buffer. **NOTE:** In the standard edition, equivalent functionality is provided by CertificateFactory.generateCertificate(InputStream).
`int`	`getBasicConstraints()` Gets the certificate constraints path length from the `BasicConstraints` extension.
`int`	`getExtKeyUsage()` Gets a 32-bit bit vector (in the form of an integer) in which each position represents a purpose for which the public key in the certificate may be used (iff that bit is set).
`byte[]`	`getFingerprint()` Gets the MD5 fingerprint of this certificate. **NOTE:** this implementation returns a byte array filled with zeros if there is no fingerprint associated with this certificate.
`java.lang.String`	`getIssuer()` Gets the name of this certificate's issuer.
`int`	`getKeyUsage()` Gets a 32-bit bit vector (in the form of an integer) in which each position represents a purpose for which the public key in the certificate may be used (iff that bit is set).
`long`	`getNotAfter()` Gets the NotAfter date from the certificate's validity period.

Member Summary

`long`	`getNotBefore()` Gets the NotBefore date from the certificate's validity period.
`PublicKey`	`getPublicKey()` Gets the public key from this certificate.
`java.lang.String`	`getSerialNumber()` Gets the printable form of the serial number of this `Certificate`.
`java.lang.String`	`getSigAlgName()` Gets the name of the algorithm used to sign the certificate.
`java.lang.String`	`getSubject()` Gets the name of this certificate's subject.
`java.lang.Object`	`getSubjectAltName()` Gets the subject alternative name or null if it was not in the certificate.
`int`	`getSubjectAltNameType()` Gets the type of subject alternative name.
`java.lang.String`	`getType()` Get the type of the `Certificate`.
`java.lang.String`	`getVersion()` Gets the raw X.509 version number of this certificate.
`java.lang.String`	`toString()` Returns a string representation of this certificate.
`void`	`verify(PublicKey pk)` Checks if this certificate was signed using the private key corresponding to the specified public key.
`static X509Certificate`	`verifyChain(java.util.Vector certs, int keyUsage, int extKeyUsage, CertStore certStore)` Verify a chain of certificates.

Inherited Member Summary

Methods inherited from class `Object`

`equals(Object), getClass(), hashCode(), notify(), notifyAll(), wait(), wait(), wait()`

Fields

CERT_SIGN_KEY_USAGE

`public static final int` **CERT_SIGN_KEY_USAGE**

Bit mask for key certificate sign key usage.

CLIENT_AUTH_EXT_KEY_USAGE

`public static final int` **CLIENT_AUTH_EXT_KEY_USAGE**

Bit mask client auth for extended key usage.

CODE_SIGN_EXT_KEY_USAGE

`public static final int` **CODE_SIGN_EXT_KEY_USAGE**

Bit code signing mask for extended key usage.

CRL_SIGN_KEY_USAGE

`public static final int` **CRL_SIGN_KEY_USAGE**

Bit mask for CRL sign key usage.

DATA_ENCIPHER_KEY_USAGE

`public static final int` **DATA_ENCIPHER_KEY_USAGE**

Bit mask for data encipherment key usage.

DECIPHER_ONLY_KEY_USAGE

`public static final int` **DECIPHER_ONLY_KEY_USAGE**

Bit mask for decipher only key usage.

DIGITAL_SIG_KEY_USAGE

`public static final int` **DIGITAL_SIG_KEY_USAGE**

Bit mask for digital signature key usage.

EMAIL_EXT_KEY_USAGE

`public static final int` **EMAIL_EXT_KEY_USAGE**

Bit email protection mask for extended key usage.

ENCIPHER_ONLY_KEY_USAGE

`public static final int` **ENCIPHER_ONLY_KEY_USAGE**

Bit mask for encipher only key usage.

IPSEC_END_SYS_EXT_KEY_USAGE

`public static final int` **IPSEC_END_SYS_EXT_KEY_USAGE**

Bit IPSEC end system mask for extended key usage.

IPSEC_TUNNEL_EXT_KEY_USAGE

`public static final int` **IPSEC_TUNNEL_EXT_KEY_USAGE**

Bit IPSEC tunnel mask for extended key usage.

IPSEC_USER_EXT_KEY_USAGE

`public static final int` **IPSEC_USER_EXT_KEY_USAGE**

Bit IPSEC user mask for extended key usage.

KEY_AGREEMENT_KEY_USAGE

`public static final int` **KEY_AGREEMENT_KEY_USAGE**

Bit mask for key agreement key usage.

KEY_ENCIPHER_KEY_USAGE

`public static final int` **KEY_ENCIPHER_KEY_USAGE**

Bit mask for key encipherment key usage.

MISSING_PATH_LENGTH_CONSTRAINT

`public static final int` **MISSING_PATH_LENGTH_CONSTRAINT**

Indicates that no information is available on the pathLengthConstraint associated with this certificate (this could happen if the certifiate is a v1 or v2 cert or a v3 cert without basicConstraints or a non-CA v3 certificate).

NO_ERROR

`public static final byte` **NO_ERROR**

Indicates a no error condition.

NON_REPUDIATION_KEY_USAGE

`public static final int` **NON_REPUDIATION_KEY_USAGE**

Bit mask for non repudiation key usage.

SERVER_AUTH_EXT_KEY_USAGE

`public static final int` **SERVER_AUTH_EXT_KEY_USAGE**

Bit mask server auth for extended key usage.

TIME_STAMP_EXT_KEY_USAGE

`public static final int` **TIME_STAMP_EXT_KEY_USAGE**

Bit time stamping mask for extended key usage.

TYPE_DNS_NAME

`public static final byte` **TYPE_DNS_NAME**

DNS name alternative name type code.

TYPE_EMAIL_ADDRESS

`public static final byte` **TYPE_EMAIL_ADDRESS**

Email address (rfc 822) alternative name type code.

TYPE_URI

`public static final byte` **TYPE_URI**

URI alternative name type code.

UNLIMITED_CERT_CHAIN_LENGTH

`public static final int` **UNLIMITED_CERT_CHAIN_LENGTH**

Indicates there is no limit to the server certificate chain length.

Constructors

X509Certificate(byte, byte[], String, String, long, long, byte[], byte[], byte[], int)

```
public X509Certificate(byte ver, byte[] rawSerialNumber,
            java.lang.String sub, java.lang.String iss,
            long notBefore, long notAfter, byte[] mod, byte[] exp,
            byte[] chash, int pLen)
            throws Exception
```

Creates an X.509 certificate with the specified attributes. This constructor is only used for creating trusted certificates.

NOTE: All signature related values in these certificates (such as the signing algorithm and signature) are set to null and invoking methods that access signature information, e.g. verify() and getSigAlgName() can produce unexpected errors.

Parameters:

`ver` - byte containing X.509 version

`rawSerialNumber` - byte array containing the serial number

`sub` - subject name

`iss` - issuer name

`notBefore` - start of validity period expressed in milliseconds since midnight Jan 1, 1970 UTC

`notAfter` - end of validity period expressed as above

`mod` - modulus associated with the RSA Public Key

`exp` - exponent associated with the RSA Public Key

`chash` - 16-byte MD5 hash of the certificate's ASN.1 DER encoding

`pLen` - Is the pathLenConstraint associated with a version 3 certificate. This parameter is ignored for v1 and v2 certificates. If a v3 certificate does not have basicConstraints or is not a CA cert, callers should pass MISSING_PATH_LENGTH_CONSTRAINT. If the v3 certificate has basicConstraints, CA is set but pathLenConstraint is missing (indicating no limit on the certificate chain), callers should pass UNLIMITED_CERT_CHAIN_LENGTH.

Throws:

`java.lang.Exception` - in case of a problem with RSA public key parameters

Methods

checkExtensions()

```
public void checkExtensions()
            throws CertificateException
```

Checks if a certificate has any (version 3) extensions that were not properly processed and continued use of this certificate may be inconsistent with the issuer's intent. This may happen, for example, if the certificate has unrecognized critical extensions.

Throws:
`javax.microedition.pki.CertificateException` - with a reason ofr BAD_EXTENSIONS if there are any bad extensions

checkValidity()

```
public void checkValidity()
            throws CertificateException
```

Checks if the certificate is currently valid. It is if the current date and time are within the certificate's validity period.

Throws:
`javax.microedition.pki.CertificateException` - with a reason of EXPIRED or NOT_YET_VALID

checkValidity(long)

```
public void checkValidity(long time)
            throws CertificateException
```

Checks if the certificate is valid on the specified time. It is if the specified time is within the certificate's validity period.

NOTE: The standard edition provides a method with this name but it throws different types of exceptions rather than returning error codes.

Parameters:
`time` - the time in milliseconds for which a certificate's validity is to be checked

Throws:
`javax.microedition.pki.CertificateException` - with a reason of EXPIRED or NOT_YET_VALID

generateCertificate(byte[], int, int)

```
public static com.sun.midp.ssl.X509Certificate
            generateCertificate(byte[] buf, int off, int len)
            throws IOException
```

Creates a certificate by parsing the ASN.1 DER X.509 certificate encoding in the specified buffer.

NOTE: In the standard edition, equivalent functionality is provided by CertificateFactory.generateCertificate(InputStream).

Parameters:

`buf` - byte array to be read

`off` - offset within the byte array

`len` - number of bytes to be read

Returns: a certificate object corresponding to the DER encoding or null (in case of an encoding problem)

Throws:

`java.io.IOException` - if there is a parsing error

getBasicConstraints()

```
public int getBasicConstraints()
```

Gets the certificate constraints path length from the `BasicConstraints` extension.

The `BasicConstraints` extension identifies whether the subject of the certificate is a Certificate Authority (CA) and how deep a certification path may exist through the CA. The `pathLenConstraint` field (see below) is meaningful only if `cA` is set to TRUE. In this case, it gives the maximum number of CA certificates that may follow this certificate in a certification path. A value of zero indicates that only an end-entity certificate may follow in the path.

Note that for RFC 2459 this extension is always marked critical if `cA` is TRUE, meaning this certificate belongs to a Certificate Authority.

The ASN.1 definition for this is:

```
BasicConstraints ::= SEQUENCE {
      cA                  BOOLEAN DEFAULT FALSE,
      pathLenConstraint   INTEGER (0..MAX) OPTIONAL
}
```

Returns: MISSING_PATH_LENGTH_CONSTRAINT if the `BasicConstraints` extension is absent or the subject of the certificate is not a CA. If the subject of the certificate is a CA and `pathLenConstraint` does not appear, `UNLIMITED_CERT_CHAIN_LENGTH` is returned to indicate that there is no limit to the allowed length of the certification path. In all other situations, the actual value of the `pathLenConstraint` is returned.

getExtKeyUsage()

`public int` **`getExtKeyUsage`**`()`

Gets a 32-bit bit vector (in the form of an integer) in which each position represents a purpose for which the public key in the certificate may be used (iff that bit is set). The correspondence between bit positions and purposes is as follows:

serverAuth	1
clientAuth	2
codeSigning	3
emailProtection	4
ipsecEndSystem	5
ipsecTunnel	6
ipsecUser	7
timeStamping	8

Returns: a bitvector indicating extended usage of the certificate public key, -1 if a critical extendedKeyUsage extension is not present.

getFingerprint()

`public byte[]` **`getFingerprint`**`()`

Gets the MD5 fingerprint of this certificate.

NOTE: this implementation returns a byte array filled with zeros if there is no fingerprint associated with this certificate. This may happen if a null was passed to the X509Certificate constructor.

Returns: a byte array containing this certificate's MD5 hash

getIssuer()

`public java.lang.String` **`getIssuer`**`()`

Gets the name of this certificate's issuer.

NOTE: The corresponding method in the standard edition is getIssuerDN() and returns a Principal.

Specified By: `getIssuer` in interface `Certificate`

Returns: a string containing this certificate's issuer in user-friendly form

getKeyUsage()

`public int` **getKeyUsage**`()`

Gets a 32-bit bit vector (in the form of an integer) in which each position represents a purpose for which the public key in the certificate may be used (iff that bit is set). The correspondence between bit positions and purposes is as follows:

digitalSignature	0
nonRepudiation	1
keyEncipherment	2
dataEncipherment	3
keyAgreement	4
keyCertSign	5
cRLSign	6
encipherOnly	7
decipherOnly	8

Returns: a bitvector indicating approved key usage of the certificate public key, -1 if a KeyUsage extension is not present.

getNotAfter()

`public long` **getNotAfter**`()`

Gets the NotAfter date from the certificate's validity period.

Specified By: `getNotAfter` in interface `Certificate`

Returns: a date after which the certificate is not valid (expiration date)

getNotBefore()

`public long` **getNotBefore**`()`

Gets the NotBefore date from the certificate's validity period.

Specified By: `getNotBefore` in interface `Certificate`

Returns: a date before which the certificate is not valid

getPublicKey()

`public com.sun.midp.ssl.PublicKey` **getPublicKey**`()`
`            throws CertificateException`

Gets the public key from this certificate.

Returns: the public key contained in the certificate

Throws:
`javax.microedition.pki.CertificateException` - if public key is not a supported type (could not be parsed).

getSerialNumber()

`public java.lang.String` **getSerialNumber**`()`

Gets the printable form of the serial number of this `Certificate`. If the serial number within the `certificate` is binary is should be formatted as a string using hexadecimal notation with each byte represented as two hex digits separated byte ":" (Unicode x3A). For example, 27:56:FA:80.

Specified By: `getSerialNumber` in interface `Certificate`

Returns: A string containing the serial number in user-friendly form; `NULL` is returned if there is no serial number.

getSigAlgName()

`public java.lang.String` **getSigAlgName**`()`

Gets the name of the algorithm used to sign the certificate.

Specified By: `getSigAlgName` in interface `Certificate`

Returns: the name of signature algorithm

getSubject()

`public java.lang.String` **getSubject**`()`

Gets the name of this certificate's subject.

NOTE: The corresponding method in the standard edition is getSubjectDN() and returns a Principal.

Specified By: `getSubject` in interface `Certificate`

Returns: a string containing this certificate's subject in user-friendly form

getSubjectAltName()

`public java.lang.Object` **getSubjectAltName**`()`

Gets the subject alternative name or null if it was not in the certificate.

Returns: type of subject alternative name or null

getSubjectAltNameType()

`public int` **getSubjectAltNameType**`()`

Gets the type of subject alternative name.

Returns: type of subject alternative name

getType()

`public java.lang.String` **getType**`()`

Get the type of the `Certificate`.

Specified By: `getType` in interface `Certificate`

Returns: The type of the `Certificate`; the value MUST NOT be `NULL`.

getVersion()

`public java.lang.String` **getVersion**`()`

Gets the raw X.509 version number of this certificate. Version 1 is 0.

Specified By: `getVersion` in interface `Certificate`

Returns: the X.509 logic version number (1, 2, 3) of the certificate

toString()

`public java.lang.String` **toString**`()`

Returns a string representation of this certificate.

Overrides: `toString` in class `Object`

Returns: a human readable string representation of this certificate

verify(PublicKey)

```
public void verify(com.sun.midp.ssl.PublicKey pk)
            throws CertificateException
```

Checks if this certificate was signed using the private key corresponding to the specified public key.

Parameters:
`pk` - public key to be used for verifying certificate signature

Throws:
`javax.microedition.pki.CertificateException` - if there is an error

verifyChain(Vector, int, int, CertStore)

```
public static com.sun.midp.ssl.X509Certificate
            verifyChain(java.util.Vector certs, int keyUsage,
            int extKeyUsage,
            com.sun.midp.ssl.CertStore certStore)
            throws CertificateException
```

Verify a chain of certificates.

Parameters:
`certs` - list of certificates with first being entity certificate and the last being the CA issued certificate.

`keyUsage` - -1 to not check the key usage extension, or a key usage bit mask to check for if the extension is present

`extKeyUsage` - -1 to not check the extended key usage extension, or a extended key usage bit mask to check for if the extension is present

`certStore` - store of trusted CA certificates

Returns: first certificate signed by a CA in the given certificate store

Throws:
`javax.microedition.pki.CertificateException` - if there is an error verifying the chain

Index