Java™ 2 Platform, Micro Edition Mobile Information Device Profile Reference Implementation Porting Guide

---

MIDP 1.0 Specification

MIDP Reference Implementation 1.0.3

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303 USA
650 960-1300 fax 650 969-9131

MIDP RI 1.0.3
September 17, 2001

901 San Antonio Road, Palo Alto, CA 94303 USA

Contents

Tables

About This Document

This document provides information for implementing the Mobile Information Device Profile (*MIDP 1.0 Specification*), on a particular mobile device. It also provides information on porting version 1.0.3 of the Sun Microsystems, Inc. reference implementation (RI) of the MIDP to a device.

Note – The RI supports the *MIDP 1.0 Specification*, which can be downloaded from: `http://java.sun.com/products/midp/#finalspec`

Who Should Use This Document

This document is intended primarily for those individuals and companies who want to implement the MIDP or to port the MIDP RI from to a new platform.

How This Document Is Organized

The topics in this document are organized as follows:

Chapter 1, "Introduction to MIDP," describes the relationship of MIDP to Java™ 2 Micro Edition™ and its configurations. (See also the J2ME web page at `http://java.sun.com/j2me`.)

Chapter 2, "MIDP Porting Considerations," discusses the high-level issues involved with porting MIDP and the possible degrees of porting.

Chapter 3, "Porting the GUI," introduces porting the GUI portion of the RI, and discusses useful porting techniques.

Chapter 4, "Porting the Storage," introduces porting the storage API, which supports MIDP's RMS and other parts of the RI.

Chapter 5, "Porting the Application Management System," introduces the AMS and gives a summary of porting it.

Chapter 6, "Porting the Networking Implementation," introduces porting the TCP/IP, HTTP and HTTPS portions of the MIDP RI, and explains how to expose the CLDC networking code.

Chapter 7, "Building a Different Executable," gives advice that will help you build smaller versions of the `midp` executable.

Chapter 8, "Thread-Safety," gives advice on making a MIDP port thread-safe.

Chapter 9, "Porting Example," gives an example of re-implementing a native method of the RI.

Chapter 10, "Compiler Requirements," summarizes the C compiler requirements to build an implementation of MIDP.

New Features in this Release

The 1.0.3 release of the MIDP RI features several key enhancements over prior releases.

- Support for OTA (over-the-air) provisioning.
- Graphical Application Management Software (AMS) included.
- Optional HTTPS support, including KSSL (subject to export controls.) (KSSL is a small-footprint Secure Socket Layer (SSL) implementation for J2ME devices.)
- Keyboard entry enhancements.

OTA Provisioning

OTA Provisioning was released as a recommended practice document to help wireless network operators deploy applications in a consistent way. It is written as an addendum to the *MIDP 1.0 Specification*. It involves extra properties in the application descriptor file (`.jad`) that are used when a new application is installed or removed. The addendum is available at:

`http://java.sun.com/products/midp/OTAProvisioning-1.0.pdf`

AMS

This facility is MIDP's on-device mechanism for installing and removing MIDlet suites. In past releases, AMS was sometimes referred to as JAM (Java Application Manager), but this is misleading because JAM properly refers to a feature of the *CLDC Specification*.

For more information on the MIDP RI's application management facility, refer to **Chapter 5, "Porting the Application Management System**."

HTTPS Support

Beginning with the MIDP 1.0.3 RI, HTTPS support is provided. This involves the (optional) inclusion of KSSL for low-level handling of SSL stream connections, including a minimal key store for persistent storage of certificates. (The exact details of this are implementation-specific).

The CLDC Generic Connection Framework (GCF) support for resources addressed as `https://` does not require any new APIs beyond what is already supported in the *MIDP 1.0 Specification* for HTTP connections.

JDK 1.3 (or later) is required for HTTPS support, because its security features are needed for the offline tool used to extract runtime keys from certificates. The MIDP runtime is based on the CLDC 1.0 KVM.

Related Documentation

The Java™ Language Specification (Java Series), Second Edition by James Gosling, Bill Joy, Guy Steele and Gilad Bracha. Addison-Wesley, 2000, ISBN 0-201-31008-2, `http://java.sun.com/docs/books/jls/index.html`

The Java™ Virtual Machine Specification (Java Series), Second Edition by Tim Lindholm and Frank Yellin. Addison-Wesley, 1999, ISBN 0-201-43294-3, `http://java.sun.com/docs/books/vmspec/index.html`

KVM Porting Guide, Sun Microsystems, Inc., available as part of the CLDC download package at `http://www.sun.com/software/communitysource/j2me/cldc/download.html`

Connected, Limited Device Configuration Specification, version 1.0, Java Community Process, Sun Microsystems, Inc.
`http://jcp.org/jsr/detail/30.jsp`

Mobile Information Device Profile Specification, version 1.0a, Java Community Process, Sun Microsystems, Inc.
`http://jcp.org/jsr/detail/37.jsp`

Java 2 Platform Micro Edition (J2ME™) Technology for Creating Mobile Devices, A White Paper, Sun Microsystems, Inc.,
`http://java.sun.com/products/cldc/wp/KVMwp.pdf`

KVM Debug Wire Protocol (KDWP) Specification, Sun Microsystems, Inc., available as part of the CLDC download package at
`http://www.sun.com/software/communitysource/j2me/cldc/download.html`

Next Generation Expert Groups

J2ME technologies are under a process of evolution within the Java Community Process. JSRs (Java Specification Requests) are already filed for CDLC and MIDP. You can view the JSRs and follow the stages of the next-generation specifications by visiting the web sites:

CLDC Next Generation,
`http://jcp.org/jsr/detail/139.jsp`

MIDP Next Generation,
`http://jcp.org/jsr/detail/118.jsp`

All the pending JSRs for J2ME can be viewed at
`http://jcp.org/jsr/ec/me.jsp`

CHAPTER 1

Introduction to MIDP

1.1 Mobile Information Device Profile (MIDP)

MIDP (also known as the *Mobile Information Device Profile*) is a profile of Java™ 2 Micro Edition™ intended primarily for small, resource-constrained devices such as cellular phones, pagers, personal organizers, mobile Internet devices, and so forth. It is intended to work with the CLDC configuration of Java 2 Micro Edition.

A Reference Implementation of MIDP is provided by Sun Microsystems, Inc. It is implemented in two parts: a high-level part implemented in the Java programming language, and the native (or low-level) part implemented in the C programming language. The MIDP Reference Implementation runs as an emulator on the Windows 2000™ platform.

Note – In the remainder of this document, the Reference Implementation is referred to as the *RI*.

The native part of MIDP can be ported onto various platforms for which an ANSI C compiler is available. The high-level part is even more portable because it is implemented in the Java programming language. Sun Microsystems, Inc. has ports running on Solaris 2.8 and RedHat Linux 6.2, although these are not supported as RIs.

The RI has also successfully been built on Windows 2000 with the free tools from the CygWin project version 1.3.1, gcc and GNUmake. For more information about these tools, see the CygWin web site at `http://sources.redhat.com/cygwin`.

Note – The CLDC configuration must also be ported to any target platform for which a MIDP implementation is contemplated. For more information on CLDC porting, refer to the *KVM Porting Guide*, Sun Microsystems, Inc. (see "Related Documentation" on page xi)

Note – This 1.0.3 release of the MIDP RI requires building with the 1.0.3 release of the CLDC.

CHAPTER 2

MIDP Porting Considerations

Before undertaking a port of MIDP, it is worthwhile to consider some larger issues involved in this undertaking, and also to consider that it might be best to proceed in phases.

2.1 Porting versus Implementation

The code base provided with this release represents the RI of MIDP by Sun Microsystems Inc. A TCK (Java™ technology compatibility kit) for MIDP is also available from Sun Microsystems Inc.[1]

The characteristics of the RI are as follows:

- It is targeted to run in the Windows 2000™ environment.
- It requires approximately 1 MB of memory, much of which is accounted for by the emulator.
- It is the testbed against which the TCK was validated.

MIDP is defined in the *MIDP 1.0 Specification* (see "Related Documentation" on page xi). Numerous implementations of MIDP can be created. Sun Microsystems Inc.'s RI is only one. Device manufacturers will generally devise an implementation that is appropriate for their particular hardware needs. To be certified as compliant with the *MIDP 1.0 Specification*, an implementation must pass the MIDP TCK. For information on obtaining the MIDP TCK, see
`http://java.sun.com/products/midp/`

What we loosely refer to as porting is really a re-implementation of MIDP on a different target hardware environment (or platform). It is seldom practical or advisable to port the entire RI to your target platform. (There could be exceptions to this. Refer to "Degrees of Implementation" below.)

1. Also available from Sun Microsystems, Inc. are a TCK and a RI for CLDC.

2.2 Degrees of Implementation

The following options are available to a manufacturer that wishes to port the MIDP.

1. **RI port**

   Almost all of the code is from the RI.

2. **A mostly-RI implementation**

   Most of the code from the RI is retained.

3. **Mostly reimplemented**

   Half or more of the code is re-implemented around the target platform.

4. **A wholly-new implementation**

   The great majority of the code is re-implemented around the target platform.

   It is often appropriate for you to proceed in stages when attempting a port of the MIDP. Feasibility can be studied, familiarity gained with the code, and prototypes created through a degree 1 or 2 implementation, followed by a degree 3 or4 implementation that is finely tuned to the device.

2.3 Porting versus Optimization

There are two reasons to port the code, one to make it work in on different platform, and the second to optimize for size or speed.

The porting to another platform can be accomplished by changing native code for the GUI, storage, Jar reader, and TCP parts of the RI. Compared to an optimizing port, porting to another platform is fairly straightforward.

The porting for optimization involves selecting which Java classes to convert to native code for either size or speed reasons. Picking which Java methods to convert is not immediately obvious. Often, you must reimplement candidate Java methods and compare the size or performance results. Native methods have more invocation overhead and cannot reuse Java code. We have found that porting small methods or methods that reuse a high amount Java code often yields little or negative value.

Porting at the public API level will generally give good results if a native equivalent of the API is already on the device. For example, if `SSLStreamConnection` and its helper classes is judged to be too big, you would replace most of the methods in the class with native methods. Then, most of the supporting classes in the package can be deleted.

2.4 Native and Java Language Portions of The RI

The architecture of the RI divides the code between the Java programming language classes and the native code, which is written in C. For details on the Java and native files, refer to `DirectoryStructure.html` (in the `docs` subdirectory of this release).

2.5 The GUI Portion of The RI

A substantial portion of the RI, like the MIDP itself, is devoted to graphical user interface functionality. The GUI portion of the code is itself divided between high-level and low-level. High-level GUI functionality is provided by "widgets," and applications using this functionality are built by calls to a toolkit. Low-level GUI functionality gives control over graphics primitives like lines, arcs, bitmaps and text. Note that both high-level and low-level apis must invoke native frame-buffer drawing routines.

High-level GUI functionality might need to be reimplemented because of the character of a particular native device, especially if native GUI facilities are provided on that device.

Low-level GUI functionality is often conducive to a degree 1 or 2 implementation. (See "Degrees of Implementation" on page 4.) All or most of the Java code implementing the low level can be brought across unchanged from the RI. The native code will have to be changed to take advantage of the device's graphics capabilities.

For more details on porting the GUI portion of the RI, see **Chapter 3, "Porting the GUI**."

2.5.1 Helper Classes Removed

The 1.0 release of the MIDP RI contained a helper class that served as a porting layer. That helper class was useful when initial porting was done. But, since many method calls had to go through that porting layer, it slowed down performance.

To improve performance, the helper class was removed in MIDP1.0.3. As a result, native methods that used to be the implementation of helper class methods now implement native methods in, for example, `Display.java`, `Graphics.java`, `Font.java`, and `Image.java`.

Note – The signatures of these native methods cannot be modified since they are now part of the API.

2.6 The non-GUI Portion of The RI

A very important portion of the MIDP is devoted to non-GUI functionality. Each of the following parts of the RI code base must be evaluated for possible re-implementation on your target platform:

- MIDlet Management Software
  (The RI's implementation of the AMS—Application Management Software)
  (For more details, refer to **Chapter 5, "Porting the Application Management System**.")
- The Storage API
  (which supports RMS, the Record Management System defined in the *MIDP 1.0 Specification*.)
  (For more details, refer to **Chapter 4, "Porting the Storage**.")
- Networking support
  (especially the implementation of `HttpConnection`)
  (For more details, refer to **Chapter 6, "Porting the Networking Implementation**.")

CHAPTER 3

Porting the GUI

This chapter contains information on the architecture of the graphics (GUI) part of the RI, and gives helpful information on porting the GUI.

3.1 MIDP GUI overview

The MIDP graphics API is in the package `javax.microedition.lcdui`. This API is conceptually divided between high-level and low-level. For example, the low-level API contains the `Canvas` class, which is a base class for writing applications that need to handle low-level events and to issue graphics calls for drawing to the device's display.

By contrast, the high-level API contains the `Screen` class, which is the common superclass of all high-level user interface classes. The high-level API supports a degree of automatic functionality. For example, when an application changes the contents of a `Screen` object while it is shown to the user, the display is updated in a timely fashion without waiting for any further action by the application.

From an application development perspective, the `Canvas` class is interchangeable with high-level `Screen` classes, so an application may mix and match `Canvas` with high-level `Screen`s as needed.

At a class level, the division between high-level and low-level is as follows.

TABLE 3-1 Classes in Low-level Graphics API

`Display.java`	`AlertType.java`	`Command.java`
`CommandListener.java`	`Displayable.java`	`Canvas.java`
`Graphics.java`	`Font.java`	`Image.java`

The `Display` object is what the MIDlet uses to present itself to the user. The MIDlet might:

- use `AlertType` objects to signal the user with sounds
- create `Command` objects for actions the user can invoke
- respond to an invoked `Command` via a `CommandListener`
- show a `Displayable` object to create output on the screen

In the low level, the only `Displayable` object is a `Canvas`. It is responsible for its own rendition on the screen, for which it relies on `Graphics`, `Font` and `Image` objects. It also must do its own handling of key presses (and optionally pointer events).

TABLE 3-2 Classes in High-level Graphics API

`Screen.java`	`Ticker.java`	`Choice.java`	`Alert.java`
`List.java`	`TextBox.java`	`Form.java`	`Item.java`
`ChoiceGroup.java`	`DateField.java`	`Gauge.java`	`TextField.java`
`ImageItem.java`	`StringItem.java`	`ItemStateListener.java`	

The high-level portion of the API adds more `Displayable` objects, called `Screens`. The `Screen` objects handle key and pointer events automatically, freeing the application from this task. There are four types of `Screen`:

`Alert`	Simple notifications
`TextBox`	Scroll and edit a block of text
`List`	Simple list of choices
`Form`	Collection of closely-related items

`Form` allows the user to interact with collections of `Item` objects. These `Item`s are

`ChoiceGroup`	Checkboxes or radio buttons
`DateField`	Contains a date and/or time
`Gauge`	Bar graph, both interactive and non-interactive
`TextField`	An editable text string
`ImageItem`	A non-interactive item containing an image
`StringItem`	A non-interactive item containing a string

Changes to interactive items are signalled through an `ItemStateListener` object that is registered on the `Form`.

3.2 Porting The Low Level

Porting the MIDP 1.0.3 RI should be done by modifying native files such as `nativeGUI.c`, `text.c`, `images.c`, and `graphics.c`.

The MIDP 1.0 RI had a helper class that served as a porting layer between the `javax.microedition.lcdui` API and the system-specific libraries. All graphics operations and event handling used to be done through a helper class.

In the MIDP1.0 *Porting Guide* we noted that one of the ways to get performance improvements was to get rid of this level of indirection. In the MIDP 1.0.3 RI, the porting layer is removed, along with the indirection.

Here is an example of how a line was drawn in the MIDP 1.0 RI:

From `javax.microedition.lcdui.Graphics`:

```
    public void drawLine(int x1, int y1, int x2, int y2) {
        if ((clipX1 > clipX2) || (clipY1 > clipY2)) return;
            Display.deviceCaps.drawLine(this, destination,
                                        x1 + transX, y1 + transY,
                                        x2 + transX, y2 + transY);
        }
    }
```

Then, the helper interface `com.sun.midp.lcdui.DeviceCaps` had a method:

```
void drawLine(Graphics g, ImageDelegate dst,
                  int x1, int y1, int x2, int y2);
```

This default helper class had a native implementation for the `drawLine` method. From `com.sun.midp.lcdui.DefaultDeviceCaps`:

```
public native void drawLine(Graphics g, ImageDelegate dst,
                                  int x1, int y1, int x2, int y2);
```

In the MIDP 1.0.3 RI, the `drawLine` method is now implemented as:

From `javax.microedition.lcdui.Graphics.java`:

```
public native void drawLine(int x1, int y1, int x2, int y2);
```

Note that `defaultLCDUI.c` and `defaultLCDUI.h` were modified to fetch correct parameters. Also, argument checks that used to be done in Java code in `javax.microedition.lcdui.Graphics`, are now done in native code in `defaultLCDUI.c`.

3.2.1 Changes to the Porting Layer

In the RI, the default helper classes call native methods that are defined in `defaultLCDUI.c`. These methods simply do the required manipulations to fetch the arguments from the VM, and then call other functions which are documented in `defaultLCDUI.h`. Porting is just a matter of implementing the functions that are documented in this header file.

The RI (on the Windows 2000 platform) defines most of these functions in files called `nativeGUI.c`, `text.c`, `images.c`, and `graphics.c`. (Note that in MIDP 1.0.1 they used to be in one file, called `nativeGUI.c`).

By examining these files, you should be able to get a good idea of how to write your own versions of these graphics methods.

The remaining native methods from `defaultLCDUI.h` are implemented in "shared" code, in the files `menus.c`, `events_midp.c`, `imageDecode.c` and `pngDecode.c`. Most of these can probably be used without changes, although you might have better ways to do menus on your device.

3.3 Porting The High Level

The high-level code in the RI is written entirely in Java. If the low-level port has been properly done, this code should work as-is. However, it will likely not have a "look and feel" that is consistent with the rest of the GUIs on your device, so as part of a degree 3 or 4 implementation, you will probably want to replace some or all of the high-level `Screens` with native equivalents. (Refer to "Degrees of Implementation" on page 4.)

The typical way of replacing a `Screen` is as follows:

1. **Remove most of the code for the `Screen` in question. Leave 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 method

   ```
   void paint(Graphics g);
   ```

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.**

CHAPTER 4

Porting the Storage

This chapter contains information on the implementation of the storage API in the RI, and provides advice on porting an implementation on your target device. This API is new for the MIDP 1.0.3 RI. One very important use of the storage API is to implement MIDP's RMS (Record Management System), but it also supports any other part of the RI that requires persistent storage, such as storage for installed applications or for public cryptographic keys.

4.1 Overview

This release of the RI implements RMS with a new *flat file* internal storage API. (A flat file system is one that puts all files in a single directory. There are no subdirectories, so each file must have a unique name.) This storage mechanism is implemented in Java classes `RandomAccessStream` and `File`, and in native code. The Java classes that implement RMS make use of this new API, as detailed below in "Relationship to RecordStoreFile and RecordStore."

This storage API was chosen for the RMS in the RI because the same API is used in other parts of the RI:

- Over-the-air provisioning (OTA)
- the certificate authority public keystore (`KSSL`)

The low level of the API is used by:

- AMS native code
- the configuration native code

Thus, the new storage API makes the porting effort more efficient. It avoids a separate porting effort for every Java class, now and in the future, that needs persistent storage.

4.2 RandomAccessStream and File

The path to the RI classes `RandomAccessStream` and `File` is `com.sun.midp.io.j2me.storage`. The native interfaces can be found in `src/share/native/RandomAccessStream.c`, `storageFile.c`, `storage.h`, and `storage.c`. Together, the Java classes and native code provide the internal interfaces for the new low-level storage mechanism.

4.2.1 Relationship to RecordStoreFile and RecordStore

A `RecordStoreFile` is a file abstraction layer between a `RecordStore` and an underlying persistent storage mechanism. The underlying storage methods are provided by the `RandomAccessStream` and `File` classes.

The `RecordStore` class can be implemented directly using the `RandomAccessStream` and `File` classes. However, `RecordStoreFile` served as the Java/native code boundary for RMS in the MIDP 1.0 RI release. It exists now for backwards compatibility with older ports.

4.3 Porting the Low Level

The high level of the API has no system-specific code, so only the low level needs to be ported. The requirement of the low level interface is in `storage.h`, and the shared implementation is in `storage.c and storageFile.c`.

There are two ways to approach the low level port. The first approach is to add a new set of `ifdef` statements in `storage.c`. Copy the code in the `ifdef UNIX` block and customize it for your platform. The second approach is to reimplement `storage.c` to directly use a flat file system interface.

4.4 Porting the High Level

If your device is severely constrained in non-volatile memory for storing classes, you might have to port the high level of the internal storage API. To port the high level API, rewrite the public methods classes of the `com.sun.midp.io.j2me.storage` package.

4.5 Porting RMS

Generally speaking, it is unnecessary to port any of the Java classes that implement RMS (`com.sun.midp.rms`). However, if your device is severely resource-constrained, you might need to replace the entire RMS with native code.

CHAPTER 5

Porting the Application Management System

The function of the application management system (AMS) is to install, run, remove, and list MIDlet suites, either from a command line or from a graphical interface. On development platforms it also runs a single MIDlet from the classpath.

The AMS requires local persistent storage. It makes use of the same low-level storage mechanism as used by the RMS. When you have accomplished the porting of `storage.c` as detailed in **Chapter 4, "Porting the Storage**," you have also have ported the AMS.

5.1 Porting Main

The start up and initialization of the AMS runtime environment, and the interface to the Java program command loop, is implemented in `main.c` (located in src/share/native).

`main.c` implements the command loop for the Java program `com.sun.midp.Main`. In `main.c`, the command line is parsed, the command state structure is created, and the Java program `com.sun.midp.Main` is called.

`main.c` uses the native storage interface. (Refer to **Chapter 4, "Porting the Storage**.")

If your implementation needs a different way of specifying the command line parameters, or if your implementation changes the command set or any command option is changed, you must provide a different `main.c`.

5.2 Porting the High Level

The high level code is in contained in the packages `com.sun.midp`, `com.sun.midp.dev`, and `com.sun.midp.midletsuite`. If a command option or options are being removed, then no change to high level is needed. If a command is being removed (and size does not matter) then no change to the high level is needed. If there are severe space constraints on your target device, it might be necessary to re-implement some of the high-level Java classes.

5.2.1 Command Categories

There two command categories:

- development only
- on-device (fit for purpose)

Removing all the development code requires a new `main.c`, a new `Main.java`, and not including the `com.sun.midp.dev` package in the build. Removing the graphical interface requires a new `main.c`, a new `Main.java`, and requires not including `Manager.java` and the png file from the `com.sun.midp.dev` package in the build. Replacing command state processing requires replacing the `com.sun.midp` package.

5.2.2 Replacing OTA

After the MIDP 1.0.1 release the native code used to install and run MIDlet applications (also known as JAM) was replaced with a graphical interface and a Java implementation of the Application Management Software (AMS). This code fully implements the recommendations for over-the-air provisioning (OTA) that were published as an addendum to the original MIDP 1.0 Specification.

Over-the-air provisioning (OTA) supports updating MIDlet suites (in addition to installing, listing, removing, loading and updating them) dynamically and over a wireless connection. Replacing the OTA implementation requires a new `com.sun.midp.midletsuite` package. Replacing the entire high level code with native code requires writing the following Java classes:

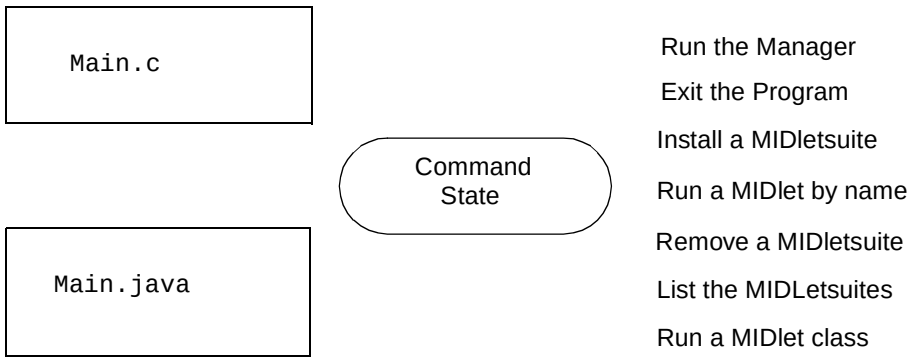
- A class that implements the `com.sun.midp.midlet.MIDletSuite` interface.
- A class that initializes the internal security.

Replacing the entire high level code with native code also requires not including these packages in the build:

```
com.sun.midp
com.sun.midp.dev
com.sun.midp.midletsuite
```

In many cases, porting the MIDP RI onto an existing device platform involves leveraging as much of the current platform native as possible. When a device already supports a native browser, the browser contains much of the same code included in the Java Installer: for example, fetching documents via HTTP connections (with appropriate `Accept` headers), and checking for appropriate MIME types of returned resources. At this time there isn't a standard interface between a browser application on the device and the Java runtime environment, so the integration will be a custom effort on most platforms.

To get a quick understanding of the Graphical AMS, examine the hand-off from the native code `src/share/native/main.c` and the initial Java code executed in `src/share/classes/com/sun/midp/Main.java.`



The Command State is used between successive launches of the virtual machine to inform the native code what to do after each iteration from the Installer and application manager.

Once in the Java AMS subsystem, the following packages support the basic MIDlet management functions.

```
com.sun.midp
com.sun.midp.security
com.sun.midp.midlet
com.sun.midp.midletsuite
com.sun.midp.dev
com.sun.midp.publickeystore
```

CHAPTER 6

Porting the Networking Implementation

This chapter contains information about the RI's networking capabilities, and gives important porting information.

6.1 CLDC Specification of Generic Connections

The CLDC 1.0 Specification establishes the basic architecture for all stream based IO Connections.

`javax.microedition.io.Connector`

`javax.microedition.io.DatagramConnection`

`javax.microedition.io.ConnectionNotFoundException`

`javax.microedition.io.StreamConnection`

`javax.microedition.io.Datagram`

`javax.microedition.io.InputConnection`

`javax.microedition.io.OutputConnection`

`javax.microedition.io.StreamConnectionNotifier`

`javax.microedition.io.ContentConnection`

`javax.microedition.io.Connection`

The CLDC reference implementation includes a basic set of classes that handle the fundamental capabilities of generic connections.

`com.sun.cldc.io.GeneralBase`

`com.sun.cldc.io.ConnectionBaseInterface`

```
com.sun.cldc.io.NetworkConnectionBase
com.sun.cldc.io.DateParser
com.sun.cldc.io.ConnectionBase
```

6.2 CLDC Implementation of TCP/IP Sockets

The CLDC RI also includes the code for basic TCP/IP socket manipulation. While this code is beyond the scope of the actual CLDC specification, it is a useful porting layer to platforms that support some form of socket library capability.

```
com.sun.cldc.io.j2me.socket.Protocol
com.sun.cldc.io.j2me.socket.PrivateInputStream
com.sun.cldc.io.j2me.socket.PrivateInputStreamWithBuffer
com.sun.cldc.io.j2me.socket.PrivateOutputStream

com.sun.cldc.io.j2me.datagram.Protocol
com.sun.cldc.io.j2me.datagram.DatagramObject

com.sun.cldc.io.j2me.serversocket.Protocol
```

The native networking code to support all of the datagrams and socket classes resides entirely in the CLDC code base. The MIDP no longer modifies these files.

6.3 MIDP Specification/Implementation of HTTP 1.1 Protocol

The MIDP RI layers the HTTP 1.1 protocol semantics in pure Java classes on top of the CLDC `socket://` URL support.

This organization provides for maximum portability, but might not provide the most optimum performance or size. If a native implementation of HTTP exists on the target platform, it might be more efficient to completely replace the MIDP implementation.

```
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
```

6.3.1 MIDP Implementation of HTTP Proxy

The HTTP implementation includes support for a proxy http server. The configuration parameter, `com.sun.midp.io.http.proxy`, can be set with a host and port number to delegate the actual HTTP requests to a different machine. (Configuration parameters are set in the configuration file.)

This can also be set at the command line. For example,

```
bin/midp -Dcom.sun.midp.io.http.proxy=webcache.east.sun.com:8080
```

6.3.1.1 Http Tunneling through Web proxy servers.

The MIDP RI requires that HTTP proxy servers that are used with MIDP application support the generic tunneling mechanism for TCP based protocols through Web proxy servers. This tunneling mechanism was initially introduced for the SSL protocol to allow secure Web traffic to pass through fire walls, but its usefulness is not limited to SSL. Implementations of this tunneling feature are commonly referred to as "SSL tunneling," although it can be used for tunneling any TCP based protocol.

For further information on HTTP Tunneling, please refer to the following documents:

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

6.3.2 6.3.2 HTTP1.1 Persistent Connections.

Prior to the implementation of persistent connections, a separate TCP connection was required to fetch each URL, increasing the load on HTTP servers and causing congestion on the Internet. The use of inline images and other associated data often required a client to make multiple requests of the same server within a short time frame.

Persistent HTTP connections have a number of advantages:

- By opening and closing fewer TCP connections, CPU time is saved in routers and hosts (clients, servers, proxies, gateways, tunnels, or caches), and memory used for TCP protocol control blocks can be saved in hosts.

- Latency on subsequent requests is reduced, since there is no time spent in TCP's connection opening handshake.
- HTTP can evolve more gracefully, since errors can be reported without the penalty of closing the TCP connection. A client using a future version of HTTP might optimistically try a new feature, but if communicating with an older server, an error would be reported. The client could then retry with old HTTP semantics using the same TCP connection.

6.3.2.1 Cleaning Up Idle Persistent Connections.

The `CONNECTION_ACTIVE_WAIT_TIME` value sets the time limit for inactive HTTP connections in the connection pool. Once that time limit is exceeded, the connection is automatically removed by the connection pool. This is useful for small devices (like Palm OS devices) with limited resources (such as connections). Such devices often need to clean up the pool, especially when active connections were not closed properly.

```
[StreamConnectionPool.java]
83    private final long   CONNECTION_ACTIVE_WAIT_TIME = 60000;
```

6.4 Enabling Additional Protocols

The MIDP input/output infrastructure uses an alternate implementation of the `Connector` class to allow default behavior that can exclude the additional protocol implementation classes. This means the MIDP implementation can expose only the HTTP protocol covered in the MIDP 1.0 specification.

To expose these additional protocols, set the configuration parameter `com.sun.midp.io.enable_extra_protocols` to *true*. This restores access to CLDC's `socket://`, `serversocket://`, `datagram://`, and `comm:` implementations, as well as `https://` protocol support.

`com.sun.midp.io.InternalConnector`

For more documentation on these protocols, rebuild the Javadoc for the internal packages, using the appropriate GNUmakefile target, as detailed in `Compile.html`.

6.5 Porting HTTPS

MIDP 1.0.3 RI includes an implementation of secure HTTP.

HTTPS subclasses HTTP, and thus inherits all of the protocol handling of the MIDP HTTP implementation. HTTPS replaces the TCP/IP `StreamConnection` with a SSL-TCP/IP `StreamConnection`. The HTTPS protocol handler overrides the methods that connect to and disconnect from the origin server. All handling of certificates and encryption of the network data is handled in the internal implementation of the `SSLStreamConnection`.

For specific HTTPS features, see the Javadocs documentation for the `com.sun.midp.io.j2me.https` package.

Note – HTTPS is prototype API, since it is not part of the MIDP specification, and is not required to be on a MIDP 1.0 compliant device. To keep the RI within the MIDP 1.0 specification, no new public APIs have been defined for this feature.

Other implementations of HTTPS have been deployed. To bring about standardization, the JSR118 expert group for "MIDP Next Generation" is considering an officially supported MIDP API for HTTPS. Be alert for developments at: `http://jcp.org/jsr/detail/118.jsp`

6.5.1 Porting the Low Level

The only platform-specific issues are with native memory allocation. See `src/share/native/crypto/kvmpilot.h`.

6.5.2 Porting the High Level

To replace the entire implementation, replace the `com.sun.midp.io.j2me.https` package.

To change the way public keys are stored, replace the `com.sun.midp.publickeystore` package.

To change the tool to store public keys, modify `MEKeyTool.java` in the `com.sun.midp.mekeytool` package (located in the `tools` subdirectory).

To change the SSL implementation the HTTPS uses, replace the `com.sun.kssl` and `com.sun.ksecurity` packages.

The current implementation of HTTPS does not expose an API to control the handshake listener. This would allow certain certificate errors to be over ridden by higher level applications. Not exposing the handshake listener allows the current MIDP 1.0 specification of HTTP to be used unchanged, but does not make it possible for an end user to override specific policy decisions about expired certificates.

CHAPTER 7

Building a Different Executable

This chapter contains advice to help you build different versions of the `midp` executable command. In addition to the discussion in this chapter, HTML documentation is included in this release for advice on setting up builds, including system requirements, configuration settings, makefile targets, and so forth.

In the `docs` directory of this release, refer to `Compile.html`, `Configuration.html`, and `Running.html`.

7.1 Make Variables

The MIDP 1.0.3 release includes many improvements in the makefiles used to create the `midp` executable. The makefiles have been streamlined to make them more modular, and a consistent approach has been used to make it easy to eliminate various optional features. You simply set a command line switch (makefile target) while building the midp.

The following table lists the optional components:

TABLE 7-1 Makefile flags

Flag	Default Value	Comments
DEBUG_COLLECTOR	*false*	A debugging version of the KVM garbage collector can be used to assist in debugging memory reclamation problems. This switch controls the compilation flag USE_DEBUG_COLLECTOR and also sets EXCESSIVE_GARBAGE_COLLECTION to help accelerate the finding of misplaced objects.
ENABLEPROFILING	*false*	Turning off profiling effects the compilation flag ENABLEPROFILING. Setting ENABLEPROFILING=*false* removes the native code collection of profiling data and the associated Java classes used in the test harness collection of performance related statistics. (perfmon.c)
DEBUG	*false*	This variable controls the amount of symbolic information included in the executable. Setting DEBUG=false eliminates the extra information included by the -g compilation flag and extra libraries used for -debug linkage.
ENABLE_DEBUGGER	*false*	The KVM debugger proxy can interact with the MIDP executable when this flag is enabled. Setting ENABLE_DEBBUGGER=false removes the native code used to communicate with an external debugger proxy. (debugger.c debuggerSocketIO.c debuggerOutputStream.c debuggerInputStream.c)
ENABLE_SCREEN_ CAPTURE	*true*	The screen capture utility is used in automated testing of the MIDP user interface. Setting ENABLE_SCREEN_CAPTURE=false removes the native code that captures a simple CRC for the current displayed graphics and an associated Java class which interfaces to the Sun Microsystems Inc. internal test harness. (screengrab.c crc32.c screenGrabber.c)

TABLE 7-1 Makefile flags

Flag	Default Value	Comments
INCLUDE_I18N	*true*	Internationalization support in the MIDP workspace includes support for character set conversions, and locale specific messages. Setting INCLUDE_I18N=*false* eliminates the extra routines, and filters out non-essential classes from the MIDP and CLDC directories. (localeMethod.c conv.c locale.c genConv.c)
INCLUDE_ALL_ CLASSES	*true*	The INCLUDE_ALL_CLASSES flag controls whether or not extra generic connection protocols are included in the midp executable. When INCLUDE_ALL_CLASSES=*false* is set only the protocols required in the specification are included. For example, only the http protocol is included in the MIDP 1.0 specification. Additional protocols that are often used are the *socket* and *comm* protocols.
ROMIZING	*true*	When ROMIZING is set to *false*, the core classes are not built into the midp executable, but are left in a separate classes.zip file. This can be done for debugging purposes, or to have a concise set of classes that can be used for compiling new applications.
SLOW_DRAWING	*false*	When SLOW_DRAWING is enabled, the native code responsible for rendering graphics requests includes artificial delays. The default slow-down is 100,000 microseconds per graphics operation.
SOUND_SUPPORTED	*false*	When SOUND_SUPPORTED is enabled, audible feedback is added to button presses and other key inputs on the phone skin.
THROTTLE	*false*	When THROTTLE is enabled an extra function is called inside the native event loop to insert fixed delays in the number of byte code executions per millisecond.
INCLUDE_HTTPS	*false*	When INCLUDE_HTTPS is *true*, the HTTPS protocol implementation classes and native code are included.
INCLUDEDEBUGCODE	*false*	When INCLUDEDEBUGCODE is *true*, it includes a large amount of debugging and logging code in the CLDC virtual machine.

The following special targets are supported:

TABLE 7-2 Special GNUmakefile targets

Target	Description
small	Builds the smallest possible midp executable.
debug	Enables all of the debugging flags.
profile	Approximates the previous release default settings, with profiling support enabled.

7.2 Removing Configuration Code

In the RI, a configuration mechanism is used to select an alternate behavior at runtime. In commercial ports of the RI, many of these design choices will have been made and fixed into the hardware itself. (For example, a color screen versus a black-and-white-only screen.)

A simple way to locate these branches in the code is to look for the use of the `Configuration.getProperty` calls. In each of these locations it might be possible to remove the code that is not required in your specific port. Here's a current list of classes that include optional behavior based on calls to access configuration parameters:

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

CHAPTER 8

Thread-Safety

This chapter describes the coding conventions that ensure thread-safety for LCDUI, the classes 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).

8.1 Requirements

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

Requirement #1

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

Requirement #2

The "event delivery" calls made by LCDUI into the application must be serialized, as required by the *MIDP 1.0 Specification*. This set of calls is defined in class documentation for `javax.microedition.lcdui.Canvas`.

Requirement #3

Applications must be allowed to define and implement their own locking policy for their data structures and be assured that they will run correctly (that is, deadlock-free and safe) when they interact with LCDUI.

8.2 Design Approach

The MIDP RI 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. The general approach 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; and
- 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" on page 38 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.

8.3 Coding 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

8.3.1 Public Methods

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

```
public void setMaxValue(int maxValue) {
    if (maxValue <= 0) {
        throw new IllegalArgumentException();
    }

    synchronized (Display.LCDUILock) {
        // ...
    }
}
```

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 synchronization 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 RI, therefore, made a design decision to avoid locking overhead in cases such as these by relying on the 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.

8.3.2 Constructors

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.

8.3.3 Event Handling Methods

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.

8.3.4 Application Callouts

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:

- `showNotify`
- `hideNotify`
- `keyPressed`
- `keyRepeated`
- `keyReleased`
- `pointerPressed`
- `pointerDragged`
- `pointerReleased`
- `paint`
- A `CommandListener` object's `commandAction` method
- An `ItemStateListener` object's `itemStateChanged` method [1]
- A `Runnable` object's `run` method resulting from an earlier call to `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 Form.java where the application's `ItemStateListener` is called:

```
void keyPressed(int keyCode) {
    Item curItem;
    ItemStateListener isl = null;

    synchronized (Display.LCDUILock) {
        curItem = items[curItemIndex];
        if (Display.getGameAction(keyCode) == Canvas.FIRE
                && curItem.select()) {
            isl = itemStateListener;
        }
    }

    if (isl != null) {
        synchronized (Display.calloutLock) {
            isl.itemStateChanged(curItem);
        }
    }
}
```

(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.

1. The *MIDP 1.0 Specification* probably ought to have included this callout in the defined set of serialized calls, but it was apparently left off the list by mistake.

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.

8.3.5 Graphics

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) {
        ...
    }
    public native void drawLine(int x1, int y1, int x2, int y2);
}

class OffscreenGraphics 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 Java code. However, the `drawLine` method is simply a direct call to a native method, which the MIDP RI 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.

8.3.6 Command Menus

The command menus are implemented in native code. Changes to the command set on the current screen are pushed to the native implementation immediately. This code ensures that the highlight is moved properly if commands are added or removed, and that something reasonable happens, such as cancellation, if all the commands on the menu are removed.

Every `Command` object is assigned a unique serial number at the time it is created. When the user invokes a command from a menu, the `Command` object's serial number is passed through the event queue. Upon receipt of this event, the event handling code searches for this serial number in the current `Displayable` object's command set, and issues the command if it is found.

This technique is necessary for several reasons. Prior to the multi-thread work, the system pushed into the event queue the index of the command within the current `Displayable` object's commands array. When this event is processed by the event handler, the commands array might have been modified, or a different `Displayable` object might be current. This might cause event handler to invoke the wrong `Command`, or it might cause an internal error. The serial number technique avoids both of these problems. If a `Command` object with a matching serial number is not found in the current `Displayable` object's command set, the event is simply ignored.

8.4 The `serviceRepaints` Method

The *MIDP 1.0 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" on page 31.) 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 1.0 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 32. 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" on page 31. 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 9

Porting Example

The example contained in this chapter illustrates a re-implementation of a native method in the RI, illustrated with real-world code. The example can be best appreciated by comparing before (from the RI) and after (as re-implemented) code with reference to the steps listed in "Porting versus Implementation" on page 3.

The functionality that was re-implemented in this case was editing of a time value. The time value is represented as part of an Item called a `DateField`. `DateField`'s API is defined as part of the MIDP specification, but the actual editing of the value (and even the means by which the user edits the value) is outside the scope of this specification. Hence there is considerable freedom about how to implement the behaviors.

The file `DateField.java` must be modified as described in "Porting versus Implementation" on page 3. In addition, new native methods must be added to the system in order to support the changes made to `DateField.java`

For this example, we assume that there is a built-in editor for time values. Furthermore, we assume that it has the following very simple 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);
```

9.1 Changes to `DateField.java`

In the RI, date and time editing are accomplished by a single helper class called `EditScreen`. Our first task is therefore to break this apart. The functionality for editing a date is retained in a new helper class, `DateEditor` (not shown). Then we write a second helper class, `TimeEditor`, which looks like this:

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

    Command Back = new Command("Back", Command.BACK, 0);
    Command OK   = new Command("Save", 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) {
        calendar.setTime(currentValue);
        int timeVal = calendar.get(Calendar.HOUR_OF_DAY)*60
                    + calendar.get(Calendar.MINUTE);

        setTime(timeVal);
    }

    /**
     * Handle a command action
     *
     * @param cmd The Command to handle
     * @param s   The Displayable with the Command
     */
    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 paint(Graphics g) {
         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());
```

Notice the use of native methods for setting and getting the time value, showing and hiding the screen, and refreshing a portion of the display. Also notice that, because the abstract command mechanism is available to us, we can do OK and Back buttons the same way we did in the RI. If the native component were to handle Back and OK itself, then we would need to devise some way for these user commands to be relayed back to the system.

We must also change some of the event handling code so that the correct editor is invoked. For example, compare the RI version of the `select` method:

```
boolean select() {
        if (editor == null) {
            editor = new EditScreen(this);
        }

        if (mode == DATE_TIME) {
            editor.setDateTime(calendar.getTime(),
                               (highlight < 2) ? DATE : TIME);
        } else {
            editor.setDateTime(calendar.getTime(), mode);
        }

        editor.layoutChanged();

        getOwner().currentDisplay.setCurrent(editor);

        return false;
    }
```

with the new one below:

```
    /**
     * Perform a selection on this DateField
     *
     * @return boolean Always returns false
     */
    boolean select() {
        Screen returnScreen = getOwner();

   if (!initialized) {
       if (mode == DATE) {
      currentDate.set(Calendar.HOUR, 0);
      currentDate.set(Calendar.MINUTE, 0);
      currentDate.set(Calendar.SECOND, 0);
      currentDate.set(Calendar.MILLISECOND, 0);
       } else if (mode == TIME) {
      currentDate.setTime(EPOCH);
       }
   }

        int editMode = this.mode;

        if (editMode == DATE_TIME) {
            editMode = (highlight < 2) ? DATE : TIME;
        }
```

```
    if (editMode == DATE) {
        if ((editor == null) || !(editor instanceof EditScreen)) {
                editor = new EditScreen(returnScreen, this);
            }
            ((EditScreen)editor).setDateTime(currentDate.getTime(),
DATE);
            ((EditScreen)editor).layoutChanged();
    } else if (editMode == TIME) {

        if ((editor == null) || !(editor instanceof TimeEditor)) {
                editor = new TimeEditor(returnScreen, this);
            }

           ((TimeEditor)editor).setDateTime(currentDate.getTime());
         }
         returnScreen.currentDisplay.setCurrent(editor);

         return false;
     }
```

9.2 Additional native methods

In the previous section, we declared native methods as part of the definition of `TimeEditor`. We must provide C language functions that satisfy these declarations. Fortunately, in the case of `TimeEditor`, these are very straightforward. They primarily consist of some stack manipulation, plus calls to the built-in time editor.

```
void
Java_javax_microedition_lcdui_TimeEditor_paint0()
{
    int clipHeight = popStack();
    int clipWidth  = popStack();
    int clipY      = popStack();
    int clipX      = popStack();

    oneLess; /* pop the reference to the TimeEditor instance */

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

void
Java_javax_microedition_lcdui_TimeEditor_setTime()
{
    int timeVal = popStack();

    oneLess; /* pop the reference to the TimeEditor instance */

    TimeEditor_setTime(timeVal);
}

void
Java_javax_microedition_lcdui_TimeEditor_getTime()
{
    /* replace the reference to the TimeEditor instance */
    oneLess;
    pushStack(TimeEditor_getTime());
}

void
Java_javax_microedition_lcdui_TimeEditor_showNotify()
{
    oneLess; /* pop the reference to the TimeEditor instance */

    TimeEditor_setVisible(TRUE);
}
```

```
void
Java_javax_microedition_lcdui_TimeEditor_hideNotify()
{
    oneLess; /* pop the reference to the TimeEditor instance */

    TimeEditor_setVisible(FALSE);
}
```

CHAPTER 10

Compiler Requirements

In order to be able to compile the native part of the MIDP, you must have a C compiler capable of compiling ANSI-compliant C files. Your compiler must define the basic C types as shown below in Table 10-1.

TABLE 10-1 Basic types

Type	Description
`char`	An 8-bit quantity. It can be signed or unsigned
signed `char`	A signed 8-bit quantity.
unsigned `char`	An unsigned 8-bit quantity
`short`	A signed 16-bit quantity.
unsigned `short`	An unsigned 16-bit quantity
`int`	A signed quantity of 32 bits.
unsigned `int`	A unsigned quantity of 32 bits.
`long`	A signed 32-bit quantity
unsigned `long`	An unsigned 32-bit quantity.
`void *`	A 32-bit pointer

All MIDP implementations must support the Java type `long`. It is not required that your compiler support 64-bit integers; however this is preferred when porting the KVM.

Your compiler must have some means of indicating additional directories to be searched for "includes" of the form:

```
#include <filename>
```

The MIDP RI has only been tested on machines with 32-bit pointers and 32-bit `int`s, and machines that do not require “far” pointers of any sort. It is unknown whether it will run successfully on platforms with pointers of other sizes.

The code base has been successfully compiled with the following compilers:

- Sun Microsystems Inc. C Compiler 5.0 and 5.2 on Solaris,
- GNU C 2.95.2 compiler on Solaris and Windows NT 4.0 (see `http://www.gnu.org/software/gcc/gcc-2.95/gcc-2.95.2.html`),
- CygWin project version 1.3.1 with gcc and gnumake (see `http://sources.redhat.com/cygwin`),
- Microsoft Visual C++ 6.0 Professional on Windows NT 4.0.

The only non-ANSI feature in the source code is its use of 64-bit integer arithmetic.