Oracle® Java Micro Edition Embedded Client

Reference Guide, Version 1.0

Part No.: 01-11-11
January 2011

Contents

Using This Documentation

This service manual explains how to use the Oracle® Java Micro Edition Embedded Client interfaces to create and test applications.

OS Commands

This document might not contain information about basic Linux or Windows commands and procedures such as shutting down the system, booting the system, and configuring devices. Refer to the following resources for this information:

- Ubuntu operating system documentation, which is found at:

  `https://help.ubuntu.com`

- Windows operating system documentation or online help
- The *Oracle Java Micro Edition Embedded Client Installation Guide* contains limited descriptions of operating system commands used with the Oracle Java Micro Edition Embedded Client.

Related Documentation

The following table lists the documentation that is related to this product.

Application	Title	Format
Release Notes	*Oracle Java Micro Edition Embedded Client Version 1.0 Release Notes*	Printed PDF
Installation Guide	*Oracle Java Micro Edition Embedded Client Installation Guide*	PDF HTML

PART I Developer Guide

Part 1 includes the following developer topics:

Introduction:

Provides an overview of the software stacks, target device software and hardware components.

Execution:

Execution and runtime information, and demos to run.

Developing Applications:

Discusses recommended developer tools and how to set up the development environment.

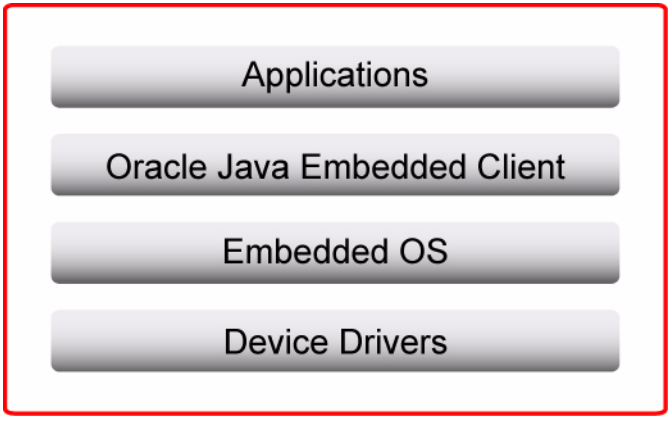
CHAPTER 1

Introduction

Oracle Java Micro Edition Embedded Client (Oracle Java ME Embedded Client) provides the Java ME platform for embedded devices. Oracle Java ME Embedded Client is based on Connected Device Configuration (CDC) technologies and caters to a wide range of embedded application and middleware needs, ranging from TV set-top boxes to VoIP phones. Oracle Java ME Embedded Client can also support various embedded platforms, such as e-book Readers, multi-functional peripheral devices, smart electric metering devices, and Blu-Ray Disc players.

1.1 Software Overview

Oracle Java Micro Edition Embedded Client provides a platform for running applications written in the Java programming language. This platform, combined with GUI elements, provides rich, robust, and scalable application domains.



The Oracle Java Micro Edition Embedded Client needs specific drivers and software to host the Java technologies on embedded devices. It is based on several layers of software:

- An embedded operating system running on the device. For example, Linux OS.
- A native graphics library or display driver to provide access to the device graphics display plane (framebuffer)

The Oracle Java ME Embedded Client environment includes the following Java technologies:

- *Connected Device Configuration* HotSpot™ *Implementation* (CDC HotSpot Implementation, version 1.1.2) is a Java ME platform that enables Java technology on resource-constrained devices.
- Foundation Profile (1.1.2) with Security Optional package (1.0.1)
- Personal Basis Profile (1.1)
- RMI Optional Package (1.0)
- JDBC Optional Package for CDC/Foundation Profile (1.0)
- J2ME Web Services

1.2 Target Device Hardware Components

These subsections outline some of the typical base requirements and features to support the Oracle Java Micro Edition Embedded Client on an embedded environment.

1.2.1 Microprocessor Requirements

Oracle Java ME Embedded Client uses the Oracle JIT (Just In Time) compiler technology, CDC-Hotspot Implementation. Ports for this virtual machine are available on numerous microprocessors. It performs best on 32-bit RISC-based microprocessors (MIPS, ARM, PowerPC, SPARC and others) and 32-bit Intel x86 microprocessor. Typical clock speeds vary from low-end devices (150-200 megahertz) to mid-range devices (300 megahertz) and faster.

1.2.2 I/O and Peripheral Requirements

This section provides a basic breakdown of core device peripherals and I/O support characteristics.

1.2.2.1 Input Ports for Flash Updating

Serial ports may be included on device but are typically only used for in factory device setup and configuration and are not accessible by developers or end users. This input mechanism can be used to flash upgrade the device.

1.2.2.2 I/O Ports for Data

An ethernet port is required for the Java networking functionality.

Typically, IR ports are also available to support input events from remote control keypads for TV and media platforms. Some devices also support USB ports for writing to flash memory or to support extra peripheral devices (camera connection or full keyboard).

1.2.2.3 Graphics Output Ports

Java represents each graphics pixel as 32 bits (ARGB) but lower color-depth displays can be supported, down to monochrome (1-bit).

1.2.3 Memory Requirements

The Oracle Java ME Embedded Client is designed to run on memory constrained devices. Oracle's CDC HotSpot implementation virtual machine by itself with core CDC classes (headless) takes less than 5.5 megabytes of ROM with an additional requirement of 1.5 megabytes if internationalization is required.

1.2.3.1 RAM

Usage of RAM typically depends on the Java application being executed. An average use case for graphics applications can require up to 32 megabytes of RAM. The memory requirement would be less in the case of non-graphics applications. This typically implies a device with 64 megabytes of physical RAM when graphics use cases are required. A larger amount of RAM might be needed to support applications with large Java heap requirements. Smaller, non-graphics applications can require as little as 4MB RAM or less.

1.2.3.2 ROM and Flash

Oracle Java ME Embedded Client requires close to 5 megabytes of ROM. An additional 1.5 megabytes is required for internationalization support.

1.3 Target Device Software Components

This section discusses the basic components required to support the Oracle Java Micro Edition Embedded Client in an embedded environment.

1.3.1 Operating System

Oracle's CDC HotSpot Implementation virtual machine runs on numerous operating systems, including Unix and Linux variants and other well known commercial offerings. Oracle Java ME Embedded Client is optimized for a Linux host OS and Windows XP OS. When the specific target device uses a proprietary operating system, the basic requirements to support Connected Device Configuration and Foundation Profile are as follows:

- Basic ANSI C and standard C library
- Multithreading
- Synchronization: mutexes, condvars, semaphores
- Millisecond timer
- File I/O (POSIX-like functionality)
- Full TCP/IP stack

CHAPTER 2

Execution

Oracle Java Micro Edition Embedded Client is the foundation for software product development for many embedded client platforms. The client stack can be ported to other platforms on request. The client software can be modified or extended to provide the appropriate platform for Java applications or middleware. Applications and middleware can be designed and implemented by partners or third parties catering to end-user needs.

2.1 Components

This section identifies the typical binary components of the Oracle Java ME Embedded Client. This reflects how the stack has been installed in various devices.

The Linux/x86 bundle (software development kit) contains prebuilt virtual machine binaries and classes for target devices as follows:

```
/docs
  /cdc-1_1_2-mrel-spec-jdoc
  /fp-1_1_2-mrel-spec-jdoc
  /j2me_rmiop-1_0-fr-spec
  /j2me_web_services-1_0-fr-spec-jdoc
  /jdbc_cdc-1_0-fr-doc
  /pbp-1_1_2-mrel-spec-jdoc
/emulator-platform
  /bin
    emulator
  btclasses.zip
  /lib
    /profiler
      /lib
    /security
    /ext
```

```
    /fonts
      /zi
        /America
        /Asia
/legal
```

2.2 Execution and Supported Runtime Parameters

The supported runtime parameters can be considered separately for the virtual machine and CDC and the supported profile.

To print the version, use the `-version` command option. The output is similar to the following:

```
Product: Oracle Java Micro Edition Embedded Client-1.0
(built on Profile: CDC 1.1.2 (JSR218)
- FP 1.1.2 (JSR219) (SecOp 1.0)
- PBP 1.1.2 (JSR217)
- (RMI JSR66)
- (JDBC JSR169)
- (WebServices JSR172)- (Specification 1.1.2)
- Toolkit: DirectFB (1.2)
JVM:     CVM rev (mixed mode)
```

2.3 Mouse Restrictions

A Personal Basis Profile implementation can also optionally provide partial support for a pointing device. The Personal Basis Profile specification states:

"If the property `java.awt.MouseEvent.isRestricted` is true, then the property `java.awt.event.MouseEvent.supportLevel` reports the level of mouse events generated by the implementation." The following table describes the mouse events generated by the implementation based on the support level.

TABLE 2-1 Level of Mouse Events Generated by the Implementation

Support Level	Mouse Events Generated
0	No mouse events generated
1	MOUSE_CLICKED, MOUSE_PRESSED, MOUSE_RELEASED, MOUSE_ENTERED, MOUSE_EXITED
2	All the events of level 1, plus MOUSE_MOVED

2.3.1 Input Mechanisms

The following sections describe the various input mechanisms: standard keyboard, events outside the PBP specification, and mouse support.

2.3.1.1 Standard Keyboard Support

Developers can query the particular key events supported, by querying the following runtime properties:

```
java.awt.event.KeyEvent.isRestricted
```

and

```
java.awt.event.KeyEvent.supportMask
```

Typical values are:

```
java.awt.event.KeyEvent.isRestricted =true
java.awt.event.KeyEvent.supportMask=0x07
```

Hence the Oracle Java ME Embedded Client supports the following subset of key events that match typical remote control input capabilities:

```
VK_LEFT and VK_RIGHT
VK_UP and VK_DOWN
VK_0 through VK_9
VK_ENTER
```

2.3.2 Mouse Support

Developers can determine the particular mouse events supported, by querying the following runtime properties:

```
java.awt.MouseEvent.isRestricted
```

and

```
java.awt.event.MouseEvent.supportLevel
```

Typical values are:

```
java.awt.MouseEvent.isRestricted=true
java.awt.event.MouseEvent.supportLevel=0
```

The last value indicates that Oracle Java ME Embedded Client supports no mouse events.

CHAPTER 3

Developing Applications

This chapter describes the basic application object and its life cycle. It also describes how to write, build and deploy simple applications.

Developing for Oracle Java Micro Edition Embedded Client is similar to developing Java Platform, Standard Edition (Java SE) applications. Experience with AWT on the desktop helps in learning to write Oracle Java ME Embedded Client applications.

Note – Please ensure that you have the specific hardware and software environment described in the *Oracle Java Micro Edition Embedded Client Installation Guide*.

3.1 Developer Tools

The Oracle Java ME Embedded Client Emulator allows application developers to develop, build and test Java ME Embedded client applications in an integrated IDE environment such as NetBeans. Developers can work with command line tools directly, rather than using an IDE interface.

The emulator runs on Windows or Linux/x86 systems. It is available when you use the OJEC platform in NetBeans or Eclipse, or it can be launched from the command line. For an overview of command line options, see the *Oracle Java Micro Edition Embedded Client Installation Guide*.

- NetBeans offers full integration with the SDK. You can build applications, run them on the emulator, and use the profiling and debugging features to test your application.
- Eclipse integration is limited to application development and emulation.

The *Oracle Java Micro Edition Embedded Client Installation Guide* describes how to configure these IDEs to work with the SDK.

3.2 Finding API Documentation

The API documentation for JSRs supported in this release can be found in the SDK installation at `/usr/local/Oracle_JavaME_Embedded_Client/1.0/docs` or `C:\Program Files\Oracle\Oracle JavaME Embedded Client\1.0\docs`.

CDC, FP, and PBP are found online here:

`http://download.oracle.com/javame/embedded.html`

If you prefer to download the optional package documentation and install it locally, consult the Java Community Process (JCP™) program web site:

`http://jcp.org/en/jsr/detail?id=218`

3.3 Application Life Cycles

The Oracle Java ME Embedded Client supports two application life cycle models. The basic model is the traditional `main()` application model and the more TV-centric model is an `Xlet` (`javax.microedition.xlet.Xlet`).

The traditional application model is quite simple: load a class, invoke its `main()` method, and wait until all non-background threads terminate or `System.exit()` is called. In version 1.0 the available target is headless, so applications for the target must use this model. You can see examples of this model in the Oracle Java Micro Edition Embedded Client Installation Guide.

For many applications, this model allows too little control over the application's behavior. Personal Basis Profile defines its own application model, similar in many ways to the MIDlet model. The `Xlet` model has been borrowed from the Java TV API, where it is used to control application life cycles in set-top boxes. The model's two key elements are the `Xlet` and `XletContext` interfaces, both found in the `javax.microedition.xlet` package. The application's main class implements the `Xlet` interface, which defines event methods for the system to invoke. The `XletContext` interface defines callback methods through which an application can obtain information about its operating environment.

3.4 The Xlet Life Cycle

As most Java platform programmers are familiar with the main application model, this section briefly describes the `Xlet` life cycle.

An `Xlet` implements the `javax.microedition.xlet.Xlet` interface, which declares four life-cycle notification methods: `initXlet()`, `startXlet()`, `pauseXlet()`, and `destroyXlet()`. Note that, unlike applets or MIDlets, an `Xlet` does not have to extend any particular class.

Like applets, `Xlet`s have four possible states:

- *loaded*: The `Xlet` instance has been constructed, but no initialization has yet occurred.
- *paused*: The `Xlet` has been initialized but is inactive.
- *active*: The `Xlet` is active.
- *destroyed*: The `Xlet` has been terminated and is ready for garbage collection.

When the Application Management System (AMS) creates an `Xlet`, it starts in the loaded state. Soon after construction, the AMS invokes the `Xlet`'s `initXlet()` method, passing the `XletContext` that the `Xlet` must use to interact with its operating context (this parameter is necessary because the `Xlet` does not extend a specific base class). After initialization, the `Xlet` changes to the paused state.

At this point, an `Xlet` behaves more like a MIDlet than an applet. At some point, the AMS activates the `Xlet` and invokes its `startXlet()` method. The `Xlet` activates its user interface, obtains the system resources it needs to function properly, then shifts to the active state.

Deactivation occurs when the `Xlet`'s state changes from active to paused. When the AMS deactivates the `Xlet`, it invokes the `Xlet`'s `pauseXlet()` method. The `Xlet` frees as many resources as possible. If the `Xlet` deactivates itself, however, the `pauseXlet()` method is not invoked.

An `Xlet` can change to the destroyed state at any time. If the AMS destroys the `Xlet`, it invokes the `destroyXlet()` method. Like MIDlets, `Xlet`s can sometimes abort their destruction by throwing an exception. If an `Xlet` destroys itself, `destroyXlet()` is not invoked.

3.5 A First Xlet

With the Oracle Java ME Embedded Client Emulator installed, you can write, build, and execute applications using the PBP 1.1 API.

In an application, the device decides what portion of the screen it wants to give to your application.

Note – If you use the SDK on Windows the emulator will mimic the device's screen size behavior. On Linux, although the SDK does not adjust the window size. In both cases, the window size is rendered correctly on the device.

This is supplied as an AWT `Container`. You can get at it from the `XletContext` that the device passes to your application's `initXlet()` method.

You now know enough about `Xlets` to create your first application. Below is the source code for `HelloXlet.java`, which simply displays a message on the screen and exits when you press any key.

```
package helloxlet;

public class HelloXlet implements javax.microedition.xlet.Xlet {
    /**
     * Default constructor without arguments should be.
     */
    public HelloXlet() {
    }

    /**
     * Put your initialization here, not in constructor.
     * If something goes wrong, XletStateChangeException should be thrown.
     */
    public void initXlet(javax.microedition.xlet.XletContext context)
        throws javax.microedition.xlet.XletStateChangeException {
        // TODO implement initialization
    }

    /**
     * Xlet will be started here.
     * If something goes wrong, XletStateChangeException should be thrown.
     */
    public void startXlet() throws
javax.microedition.xlet.XletStateChangeException {
        // TODO implement
    }
```

```
    /**
     * Free resources, stop unnecessary threads, remove itself from the screen.
     */
    public void pauseXlet() {
        // TODO implement
    }

    /**
     * Destroy your xlet here. If parameter is false, you can try to not
     * destroy xlet by throwing an XletStateChangeException
     */
    public void destroyXlet(boolean unconditional)
        throws javax.microedition.xlet.XletStateChangeException {
        // TODO implement
    }
}
```

Most of the excitement happens in the `initXlet()` method, which retrieves the `Xlet`'s visual space and saves it in the member variable `context`. Then an inner key listener is registered to exit the application when a key is pressed.

When the `Xlet` is started or paused, `context` is shown or hidden as appropriate. The call to `requestFocus()` ensures that key events are delivered to the `Xlet`'s container, which is where the listener is registered.

This `Xlet` uses a very simple `Component` subclass to supply its visual content. The default Main template provides sample code that will work with `HelloXlet.java`.

```
package helloxlet;
import javax.microedition.xlet.*;
import java.awt.BorderLayout;
import java.awt.Component;
import java.awt.Container;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Font;

// Create the Main class.
public class Main extends Component implements Xlet {
    private Container rootContainer;
    private Font font;

    // Initialize the xlet.
    public void initXlet(XletContext context) {
        log("initXlet called");
        // Setup the default container
        // This is similar to standard JDK programming,
```

```
        // except you need to get the container first.
        // XletContext.getContainer gets the parent
        // container for the Xlet to put its AWT components in.
        // and location is arbitrary, so needs to be set.
        // Calling setVisible(true) make the container visible.
        try {
            rootContainer = context.getContainer();
            rootContainer.setSize(400, 300);
            rootContainer.setLayout(new BorderLayout());
            rootContainer.setLocation(0, 0);
            rootContainer.add("North", this);
            rootContainer.validate();
            font = new Font("SansSerif", Font.BOLD, 20);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    // Start the xlet.
    public void startXlet() {
        log("startXlet called");
        //make the container visible
        rootContainer.setVisible(true);
    }

    // Pause the xlet
    public void pauseXlet() {
        log("pauseXlet called");
        //make the container invisible
        rootContainer.setVisible(false);
    }

    // Destroy the xlet
    public void destroyXlet(boolean unconditional) {
        log("destroyXlet called");
        //some cleanup for the xlet..
        rootContainer.remove(this);
    }

    void log(String s) {
        System.out.println("SimpleXlet: " + s);
    }

    public void paint(Graphics g) {
        int w = getSize().width;
        int h = getSize().height;
        g.setColor(Color.blue);
        g.fill3DRect(0, 0, w - 1, h - 1, true);
        g.setColor(Color.white);
```

```
        g.setFont(font);
        g.drawString("Hello Java World", 20, 150);
    }

    public Dimension getMinimumSize() {
        return new Dimension(400, 300);
    }

    public Dimension getPreferredSize() {
        return getMinimumSize();
    }
}
```

▼ Use NetBeans to Create HelloXlet

Follow these steps to create a Netbeans project from two source files. The steps are very similar for Eclipse.

1. **Create a new project.**

   The project wizard opens.

2. **On the New Project page, select the category Java ME and the CDC Application project, and click Next.**

3. **On the Name and Location page, specify the name HelloXlet. Check Create Main Class and click Next.**

4. **On the Select Platform page, select Oracle Java Micro Edition Embedded ClientEmulator as the platform. Choose any device, and for the profile, select JEC-1.0. Click Finish.**

   The Main file is created automatically. Main uses the container in the `HelloXlet.java` file discussed earlier. The application prints the message "Hello Java World" in white type in a window with a blue background.

5. **Build the project and run it. You'll see the emulator window pop up with the "Hello Java World" message. Hit any key to exit.**

   Note, on Windows the size of the window in the emulator is determined by the device selection. On Linux the emulator cannot set the window size, but it will be displayed properly on the device.

3.6 Components and Layouts

Oracle Java ME Embedded Client does not include a user interface (UI) toolkit. You don't get any buttons, lists, combo boxes, or any other user interface components. This is both challenging and liberating.

You're not left out in the cold entirely. Oracle Java ME Embedded Client does include most of the infrastructure you need to build a UI toolkit, just not the actual components (widgets) themselves.

If you've done any work with AWT or Swing on the desktop, you'll feel right at home. In essence, Oracle Java ME Embedded Client provides AWT without the usual components.

The fundamental structure is defined by `java.awt.Component` and `java.awt.Container`. A `Component` is something that shows up on the screen, like a button, text field, or movie. A `Container` is simply a visual group of `Components`. `Container` has a very useful concept, the layout manager which is an object that places the Components of a Container in a certain way. Oracle Java Platform ME Embedded Client includes several useful layout managers.

Most user interfaces are composed of a variety of `Components`, `Containers`, and `LayoutManagers`. This kind of composition is possible because a `Container` can hold other `Containers`.

The `XletContext` passed to your application's `initXlet()` method has a reference to a `Container`. Your application builds its entire user interface on this `Container`.

3.7 Creating a User Interface

You can use someone else's UI toolkit, or you can create your own.

This section describes how to create a simple UI button from scratch using Xlets and containers. The first step is to create a subclass of `Component`, which provides lots of useful plumbing.

Your subclass of `Component` has to provide implementations for a few methods to appear on the screen:

- Your implementations of `getPreferredSize()`, `getMinimumSize()`, and `getMaximumSize()` help the device in laying out your component in its parent container.

- The body of your component's `paint()` method determines how the component shows itself on the screen.

Below is a simple class, `DTVButton`, which shows a rudimentary round-cornered rectangular button with a text label. The button exists in one of three states, either normal, focussed, or pressed, which determine the colors that are used to display the button.

`DTVButton` uses its text label to calculate how big it wants to be. It returns the same size for the preferred, minimum, and maximum sizes.

```
package dtvui;

import java.awt.*;

public class DTVButton extends Component {
  private static final int kPad = 6;

  public static final int kNormal = 0;
  public static final int kFocus = 1;
  public static final int kPressed = 2;

  private static Color sBG = Color.darkGray;
  private static Color sFG = Color.gray;
  private static Color sBGH = Color.pink.darker();
  private static Color sFGH = Color.red.darker();
  private static Color sBGC = Color.pink;
  private static Color sFGC = Color.red;

  private String mLabel;
  private Dimension mPreferredSize;
  private int mState;

  public DTVButton(String label) {
    setLabel(label);
    mState = kNormal;
  }

  public void setLabel(String label) { mLabel = label; }
  public String getLabel() { return mLabel; }

  private void calculate() {
    Graphics g = getGraphics();
    FontMetrics fm = g.getFontMetrics();
    int tw = fm.stringWidth(mLabel) + kPad * 2;
    int th = fm.getHeight() + kPad * 2;
    mPreferredSize = new Dimension(tw, th);
  }
```

```
  public void setState(int state) {
    boolean dirty = (mState != state);
    mState = state;
    if (dirty) repaint();
  }

  public int getState() { return mState; }

  public void paint(Graphics g) {
    int w = getWidth();
    int h = getHeight();

    Color bg, fg;

    switch(getState()) {
      case kFocus:   bg = sBGH; fg = sFGH; break;
      case kPressed: bg = sBGC; fg = sFGC; break;
      case kNormal:
      default:       bg = sBG;  fg = sFG;  break;
    }

    g.setColor(bg);
    g.fillRoundRect(0, 0, w, h, kPad, kPad);

    g.setColor(fg);
    g.drawRoundRect(0, 0, w - 1, h - 1, kPad, kPad);

    FontMetrics fm = g.getFontMetrics();
    int tw = (int)mPreferredSize.width;
    int th = (int)mPreferredSize.height;

    g.drawString(mLabel,
        (w - tw) / 2 + kPad,
        (h - 1 - th) / 2 + fm.getAscent() + kPad);
  }

  public Dimension getPreferredSize() {
    if (mPreferredSize == null) calculate();
    return mPreferredSize;
  }

  public Dimension getMinimumSize() { return getPreferredSize(); }
  public Dimension getMaximumSize() { return getPreferredSize(); }

}
```

Now you can create one or more `DTVButtons` and show them on the screen, but they can't do anything unless you add some event handling.

The `DTVButtonGroup` class, below, serves to group buttons and handles key events. It enables the user to navigate through a list of buttons using the arrow keys. A button can be pressed with the Enter or select key. In this case, an event is fired to a listener object.

```
package dtvui;

import java.awt.event.*;
import java.util.*;

public class DTVButtonGroup implements KeyListener {
  private int mFocus;
  private List mButtonList;

  private DTVButtonListener mListener;

  public DTVButtonGroup() {
    mFocus = 0;
    mButtonList = new ArrayList();
  }

  public void add(DTVButton button) {
    mButtonList.add(button);
    if (mButtonList.size() == 1)
      button.setState(DTVButton.kFocus);
  }

  public void setDTVButtonListener(DTVButtonListener listener) {
    mListener = listener;
  }

  // KeyListener methods.

  public void keyPressed(KeyEvent ke) {
    int newState = DTVButton.kFocus;

    DTVButton old = (DTVButton)mButtonList.get(mFocus);
    old.setState(DTVButton.kNormal);

    int code = ke.getKeyCode();

    if (code == KeyEvent.VK_UP || code == KeyEvent.VK_LEFT) {
      mFocus--;
      if (mFocus < 0)
        mFocus = mButtonList.size() - 1;
    }
    else if (code == KeyEvent.VK_DOWN || code == KeyEvent.VK_RIGHT) {
      mFocus++;
      if (mFocus >= mButtonList.size())
```

```
        mFocus = 0;
    }
    else if (ke.getKeyCode() == KeyEvent.VK_ENTER) {
      newState = DTVButton.kPressed;
    }

    DTVButton b = (DTVButton)mButtonList.get(mFocus);
    b.setState(newState);
  }

  public void keyReleased(KeyEvent ke) {
    int code = ke.getKeyCode();

    if (code == KeyEvent.VK_ENTER) {
      DTVButton b = (DTVButton)mButtonList.get(mFocus);
      b.setState(DTVButton.kFocus);
      if (mListener != null)
        mListener.pressed(b);
    }
  }

  public void keyTyped(KeyEvent ke) {}
}
```

Typical usage of these classes is to create some `DTVButtons`, add them to a group and add them to a visual `Container`, and register a listener for the group. The listener interface is very simple:

```
package dtvui;

public interface DTVButtonListener {
  public void pressed(DTVButton button);
}
```

The `Main.java` file sets up the containers and manages the Xlet status. This is similar to JDK programming, except you must get the container first. `context.getContainer` gets the parent container where the Xlet puts its AWT components. The size and location is arbitrary, so the values must be set. Calling `setVisible(true)` makes the container visible.

```
package dtvui;

import javax.microedition.xlet.*;
import java.awt.BorderLayout;
import java.awt.Component;
import java.awt.Container;
import java.awt.Color;
import java.awt.Dimension;
```

```
import java.awt.Graphics;
import java.awt.Font;
// Create the Main class.
public class Main extends Component implements Xlet, DTVButtonListener{

  private Container rootContainer;
  private Font font;
  private DTVButtonGroup group;
  private DTVButton b1, b2, b3;

    // Initialize the xlet.
    public void initXlet(XletContext context) {
        log("initXlet called");
        // Setup the default container
        try {
            rootContainer = context.getContainer();
            rootContainer.setSize(600, 600);
            rootContainer.setLayout(new BorderLayout());
            rootContainer.setLocation(0, 0);
            rootContainer.add("North", this);
            font = new Font("SansSerif", Font.BOLD, 20);
            group = new DTVButtonGroup();
            rootContainer.addKeyListener(group);

            b1 = new DTVButton("Uno");
            group.add(b1);
            b1.setVisible(true);
            rootContainer.add("West", b1);

            b2 = new DTVButton("Due");
            b2.setVisible(true);
            group.add(b2);
            rootContainer.add("South", b2);

            b3 = new DTVButton("Tre");
            b3.setVisible(true);
            group.add(b3);
            rootContainer.add("East", b3);

            group.setDTVButtonListener(this);
            rootContainer.validate();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    // Start the xlet.
    public void startXlet() {
        log("startXlet called");
```

```
        //make the container visible
        rootContainer.setVisible(true);
        rootContainer.update(rootContainer.getGraphics());
            b1.setVisible(true);
            b2.setVisible(true);
            b3.setVisible(true);
    }

    // Pause the xlet
    public void pauseXlet() {
        log("pauseXlet called");
        //make the container invisible
        rootContainer.setVisible(false);
    }

    // Destroy the xlet
    public void destroyXlet(boolean unconditional) {
        log("destroyXlet called");
        //some cleanup for the xlet
        rootContainer.remove(this);
    }

    void log(String s) {
        System.out.println("SimpleXlet: " + s);
    }

    public void paint(Graphics g) {
        log("Main Paint");
        int w = getSize().width;
        int h = getSize().height;
        g.setColor(Color.blue);
        g.fill3DRect(0, 0, w - 1, h - 1, true);
        g.setColor(Color.white);
        g.setFont(font);
        g.drawString("Button Layout Example", 20, 150);
    }

    public Dimension getMinimumSize() {
        return new Dimension(400, 300);
    }

    public Dimension getPreferredSize() {
        return getMinimumSize();
    }

    public void pressed(DTVButton button) {
        group.keyPressed(null);
    }
}
```

Examine the source code to see how you can respond to button presses to make interactive applications.

3.8 Profiling and Debugging with NetBeans

To support debugging and profiling for the Oracle Java ME Embedded Client, NetBeans establishes a connection between the IDE and the target device. The target can be an embedded device running the Oracle Java ME Embedded Client, or the emulator available with the Oracle Java ME Embedded Client SDK, which is typically installed on the same machine as the NetBeans IDE. In both debugging and profiling, the CVM sets up a server over a server socket on the target device, and NetBeans connects as a client. In the case of the profiler, the methods to be profiled are rewritten with extra bytecode so that profiling can take place. The VM is started with special flags. The handshaking must take place before you launch the application.

Note – The NetBeans profiling features Take Snapshot and Take Heap Dump are not supported in this release. Also, profiling of ROMized system classes is not supported.

3.8.1 Profiling and Debugging VMs

If you are profiling or debugging locally (using the emulator) the installation includes everything required. If you are using a remote target, download and install the ARM runtime and mount it from the target. This process is described in the *Oracle Java Micro Edition Embedded Client Installation Guide*.

In the SDK the CVM executable is found at:

InstallDir`/Oracle_JavaME_Embedded_Client/1.0/emulator-platform/bin/cvm`

On the target the distribution offers the JVMTI (JVM tooling interface) and Production binaries. Debugging and profiling requires the JVMTI virtual machine. This is found at:

InstallDir`/Oracle_JavaME_Embedded_Client/1.0/binaries/jvmti/bin/cvm`

To profile or debug you launch the JVMTI version of the CVM in a special "server" mode on the target. NetBeans attaches to this as a profiling or debugging "client".

Both debugging and profiling work this way, with the server on the target VM and the client running in NetBeans.

3.8.2 Profiling Prerequisites

To profile you must know the full IP address or host name of the target device.

Note – If are profiling locally (on the SDK installation host) you must still supply an IP address when you are running the SDK and the VM on the host machine. Using Localhost or 127.0.0.1 does not work.

The following assets are used in profiling and must be available on the target device.

- The profiler interface library.

  This can be in the form of a shared object file (`libprofilerinterface.so` on Linux or a `.dll` file on Windows).
- The `jfluid-server.jar` file.
- The `jfluid-server-cvm.jar` file.
- Calibration data. Produced when you run the calibration script (see Step 1 in "Profiling a Test Application" on page 3-16).

▼ Profiling a Test Application

This procedure describes profiling an application running on a local host, but the process is basically the same with a remote target. For information on installing the OJEC stack on a client and placing applications on a client, see the *Installation Guide*.

To see profiling results your application must create sufficient data. Use your own program, or create an Oracle Java ME Embedded Client project named Test which uses the following `Main.java` source code:

```
package test;
public class Main {
    public static void main(String[] args) {
        int i, j, k;
        boolean prime;
        String demoString;
        i = 98890000;
        while (true) {
            prime = true;
            for (j=2; (j*j) <= i; j++) {
                k = i / j;
                if ((k*j)==i) {
                    demoString = new String(String.valueOf(i));
                    prime = false;
                }
```

```
            }
            if (prime) try {
                System.out.println(i + " is prime");
                Thread.sleep(100);
                demoString = null;
            } catch (Exception e) {
                //Ignore
            }
            i++;
        }
    }

}
```

Follow these steps to launch a profiling session:

1. **Generate calibration data**

   In the profiling directory, run the calibration script. You only need to do this once.

   Linux:

```
/usr/local/Oracle_JavaME_Embedded_Client/1.0/emulator-platform/
lib/profiler/calibrate.sh
```

   Linux Target:

```
installdir/Oracle_JavaME_Embedded_Client/1.0/binaries/jvmti/
lib/profiler/calibrate.sh
```

   Windows XP:

```
\Program Files\Oracle\Oracle JavaME Embedded Client\1.0\
emulator-platform\lib\profiler\calibrate.bat
```

   On Linux the calibration data is stored in your home directory in a directory named: `.nbprofiler`. On Windows, the calibration data is stored in: `Documents and Settings\`*User*`\.nbprofiler`.

2. **Set the NetBeans profiling settings:**

   a. **In the "Attach to" dropdown choose "External Application" (the default).**

   b. **Select the profiling type (Monitor, CPU, or Memory).**

      In this example we chose memory.

   c. **Next to Attach Mode, click the `define` link to open the Attach Wizard.**

   d. **Choose Application in the drop-down menu.**

e. **Choose the Remote attach method and Direct invocation. Click Next.**

f. **After the review screen, click Finish.**

These settings will persist. You do not have to repeat the define steps. You can change the profiling type each time you attach the profiler.

3. **Launch the project from the command line.**

This example refers to a project named Test and the commands are issued from the `emulator-platform` directory.

Linux:

```
./bin/cvm -Xms32m
-agentpath:./lib/libprofilerinterface.so=./lib/profiler/lib,5140
-Xbootclasspath/p:./lib/profiler/lib/jfluid-server-cvm.jar
-cp path-to-NetBeansProjects/Test/dist/Test.jar test.Main
```

Linux Target:

On the target the directory structure is slightly different, as described in Section 3.8.1 "Profiling and Debugging VMs" on page 3-15. The call is much the same but you use the VM found in:
`/Oracle_JavaME_Embedded_Client/binaries/jvmti/bin`.

Windows:

```
bin\emulator.exe -Xms32m
-agentpath:bin\profilerinterface.dll=lib\profiler\lib,5140
-Xbootclasspath/p:lib\profiler\lib\jfluid-server-cvm.jar
-cp path-to-NetBeansProjects\Test\dist\Test.jar test.Main
```

Note – Xms is an optional argument to increase the heap size

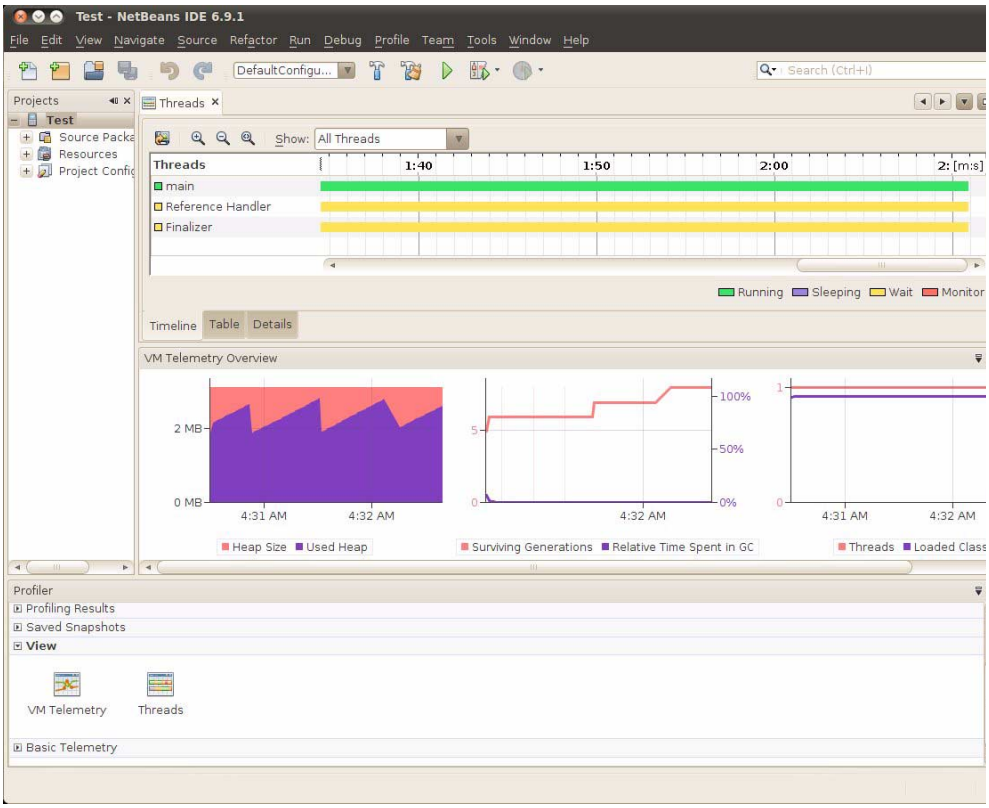
The CVM launches the application and waits for the profiler to connect. You see:

```
Profiler Agent: Initializing
Profiler Agent: Options: >./lib/profiler/lib,5140
Profiler Agent: Initialized successfully 15.28
Profiler Agent: Waiting for connection on port 5140
```

4. **Attach NetBeans to the waiting application.**

Select Profile > Attach Profiler. You can change the type of profiling (Monitor, CPU, or Memory) and set type-specific options before selecting Attach.

To see the results, select Window > Profiling and choose the views you want displayed. For example, Telemetry Overview. These options are fully described in the NetBeans online help.



The Save Current View to Image feature takes a graphic snapshot of the current view :

Thread	Running	Sleeping	Wait	Monitor
main	1:51.064 (99.5%)	0.502 (0.4%)	0.0 (0.0%)	0.0 (0.
Reference Handler	0.0 (0.0%)	0.0 (0.0%)	1:51.566 (100.0%)	0.0 (0.
Finalizer	0.0 (0.0%)	0.0 (0.0%)	1:51.566 (100.0%)	0.0 (0.

If you are finished looking at the results and the application is still running, stop it from the command line with CTRL-C.

▼ Local Debugging with NetBeans

You can debug applications on the SDK host using standard NetBeans tools. Although it is not the same as debugging on the target, local debugging is worth doing because it is simple and it's useful for fixing generic problems unrelated to device functionality.

Note – Only interpreted code can be debugged.

1. **Make your project the Main project.**
2. **To set a breakpoint right click on a line of code and choose Toggle Line Breakpoint.**

3. **Click on the debug icon in the tool bar, and select Debug Main Project to run the project in debug mode.**

The application runs, stopping at any breakpoints. You can continue or step through the code. See the NetBeans help for details on the options in the Debug menu. To end the session, choose Debug > Finish Debugger Session.

▼ Remote Debugging with NetBeans

To debug an application running on a target device you must configure a connection between the NetBeans IDE and the device. When the application is launched you must invoke an agent that can communicate the JDWP protocol so that debugging can take place.

1. **Set any breakpoints in your application, compile it, and transfer the built project to the target device (or mount it, as discussed in the *Installation Guide*).**

2. **On the target, start the application from the command line using the jvmti binary and debug options.**

   Before attempting to run the debugger on your remote system, verify that you can run the application without debugging. When your application runs successfully, add the following parameters to the command you are using to launch the application:

```
-Xdebug -agentlib:jdwp=transport=dt_socket,address=53955,server=y,suspend=y
```

   Note: the `agentlib:jdwp` arguments are separated with commas; do not insert spaces in the `jdwp` argument string.

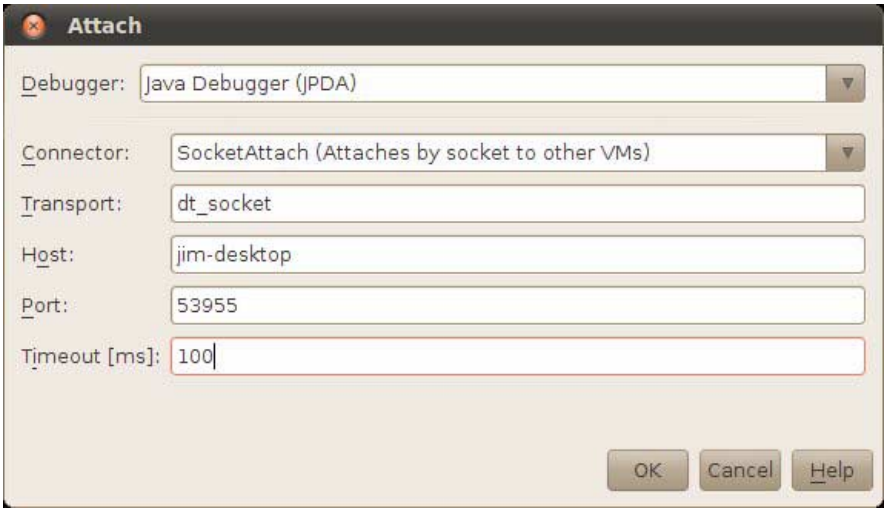
```
InstallDir/Oracle_JavaME_Embedded_Client/binaries/jvmti/bin/cvm -Xdebug
-agentlib:jdwp=transport=dt_socket,address=53955,server=y,suspend=y
-cp YourProjectDir/Test/build/compiled test.Main
```

   When the connection is formed you see the following message in the console:

```
Listening for transport dt_socket at address: 53955
```

3. **Click on the debug icon in the tool bar, and select Attach Debugger.**

This opens the Attach dialog window. The settings should be similar to those shown below.



The host can be an IP address or a server name, for example, `server.sfbay.sun.com`.

The port value 53955 is the default. You can change the port value as long as it matches the value of the address argument you specified in Step 2.

Adjust the Timeout value as you see fit.

Click OK. The application loads and runs until it reaches a breakpoint.

The output appears in a Debugger Console tab in the Output window.

Consult the NetBeans help for descriptions of the NetBeans debugging options.

PART II Java Virtual Machine Reference

Part two describes attributes of the Java Virtual Machine.

Java Virtual Machine Capabilities:

Discusses developer interaction with the JVM.

Internal Memory Allocator:

Describes JVM memory management.

Threading:

Discusses the threading behavior of the JVM.

Internationalization:

Discusses ROMized and Non-ROMized locales and character converters.

External PBP Porting Layer Plugin:

Discusses the JGFX graphics framework.

CHAPTER 4

Java Virtual Machine Capabilities

The Oracle Java ME Embedded Client Java VM implementation fully supports the Java Native Interface (JNI) specification version 1.2.

4.1 Resource Registry for Cleanup Resources

This feature allows the application to control the resources consumed by the PBP graphic layer by setting a `ResourceRegistry` implementation and have the ability to track which resources are currently consumed and have the ability to dispose them at any time.

The class `misc.bluray.jvmbi.ResourceRegistry` defines APIs to register and unregister objects with a disposer to dispose them. The stack has a singleton resource registry available. By default, the registry does nothing. However, the application can set its own implementation of the registry using the following API:

```
ResourceRegistry.setInstance(ResourceRegistry registry)
```

The PBP layer implementation registers the following objects with their disposers to the available registry:

- `java.awt.Graphics` objects
- `java.awt.Image` objects
- `sun.xlet.ixc.Worker` threads
- `javax.microedition.xlet.XletContext` objects
- Names of bound `javax.microedition.xlet.XletContext` objects
- Event listeners of all types: window, key, mouse, etcetera.
- `java.awt` event dispatch thread.
- Timer threads

4.2 Override Runtime Properties

This feature can be used for Java applications to override Java system property values at runtime. This is useful in cases where the Java property value can vary according to the context in which it is queried.

▼ Using Dynamic Properties

Let us consider an example of property `user.dir`. When running in a multiple Xlet environment, the value of this property can be different for each Xlet. The Java application would like to control the value of this property and provide the correct value according the context of the thread querying the value.

Dynamic properties are implemented using an interface `com.sun.cdc.config.PropertyProvider` and a control manager called `com.sun.cdc.config.DynamicProperties`.

Follow these steps to use dynamic properties.

1. Implement the PropertyProvider interface:

```
/**
 * Returns the current value of the dynamic
 * propertycorresponding to this
 * <code>PropertyProvider</code>.
 *
 * @param key key for the dynamic property.
 * @param fromCache indicates whether property value should
 * be taken from internal cache. It can be ignored if properties
 * caching is not supported by underlying implementation.
 * @return the value that is returned by
 * <code>System.getProperty()</code> call for the corresponding
 * dynamic property.
 */
public String getValue(String key, boolean fromCache);

/**
 * Tells underlying implementation to cache values of all the
 * properties corresponding to this particular class. This call
 * can be ignored if property caching is not supported.
 *
 * @return <code>true</code> on success, <code>false</code>
```

```
 * otherwise
 */
public boolean cacheProperties();
```

2. **Register the properties you want to control using** `DynamicProperties.put` **method:**

```
/**
 * Sets the specified <code>PropertyProvider</code> object to be
 * called for dynamic property resolution. This method should be
 * used at the time of properties initialization.
 *
 * @param key key for the dynamic property.
 * @param provider <code>PropertyProvider</code> instance that
 * is used to resolve the dynamic property with the
 * specified key.
 */
public static void put(String key, PropertyProvider provider);
```

3. **In the** `PropertyProvider.getValue` **method implementation you can override the value of the registered properties. The property providers are called before the system properties are used. For any property you registered for (using the** `DynamicProperties.put` **method), if you do not return null you are overriding the system values.**

   This is a sample implementation:

```
import com.sun.cdc.config.DynamicProperties;
import com.sun.cdc.config.PropertyProvider;

public class MyProperties implements PropertyProvider {

    public MyProperties() {
         /*
          * to register a property to be in your control
          */
        DynamicProperties.put("my.prop", this);
    }

    /*
     * Implementation of PropertyProvider interface
     */
    public String getValue(String key, boolean fromCache)
       {
        if (key.equals("my.prop")) {
            String myValue;
            // calculate the value
            ...
```

```
            return myValue;
            }
            return null;
    }

    public boolean cacheProperties() {
        /* not supported */
        return false;
    }
}
```

4.3 Enable/Disable JAR Caching

Caching of an accessed JAR file can be enabled or disabled using the system-wide property `java.net.enableJarCaching`.

Caching enabled by default. If the property is not set, the default value is set to TRUE.

4.4 Heap Monitor

The heap monitor is intended to detect low Java heap availability in order to avoid a Java out of memory exception. A heap status monitor task monitors the Java heap size at a certain periodicity. When the Java heap size grows beyond a certain High Water mark, the following measures are taken:

1. Trigger system garbage collection.
2. Application registers callback to identify which Xlet to kill.
3. Destroy the last Xlet that was initiated.

These actions are repeated in the next period until the heap size falls below the High Watermark.

The default policy is to destroy the latest Xlet initialized (LIFO). Since the motive is to kill Xlets, this Monitor task only starts when an Xlet is initialized. It is canceled when there is no Xlet on the system.

To set your callback to identify which Xlet must be killed, use the following API:

```
JNIEXPORT void JNICALL
JVM_SetDestroyXletIdentifierHook(jobject
(*JVMdestroyXletIdentifier)
    (void));
```

The object returned by your callback is of type `java.lang.Thread`.

You can use any thread that belongs to the Xlet you want to kill. The VM extracts the Xlet application ID (see Section 3.4 "The Xlet Life Cycle" on page 3-3) and use it to kill the Xlet.

Periodicity of monitoring and the **Heap High Watermark** can be specified by the user as command line parameter at VM startup time.

The following are the properties with brief description:

- **Heap High Watermark:**

  `sun.misc.heapHighWaterMark`: It is percentage and not absolute size. So if `-Dsun.misc.heapHighWaterMark=70`, then Xlet killing would be attempted when the Java heap size grows beyond 70% of Max heap size.

- **Heap Check Period:**

  `sun.misc.heapCheckPeriod`: Period specified in milliseconds. Minimum is fixed at 1 sec, so this property cannot be set below 1000ms. If set, it would default to 1000ms.

CHAPTER 5

Internal Memory Allocator

Oracle JVM uses an Internal Memory Allocation and Statistics module for allocating memory from native heap. It provides this functionality:

- Replace ANSI C (`glibc`, `uclibc`), `malloc()`, `calloc()`, `realloc()`, `free()` with the Real Time OS functionality, allowing the JVM to use these APIs on platforms which lack memory allocation features.
- Gather and provide memory usage statistics (amount of memory used, time spent in allocation, etcetera) which are used to fine tune the memory behavior of the application. For example, if you know which buffer sizes are used the most you can define an exact allocation size, achieving minimal internal fragmentation without losing allocation speed.
- Enable full control over the memory allocated by the VM, including deallocating all memory at VM shutdown, monitoring memory allocations per Xlet, and enabling separate pools for separate needs. For example, separate pools can be created to support different types of hardware memory, or to limit a specific pool to specific size.
- See Sections Section 5.1 “Red Zone ” on page 5-1, Section 5.2 “Statistics ” on page 5-2, and Section 5.3 “GC Triggered When Native Memory is Low” on page 5-2 for additional memory allocator features.

5.1 Red Zone

The memory allocator 'RedZone' feature, marks 32 bits right before an allocated area and 32 bits after, and checks these bytes for validity after they are freed. In the debug version a warning is printed if stamps are not valid, as the cause is most likely an out of bounds array write or something similar. This feature also improves the stability of the internal allocator, as it serves to indicate that the internal control structure which

resides in the area may be damaged and should be discarded. The ‘RedZone’ is enabled by default after verification that the performance impact of the 32-bit comparisons is negligible.

5.2 Statistics

The CVM's Memory Allocator module has been enhanced to record and report memory allocations at configurable granularity and size range(s).

This module provides statistics of the following kind for each configured range at the configured granularity:

- Allocation type
- Total number of allocation
- Total number of free
- Maximum outstanding buffers seen
- Time spent in allocation
- Time spent in free

At the Java application exit, the collected statistics would be printed on console.

Note – To avoid performance overhead at runtime, statistics collection is not enabled in the optimized binary. Statistics collection can be enable on request.

5.3 GC Triggered When Native Memory is Low

The VM memory allocator can trigger garbage collection (GC) when system malloc returns NULL. This is useful on devices where the amount of RAM is small and GC must occur to release native resources. In this case, GC is automatically triggered by the VM.

CHAPTER 6

Threading

This chapter discusses basic threading tasks.

6.1 Monitoring Java Threads Life-Cycle

This feature is relevant for applications that invoke the JVM from a shared library (`cvmi.dll` for win32 or `libcvm.so` for Linux).

We have defined two types of callbacks that applications can register:

```
void (*JVMthreadStartHook_G)
     (char* name,int stackSize,char*groupName)
void (*JVMthreadEndHook_G)(char* name)
```

The `JVMthreadStartHook_G` is called when a Java thread starts execution and passes the application information about the thread (name, stack size and Java thread group name). The `JVMthreadEndHook_G` is called when the Java thread exits.

Applications can use these callbacks to monitor the life cycle of Java threads for debugging purposes.

6.2 Registering Callbacks

Use the following API to register your callbacks:

```
JNIEXPORT void JNICALL
```

```
JVM_SetThreadRuntimeHooks( void
    (*JVMthreadStartHook)(char* name,int stackSize, char *groupName),
    void (*JVMthreadEndHook)(char* name));
```

This API is defined in `jni.h`.

(Note: This extension to file `jni.h` can be obtained on request.)

For example:

```
void * myJVMthreadStartHook(char* name, int stackSize,
                            char* groupName)
{
    printf("[JavaThread Created Name=%s,group=%s]\n"
            name, groupName == NULL ? "NULL" : groupName);
}

void myJVMthreadEndHook(char* name)
{
    printf("[JavaThread Destroyed Name=%s]\n", name);
}
```

In your application initialization code write:

```
…
JVM_SetThreadRuntimeHooks(myJVMthreadStartHook,myJVMthreadEndHook);
…
```

6.3 Thread Quota

This feature supports limiting the number of Java threads that are alive at the same time. It is also possible to limit system threads and application threads. The definition of system thread is a thread whose class loader is the system class loader. Any other thread is an application thread.

Usage:

The following properties configure quotas for system threads, application threads, and all threads (application + system) respectively:

- `java.lang.maxNumberOfSystemThreads`
- `java.lang.maxNumberOfApplicationThreads`
- `java.lang.maxNumberOfThreads`

In addition, the property `java.lang.maxThreadControl` should be set to “true” to activate thread quota.

Example:

```
./bin/cvm -Djava.lang.maxThreadControl=true
    -Djava.lang.maxNumberOfSystemThreads=100
    -Djava.lang.maxNumberOfApplicationThreads=50
    -Djava.lang.maxNumberOfThreads=100
    -jar myjar.jar
```

This code limits the number of application threads to 50, and system threads to 100 while limiting the total number of threads to 100.

6.4 Thread.stop Implementation

The original `Thread.stop()` functionality (as well as `Thread.suspend` and `Thread.resume`) was marked as a deprecated method and removed from CDC some time ago.

Re-enabling the original methods would break TCK compliance, as it would mean addition of new public API. In order to achieve the functionality without jeopardizing compliance, a new class, `sun.misc.ThreadControl` has been implemented. This class provides the following static public APIs that act on supplied Thread objects:

```
public static void Thread.stop(Thread)
public static void Thread.suspend(Thread)
public static void Thread.resume(Thread)
```

For example, to use `Thread.stop()`, an application developer could do the following;

```
...
Thread somethread;
<initialise and start thread>
...
ThreadControl.stop(somethread);
```

USAGE:

To activate or deactivate this feature set the build-time flag `CVM_THREAD_SUSPENSION`. The current default is true.

CHAPTER 7

Internationalization

Internationalization support is fully integrated in the CVM classes and packages that provide language or culture-dependent functionality.

Some of the character converters and locales are ROMized in the CVM and some are supported as JAR files to be included at runtime through the runtime parameter `-Xbootclasspath`.

7.1 ROMized Character Converters

The following character converters are supported by default and are ROMized in the CVM:

- ISO8859_1
- UTF8
- ASCII
- UTF16
- Unicode

The list of character converters in the `charsets.jar` file, to be included at run time is given in TABLE 7-1.

The following character converters are compiled in the `charsets.jar` file.

TABLE 7-1 Character Converters

Canonical Name for java.io and java.lang API	Description
Ascii	American Standard Code for Information Interchange
Cp1250	Windows Eastern European
Cp1251	Windows Cyrillic
Cp1252	Windows Latin-1
Cp1253	Windows Greek
Cp1254	Windows Turkish
Cp1257	Windows Baltic
ISO8859_2	Latin Alphabet No. 2
ISO8859_4	Latin Alphabet No. 4
ISO8859_5	Latin/Cyrillic Alphabet
ISO8859_7	Latin/Greek Alphabet
ISO8859_9	Latin Alphabet No. 5
ISO8859_13	Latin Alphabet No. 7
ISO8859_15	Latin Alphabet No. 9
KOI8_R	KOI8-R, Russian
Cp1255	Windows Hebrew
Cp1256	Windows Arabic
Cp1258	Windows Vietnamese
ISO8859_3	Latin Alphabet No. 3
ISO8859_6	Latin/Arabic Alphabet
ISO8859_8	Latin/Hebrew Alphabet
MS932	Windows Japanese
EUC_JP	JISX 0201, 0208 and 0212, EUC encoding Japanese
EUC_JP_LINUX	JISX 0201, 0208, EUC encoding Japanese
SJIS	Shift-JIS, Japanese
ISO2022JP	JIS X 0201, 0208, in ISO 2022 form, Japanese
MS936	Windows Simplified Chinese
GB18030	Simplified Chinese, PRC standard

TABLE 7-1 Character Converters *(Continued) (Continued)*

Canonical Name for java.io and java.lang API	Description
EUC_CN	GB2312, EUC encoding, Simplified Chinese
GBK	GBK, Simplified Chinese
ISCII91	ISCII91 encoding of Indic scripts
MS949	Windows Korean
EUC_KR	KS C 5601, EUC encoding, Korean
ISO2022KR	ISO 2022 KR, Korean
MS950	Windows Traditional Chinese
MS950_HKSCS	Windows Traditional Chinese with Hong Kong extensions
EUC_TW	CNS11643 (Plane 1-3), EUC encoding, Traditional Chinese
Big5	Big5, Traditional Chinese
Big5_HKSCS	Big5 with Hong Kong extensions, Traditional Chinese
TIS620	TIS620, Thai
Big5_Solaris	Big5 with seven additional Hanzi ideograph character mappings for the Solaris zh_TW.BIG5 locale
Cp037	USA, Canada (Bilingual, French), Netherlands, Portugal, Brazil, Australia
Cp273	IBM Austria, Germany
Cp277	IBM Denmark, Norway
Cp278	IBM Finland, Sweden
Cp280	IBM Italy
Cp284	IBM Catalan/Spain, Spanish Latin America
Cp285	IBM United Kingdom, Ireland
Cp297	IBM France
Cp420	IBM Arabic
Cp424	IBM Hebrew
Cp437	MS-DOS United States, Australia, New Zealand, South Africa
Cp500	EBCDIC 500V1
Cp737	PC Greek
Cp775	PC Baltic
Cp838	IBM Thailand extended SBCS

TABLE 7-1 Character Converters *(Continued) (Continued)*

Canonical Name for java.io and java.lang API	Description
Cp850	MS-DOS Latin-1
Cp852	MS-DOS Latin-2
Cp855	IBM Cyrillic
Cp856	IBM Hebrew
Cp857	IBM Turkish
Cp858	Variant of Cp850 with Euro character
Cp860	MS-DOS Portuguese
Cp861	MS-DOS Icelandic
Cp862	PC Hebrew
Cp863	MS-DOS Canadian French
Cp864	PC Arabic
Cp865	MS-DOS Nordic
Cp866	MS-DOS Russian
Cp868	MS-DOS Pakistan
Cp869	IBM Modern Greek
Cp870	IBM Multilingual Latin-2
Cp871	IBM Iceland
Cp874	IBM Thai
Cp875	IBM Greek
Cp918	IBM Pakistan (Urdu)
Cp921	IBM Latvia, Lithuania (AIX, DOS)
Cp922	IBM Estonia (AIX, DOS)
Cp930	Japanese Katakana-Kanji mixed with 4370 UDC, superset of 5026
Cp933	Korean Mixed with 1880 UDC, superset of 5029
Cp935	Simplified Chinese Host mixed with 1880 UDC, superset of 5031
Cp937	Traditional Chinese Host mixed with 6204 UDC, superset of 5033
Cp939	Japanese Latin Kanji mixed with 4370 UDC, superset of 5035
Cp942	IBM OS/2 Japanese, superset of Cp932

TABLE 7-1 Character Converters *(Continued) (Continued)*

Canonical Name for java.io and java.lang API	Description
Cp942C	Variant of Cp942
Cp943	IBM OS/2 Japanese, superset of Cp932 and Shift-JIS
Cp943C	Variant of Cp943
Cp948	OS/2 Chinese (Taiwan) superset of 938
Cp949	PC Korean
Cp949C	Variant of Cp949
Cp950	PC Chinese (Hong Kong, Taiwan)
Cp964	AIX Chinese (Taiwan)
Cp970	AIX Korean
Cp1006	IBM AIX Pakistan (Urdu)
Cp1025	IBM Multilingual Cyrillic: Bulgaria, Bosnia, Herzegovinia, Macedonia (FYR)
Cp1026	IBM Latin-5, Turkey
Cp1046	IBM Arabic – Windows
Cp1047	Latin-1 character set for EBCDIC hosts
Cp1097	IBM Iran (Farsi)/Persian
Cp1098	IBM Iran (Farsi)/Persian (PC)
Cp1112	IBM Latvia, Lithuania
Cp1122	IBM Estonia
Cp1123	IBM Ukraine
Cp1124	IBM AIX Ukraine
Cp1140	Variant of Cp037 with Euro character
Cp1141	Variant of Cp273 with Euro character
Cp1142	Variant of Cp277 with Euro character
Cp1143	Variant of Cp278 with Euro character
Cp1144	Variant of Cp280 with Euro character
Cp1145	Variant of Cp284 with Euro character
Cp1146	Variant of Cp285 with Euro character
Cp1147	Variant of Cp297 with Euro character
Cp1148	Variant of Cp500 with Euro character

TABLE 7-1 Character Converters *(Continued) (Continued)*

Canonical Name for java.io and java.lang API	Description
Cp1149	Variant of Cp871 with Euro character
Cp1381	IBM OS/2, DOS People's Republic of China (PRC)
Cp1383	IBM AIX People's Republic of China (PRC)
Cp33722	IBM-eucJP - Japanese (superset of 5050)
ISO2022_CN_CNS	CNS11643 in ISO 2022 CN form, Traditional Chinese (conversion from Unicode only)
ISO2022_CN_GB	GB2312 in ISO 2022 CN form, Simplified Chinese (conversion from Unicode only)
JISAutoDetect	Detects and converts from Shift-JIS, EUC-JP, ISO 2022 JP (conversion to Unicode only)
MS874	Windows Thai
MacArabic	Macintosh Arabic
MacCentralEurope	Macintosh Latin-2
MacCroatian	Macintosh Croatian
MacCyrillic	Macintosh Cyrillic
MacDingbat	Macintosh Dingbat
MacGreek	Macintosh Greek
MacHebrew	Macintosh Hebrew
MacIceland	Macintosh Iceland
MacRoman	Macintosh Roman
MacRomania	Macintosh Romania
MacSymbol	Macintosh Symbol
MacThai	Macintosh Thai
MacTurkish	Macintosh Turkish
MacUkraine	Macintosh Ukraine

7.2 ROMized Locales

The US English locale is supported by default and is ROMized in the CVM.

TABLE 7-2 lists other locales in the file `localedata.jar`.

TABLE 7-2 Locales

Locale ID	Country	Language
ar_SA	Saudi Arabia	Arabic
zh_CN	China	Chinese (Simplified)
zh_TW	Taiwan	Chinese (Traditional)
nl_NL	Netherlands	Dutch
en_AU	Australia	English
en_CA	Canada	English
en_GB	United Kingdom	English
fr_CA	Canada	French
fr_FR	France	French
de_DE	Germany	German
iw_IL	Israel	Hebrew
hi_IN	India	Hindi
it_IT	Italy	Italian
ja_JP	Japan	Japanese
ko_KR	South Korea	Korean
pt_BR	Brazil	Portuguese
es_ES	Spain	Spanish
sv_SE	Sweden	Swedish
th_TH	Thailand	Thai (Western digits)
th_TH_TH	Thailand	Thai (Thai digits)
sq_AL	Albania	Albanian
ar_DZ	Algeria	Arabic
ar_BH	Bahrain	Arabic
ar_EG	Egypt	Arabic

TABLE 7-2 Locales *(Continued) (Continued)*

Locale ID	Country	Language
ar_IQ	Iraq	Arabic
ar_JO	Jordan	Arabic
ar_KW	Kuwait	Arabic
ar_LB	Lebanon	Arabic
ar_LY	Libya	Arabic
ar_MA	Morocco	Arabic
ar_OM	Oman	Arabic
ar_QA	Qatar	Arabic
ar_SD	Sudan	Arabic
ar_SY	Syria	Arabic
ar_TN	Tunisia	Arabic
ar_AE	United Arab Emirates	Arabic
ar_YE	Yemen	Arabic
be_BY	Belorussia	Belorussian
bg_BG	Bulgaria	Bulgarian
ca_ES	Spain	Catalan
zh_HK	Hong Kong	Chinese
hr_HR	Croatia	Croatian
cs_CZ	Czech Republic	Czech
da_DK	Denmark	Danish
nl_BE	Belgium	Dutch
en_IN	India	English
en_IE	Ireland	English
en_NZ	New Zealand	English
en_ZA	South Africa	English
et_EE	Estonia	Estonian
fi_FI	Finland	Finnish
fr_BE	Belgium	French
fr_LU	Luxembourg	French
fr_CH	Switzerland	French

TABLE 7-2 Locales *(Continued) (Continued)*

Locale ID	Country	Language
de_AT	Austria	German
de_LU	Luxembourg	German
de_CH	Switzerland	German
el_GR	Greece	Greek
hu_HU	Hungary	Hungarian
is_IS	Iceland	Icelandic
it_CH	Switzerland	Italian
lv_LV	Latvia	Latvian
lt_LT	Lithuania	Lithuanian
mk_MK	Macedonia	Macedonian
no_NO	Norway	Norwegian (Bokmål)
no_NO_NY	Norway	Norwegian (Nynorsk)
pl_PL	Poland	Polish
pt_PT	Portugal	Portuguese
ro_RO	Romania	Romanian
ru_RU	Russia	Russian
sr_YU	Yugoslavia	Serbian (Cyrillic)
sh_YU	Yugoslavia	Serbo-Croatian
sk_SK	Slovakia	Slovak
sl_SI	Slovenia	Slovenian
es_AR	Argentina	Spanish
es_BO	Bolivia	Spanish
es_CL	Chile	Spanish
es_CO	Colombia	Spanish
es_CR	Costa Rica	Spanish
es_DO	Dominican Republic	Spanish
es_EC	Ecuador	Spanish
es_SV	El Salvador	Spanish
es_GT	Guatemala	Spanish
es_HN	Honduras	Spanish

TABLE 7-2 Locales *(Continued) (Continued)*

Locale ID	Country	Language
es_MX	Mexico	Spanish
es_NI	Nicaragua	Spanish
es_PA	Panama	Spanish
es_PY	Paraguay	Spanish
es_PE	Peru	Spanish
es_PR	Puerto Rico	Spanish
es_UY	Uruguay	Spanish
es_VE	Venezuela	Spanish
tr_TR	Turkey	Turkish
uk_UA	Ukraine	Ukrainian

7.3 Using Non-ROMized Locales and Character Converters

To start the CVM with the locales and charsets enabled, use the following command:

```
path/bin/cvm -Xbootclasspath/p:/emulator-platform/lib
```

path is the path to the CVM `lib/` directory containing the files `charsets.jar` and `locales.jar`.

More details about internationalization can be found at the following link:
(http://java.sun.com/docs/books/tutorial/i18n/)

CHAPTER 8

External PBP Porting Layer Plugin

The Oracle Java ME Embedded Client has a feature to plug in a 3rd party porting interface for Personal Basis Profile (PBP) 1.1. This feature implements a framework named JGFX. The JGFX framework was designed to best leverage the commonalities between different graphics library ports for PBP and also allow 3rd party plugins implementing the native interfaces required to simply PBP to be easily integrated. The native porting layer can either be plugged in as a shared library or as a static archive.

The native porting layer plugin might implement a set of javacalls and additionally some algorithms to best leverage the underlying graphics platform capabilities. The set of javacalls must be implemented, while the algorithms may or may not be implemented depending on platform capabilities.

Note – In version 1.0 the only available target is headless, therefore applications developed with PBP libraries can only be run with the SDK and the emulator.

8.1 Javacalls

As mentioned above, a 3rd party PBP porting layer must implement a set of javacall APIs which the core PBP implementation uses to achieve functionalities required from the native platform. These javacalls implement very specific functionalities and below is a brief description of each one of them.

`jgfx_javacall_open_screen`: Method to initialize screen, registering algorithms for the primary display surface.

`jgfx_javacall_set_resolution`: Method to set screen resolution.

`jgfx_javacall_release_surface`: Method to release native window/surface.

`jgfx_javacall_reset_context`: Method to reset platform specific context pointer, context pointer could point any porting specific data structure.

`jgfx_javacall_destroy_context`: Method to destroy the context pointer.

`jgfx_javacall_set_surface_color_parms`: Method to set up AlphaComposite rules, foreground/background colors of a native surface/window.

`jgfx_javacall_sync_clip`: Method to set/update clip region of a native surface/window.

`jgfx_javacall_create_font`: Method to create/register a font with the native Graphics system.

`jgfx_javacall_destroy_font`: Method to destroy/unregister a font created/registered with a native Graphics system.

`jgfx_javacall_get_string_width`: Method to get width of an input string.

`jgfx_javacall_get_string_bounds`: Method to get dimensions of the bounding rectangle for an input string.

`jgfx_javacall_init_events`: Method to initialize input event module (key events, mouse events, etcetera).

8.2 Algorithms

The 3rd party PBP porting layer plugin can implement some algorithms which are used by the PBP core to best leverage the graphics capabilities of a platform. Implementing these algorithms is optional, but the porting layer plugin may implement them for more optimized PBP implementation, such as leveraging graphics hardware accelerations and related features.

In situations where these algorithms are not implemented, PBP uses its software modules to implement the functionality achieved by these algorithms. There is a facility to associate capabilities of each algorithms. If one algorithm is not capable of handling a certain case then the core PBP JGFX framework leverages its software module to provide the missing capability.

Brief descriptions of these algorithms follow:

`JGFXAlg_CreateSurface`: Algorithm to create off-screen surface(s). If it is not implemented PBP uses memory allocation modules at runtime to allocate memory to hold surfaces.

`JGFXAlg_ReleaseSurface`: Algorithm to release off screen surface(s). If an implementation provides an algorithm to create a surface, then it must provide an algorithm to release it as well.

`JGFXAlg_GetSurfaceMemory`: Algorithm to get the handle to a surface memory. On some platforms, surface memory is retrieved by locking the surface.

`JGFXAlg_ReleaseSurfaceMemory`: Algorithm to release the handle to a surface memory. On some platforms, the handle to surface memory is retrieved by locking the surface. This method is supposed to unlock surface memory during release operation.

`JGFXAlg_GetPixel`: Algorithm to get pixel value in raw pixel format. A platform may have a different mechanism of storing pixel data or even have APIs to access pixel information rather than direct access to the screen or window. In such cases implementation of this algorithm is necessary.

`JGFXAlg_GetPixels`: Algorithm to get an array of pixel values in raw pixel format. Some platforms may have faster ways of accessing this information rather than going over all the members of the array list. Others might iterate over all the elements and get the data.

`JGFXAlg_SetPixel`: Algorithm to set a pixel value in raw pixel format. As in the case of `JGFXAlg_GetPixel`, `JGFXAlg_SetPixel` might need to use a special API to access this information on some platforms. On others, the way pixel information is stored could be different. It is better if the porting plug-in defines this algorithm.

`JGFXAlg_SetPixels`: Algorithm to set an array of pixel values in raw pixel format. Some platforms have faster ways of achieving this other than iterating over all the elements of the array.

`JGFXAlg_GetColor`: Algorithm to get the color value of a pixel in ARGB8888 format. For many platforms functionality achieved by this algorithm could be the same as `JGFXAlg_GetPixel`. For color model other than ARGB8888, the implementation must convert the color to ARGB8888 format.

`JGFXAlg_SetColor`: Algorithm to set color value of a pixel in ARGB8888 format. For many platforms the functionality could be the same as `JGFXAlg_SetPixel`. For platforms that support a color models other than ARGB8888, the implementation needs to convert the color from ARGB8888 mode to the platform color model.

`JGFXAlg_DrawLine`: Algorithm to draw a line. It can be any form, and is not restricted to vertical or horizontal. The implementation can leverage any hardware acceleration that might be present on the platform for faster rendering.

`JGFXAlg_DrawLines`: Algorithm to draw a set of lines with specified array of coordinates. Some platforms may have a graphics library API to achieve this. Others the implementation must iterate over the coordinates and draw all the lines.

`JGFXAlg_DrawRect`: Algorithm to draw a rectangle with specified coordinates. Absence of this algorithm directs JGFX core to draw a rectangle using the put pixel mechanism. It is a good idea to implement this algorithm to leverage any hardware acceleration available on the platform for this operation.

`JGFXAlg_FillRect`: Algorithm to draw a filled rectangle with specified coordinates. In many platforms this operation is hardware accelerated, so a platform specific implementation for this algorithm is highly desirable.

`JGFXAlg_FillRectRGB`: Algorithm to draw a filled rectangle with specified coordinates and color. Like the `FillRect` operation, this operation is hardware-accelerated on many platforms. In such cases it is highly desirable to have a platform-specific implementation for this algorithm.

`JGFXAlg_DrawArc`: Algorithm to draw or fill an arc with specific coordinates and angles. If native graphics library supports this operation then it is ideal to implement this algorithm.

`JGFXAlg_RoundRect`: Algorithm to draw or fill a rounded rectangle with specified coordinates for the rectangle and horizontal and vertical diameters of the arc at the four corners. If a native graphics library supports this operation, then it is desirable to implement this algorithm to leverage any hardware acceleration facility available.

`JGFXAlg_DrawString`: Algorithm to draw a text string at a specific coordinate. Since different graphics platform follow different ways of rendering a text string, JGFX core expects an implementation to implement this algorithm.

`JGFXAlg_LoadImage`: Algorithm to load and decode images. Most graphics libraries have support for this feature and an implementation can implement this to take advantage of hardware-level decoding or any software optimizations present in the native graphics system. In absence of this algorithm, the JGFX core falls back on internal image decoders.

Implementation can also choose to implement different image decoders for GIF, JPEG and PNG. Or implement only the ones available for the native graphics platform, the others are decoded via built-in software image decoders with JVM.

`JGFXAlg_Flip`: Algorithm to flip the contents of the back-buffer to the front-buffer of the double-buffered surface (could be primary screen surface or double-buffered off-screen surfaces). Most graphics platforms have hardware acceleration support for this operation and it is highly desirable for the porting layer to implement this algorithm. Absence of this algorithm would force JGFX core to use pixel copy functions to flip contents of back-buffer to the front-buffer and this could be a slow process.

`JGFXAlg_Blit`: Algorithm to block transfer of contents from one surface to another or from one part of a same surface to the other. Usually such operations have hardware acceleration support from graphics subsystem. Since this API is used frequently, it is highly desirable to have an implementation for this algorithm with native graphics system APIs, leveraging available hardware acceleration.

`JGFXAlg_StretchBlit`: Algorithm to block transfer and stretch and shrink the contents from one surface to another. If the platform has hardware acceleration for these operations it is desirable to have an implementation of this algorithm to leverage hardware acceleration. Without an implementation the JGFX core uses software algorithms to achieve this functionality and it could be a slow process.

`JGFXAlg_GetEvent`: Algorithm to get input events on a platform from sources such as remote controller, key board, etc. Many graphics subsystems map the input events in their own suitable form and pass on these events to applications. It is desirable to implement this algorithm to suit the native graphics system.

Unlike javacalls, these algorithms must be registered at surface creation. This allows an implementation to have an algorithm registered for the primary screen surface, but not for an off-screen surface. This is useful if the graphics platform does not support the operation for off-screen surfaces.

The JGFX framework also enables the integrator to specify capabilities of each algorithm. The capabilities are as follows:

```
JGFX_ALGFLAG_HWACCEL = 0x00000001
/* whether the operation is hw accelerated */
JGFX_ALGFLAG_NEED_SURFMEM = 0x00000002
/* whether the operation required surface memory to be mapped */
JGFX_ALGFLAG_STRETCHFLIP = 0x00000004
/* whether the algorithm can flip pixels around while stretching */
JGFX_ALGFLAG_HANDLESCLIP = 0x00000008
/* whether the algorithm can handle clip settings */
JGFX_ALGFLAG_NEED_SW_PIXELS = 0x00000010
/* whether the algorithm needs foreground pixel to be calculated */
```

Depending on the capabilities attached, the JGFX core acts according to the situation when it performs the actual operation. For example, if the algorithm is not capable of handling clip regions and JGFX understands the need to render with respect to a particular clip setting, then it might handle the operation using its own software modules instead of calling the registered algorithm.

Along with these capabilities, some of the algorithms also have a facility to add the Alpha Composite rules that the algorithm supports. The possible modes are as follows:

```
JGFXCR_CLEAR      = 0x01,
JGFXCR_SRC        = 0x02,
JGFXCR_SRC_OVER   = 0x04,
JGFXCR_XOR        = 0x08,
JGFXCR_ALL        = 0x0F,
```

Ideally an algorithm could support ALL of these modes, but if the algorithm lacks support for any of these rules (due to absence of platform support or something else), then the algorithm can notify JGFX core about the missing support. JGFX core then falls back on its software algorithms to implement the functionality for the missing Alpha Composite rule. For example, if an algorithm cannot implement operations in XOR mode due to lack of platform support, then JGFX can handle these operations via the software algorithm once notified.

Note – JGFX header files are not included in the release bundle, but they can be obtained from Oracle on request. They are important only if a plugin for JGFX must be implemented, otherwise they are not of much importance.

8.3 Native Image Decoding

The image decoding module can use native graphics system APIs to leverage any platform support available for image decoding and thus improving performance on image decodes.

The Oracle Java ME Embedded Client implementation contains built-in Java image decoders. On graphics platforms where there is no support for image decoding as a whole or for a particular image format, the PBP implementation can still fall back on the built-in image decoders. The implementation falls back on built-in image decoders even in cases where a particular image cannot be decoded properly by the native graphics system.

The important runtime properties to control the native image decoding feature are as follows:

- `java.awt.NativeImageDecoding`

  The default value of this property is true.

  Setting this property to false disables the native image decoding module completely and forces usage of built-in Java decoders.

- `java.awt.NativeImageDecoding.PNG`

  The default value of this property is true.

  Setting this property to false disables the native image decoding of PNG images and these images are decoded using built-in Java decoders.

- `java.awt.NativeImageDecoding.JPEG`

  The default value of this property is true.

  Setting this property to false disables the native image decoding of JPEG images and these images are decoded using built-in Java decoders.

- `java.awt.NativeImageDecoding.GIF`

  The default value of this property is false.

  Setting this property to true enables native image decoding of GIF images.

  Note: This property is set to false by default, because Java image decoders for GIF images had better performance in JIT mode.

8.4 Image Caching

When many images are decoded and loaded to memory there is eventually a need to dispose of some of the images. It is possible to cache the decoded images on the hard disk automatically. When the image is loaded again, it is loaded directly from the cached file. This saves CPU cycles as the image is decoded only once, even if its memory is disposed.

8.5 Java VM Shutdown With PBP Application

This enhancement allows JVM to shutdown properly when a PBP application chooses to gracefully exit by meeting the following conditions:

- Java Main thread exits
- No User threads are alive
- No components are displayable
- No native events are pending to be processed

JVM cleans up all the resources allocated by itself and the application while exiting.

8.6 External BLIT Interface

The purpose of this interface is to allow the BD-J implementation to BLIT Java images from native functions. It is used to implement the Frame Accurate Animation (FAA) API.

8.7 Dynamic Set Resolution

Java API `java.awt.Frame.setSize()` supports screen or window resolution modification. The specified width and height parameter values are used to determine the new resolution.

8.8 Configurable Background Color

To comply with the BD-J specification, the Oracle Java ME Embedded Client stack has been enhanced to set the background color of the main Window to Clear.

This enhancement can affect certain interactive TCK tests, which assumes a Java SE Solid Non-Black background.

Given the PBP stack, a Java SE AWT implementation can be used to set the default background color to Light Gray. At runtime the parameter can reset the background color (in ARGB format) at JVM startup. The new background color affects the main AWT Window plus any default graphics instances that do not inherit a background color.

USAGE:

To modify the default background color, set the runtime flag:

```
java.awt.DefaultBackgroundColor=[0x]<32bit ARGB hex value>
-Djava.awt.DefaultBackgroundColor=0xFFFF0000 (for solid Red)
-Djava.awt.DefaultBackgroundColor=0x8000FF00 (for 50% Green)
-Djava.awt.DefaultBackgroundColor=0x00000000 (for Clear)
```

8.9 IXC Generic Implementation (`com.sun.xlet` package)

The IXC generic implementation supports `javax.tv.xlet.XletContext` as well as `javax.microedition.xlet.XletContext` and can be used for `javax.tv.xlet` based Xlets.

The `IxcRegistry` implementation is generic accordingly and is integrated with `com.sun.xlet.ixc` package. Oracle provides an implementation for the `org.dvb.io.ixc.IxcRegistry` class that supports the DVB persistent storage requirements. Since the Class lacks support for the `unbindAll()` API, `com.sun.dvb.io.ixc.IxcRegistryExtension` provides this functionality.

PART III Working Without An IDE

This section discusses porting tools and techniques.

Legacy Tools

Legacy information on building and running applications.

APPENDIX **A**

Legacy Tools

This appendix contains information about building and running applications with the Oracle Java ME Embedded Client. The techniques described here have been superseded by the information in Chapter 3.

A.1 Tools

This section describes basic methods for building applications for Oracle Java ME Embedded Client. Chapter 3 describes simpler methods based on NetBeans and an emulator. Even so, reading this section helps you understand what is happening behind the scenes.

The fundamental tools you need are in the Java 2 Platform, Standard Edition Software Development Kit (J2SE SDK) version 1.4.2. At this stage JDK 1.4.2 is end-of-life'd and version 1.4.2_19 is archived here:

`http://java.sun.com/products/archive/j2se/1.4.2_19/index.html`

Note – Because the CDC VM requires JDK 1.4.2 classes at build time this section refers to a 1.4.2 installation. You can use a more recent version of the JDK if you specify the option `-target 1.4`.

You also need some kind of scripting tool to automate your builds. You can use shell scripts or `make`, but this chapter describes the process using Ant. Ant reads XML build scripts and is useful for automating Java platform development tasks. Ant is available here:

`http://ant.apache.org/`

After you install the J2SE SDK 1.4.2 and Ant, you can test your installation like this:

```
$ java -version
java version "1.4.2_19"
Java(TM) 2 Runtime Environment, Standard Edition
    (build 1.4.2_19-b04)
Java HotSpot(TM) Client VM (build 1.4.2_19-b04, mixed mode)
$ ant -version
Apache Ant version 1.7.1 compiled on October 1 2008
```

Building an application for Oracle Java ME Embedded Client is a simple matter of using `javac` to compile source files. Because Oracle Java ME Embedded Client is based on a different software stack than the J2SE platform, use the `-bootclasspath` option to tell the compiler where to find the classes for Oracle Java ME Embedded Client.

When the classes are compiled, use `jar` to create an archive of them. Transfer this archive to your Oracle Java ME Embedded Client device. Then run your application with the Oracle Java ME Embedded Client virtual machine (`cvm`).

The next section shows how to create and run a simple application. The subsequent section describes how to automate the development cycle using Ant.

A.2 Compiling the Hard Way

Start with the `Main.java` source files described in Section 3.5 "A First Xlet" on page 3-4. Create a directory named src and save `Main.java` there.

This section describes how to compile, package, and run your application directly from the command line. You should read it just so you understand how it all works. The next section includes an Ant build script that makes everything much cleaner.

Note – This section assumes you have installed JDK 1.4.2_19 as described in Section A.1 "Tools" on page A-1. Download it from `http://java.sun.com/products/archive/j2se/1.4.2_19/index.html`

To keep things neat as you're building, place the compiled bytecode (`.class` files) in a `HelloXlet/build/classes` directory.

At the command line, you can compile the source files like this (long lines are split for clarity):

```
$ export OJEC=/usr/local/Oracle_JavaME_Embedded_Client/1.0/emulator-platform
$ mkdir build
$ mkdir build/classes
$ javac -sourcepath src -bootclasspath $OJEC/lib/btclasses.zip
  -classpath $OJEC/lib/basis.jar src/*.java -d build/classes
```

It's tidier as part of an Ant script, as shown in the next section.

The next step in building your application is bundling the class files. In a more complex application, resource files like images and sounds are also bundled with your class files.

At the command line, use the `jar` tool to create your application package, `HelloXlet/dist/HelloXlet.jar`:

```
$ mkdir dist
$ jar cvf dist/HelloXlet.jar -C build/classes/ .
added manifest
adding: helloxlet/(in = 0) (out= 0)(stored 0%)
adding: helloxlet/Main.class(in = 2498) (out= 1355)(deflated 45%)
```

Finally, to run this application, invoke `cvm`. You have to supply a security policy, which details which operations the application is allowed to perform. Save the following wide-open security policy as `HelloXlet/unsafe.policy`:

```
grant {
  permission java.security.AllPermission;
};
```

Now you are ready to run your application as follows (with newlines for readability):

```
$ $OJEC/bin/cvm
  -Djava.security.policy=unsafe.policy com.sun.xlet.XletRunner
  -name helloxlet.Main
  -path dist/HelloXlet.jar
@@XletRunner starting Xlet helloxlet.Main
```

You'll see a window the with words "Hello Java World".

A.3 Automating With Ant

You can automate these complicated command lines using any kind of scripting tool. This section describes a build script that automates the process using Ant.

The Ant script contains the three main targets that are described on the command line in the previous section. The `compile` target takes care of compiling the source code. Packaging occurs in `jar`. Finally, the `run` target launches your application using `cvm`.

A fourth target, `clean`, removes compiled classes and the packaged application.

To use the following script, just edit the value of the `ojec` property near the top so it points to your installation of Oracle Java ME Embedded Client.

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="helloxlet.Main" default="run" basedir=".">
  <!-- Modify this property to point to your installation. -->
  <property name="ojec"
     value="/usr/local/Oracle_JavaME_Embedded_Client/1.0/emulator-platform/"/>
   <property name="jarname" value="HelloXlet"/>
   <target name="run" depends="jar">
      <exec executable="${ojec}/bin/cvm">
          <arg line="-Djava.security.policy=unsafe.policy"/>
          <arg line="com.sun.xlet.XletRunner"/>
          <arg line="-name ${ant.project.name}"/>
          <arg line="-path dist/${jarname}.jar"/>
      </exec>
   </target>
   <target name="jar" depends="compile">
      <mkdir dir="dist"/>
      <jar basedir="build/classes"
      jarfile="dist/${jarname}.jar"/>
   </target>
   <target name="compile">
      <mkdir dir="build/classes"/>
      <javac destdir="build/classes" srcdir="src"
      bootclasspath="${ojec}/btclasses.zip"
      classpath="${ojec}/lib/basis.jar"/>
   </target>
   <target name="clean">
      <delete dir="build"/>
      <delete dir="dist"/>
   </target>
</project>
```

To compile, package, and run in one easy step, just type `ant run` at the command line.

Index

L

M

N

P

R

S

T

U

V

X