Basic Customization Guide

Sun Java™ Wireless Toolkit for CLDC

Version 2.5.1

Sun Microsystems, Inc.
www.sun.com

April 2007

Contents

Preface

The *Sun Java™ Wireless Toolkit for CLDC Basic Customization Guide* provides technical details for customizing the Sun Java Wireless Toolkit for CLDC.

Who Should Use This Book

This guide is intended for developers who need to customize the Sun Java Wireless Toolkit for CLDC for their own use, such as to be able to accommodate a new device or modifying the source code to accommodate new APIs for emulators.

How This Book Is Organized

This guide contains the following chapters and appendices:

Chapter 1 gives a brief overview of the customization possibilities.

Chapter 2 describes how to create new emulator skins.

Chapter 3 describes how to create obfuscator plug-ins.

Using Operating System Commands

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

For Windows and Linux, this document represents the shell prompt as *directory.*

The examples in this document are for the Windows platform. On Linux systems, replace backslash path separators (\) with forward slashes (/). For example:

Windows	*toolkit*`\wtklib\devices\DefaultColorPhone\DefaultColorPhoneIcon.png`
Linux	*toolkit*`/wtklib/devices/DefaultColorPhone/DefaultColorPhoneIcon.png`

Typographic Conventions

TABLE P-1 Typographic Conventions

Typeface	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `% You have mail.`
`AaBbCc123`	What you type, when contrasted with on-screen computer output	`% su` `Password:`
AaBbCc123	Book titles, new words or terms, words to be emphasized	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be superuser to do this.
	Command-line variable; replace with a real name or value	To delete a file, type `rm` *filename*.

Related Documentation

This section lists related Java Platform, Micro Edition (Java ME) specifications. Java ME was formerly referred to as the Java 2 Platform, Micro Edition, or J2ME™, as you see in some of the specification names.

TABLE P-2 Related Documentation

Topic	Title
Sun Java Wireless Toolkit for CLDC	*Sun Java Wireless Toolkit for CLDC User's Guide*
Customization	*Sun Java Wireless Toolkit for CLDC Advanced Customization Guide**
Unified Emulator Interface	*Unified Emulator Interface Specification*
PDAP Optional Packages - JSR 75	*PDA Optional Packages for the J2ME Platform*
Bluetooth and OBEX - JSR 82	*Java APIs for Bluetooth*
MIDP 2.1 - JSR 118	*Mobile Information Device Profile 2.0*
CLDC 1.1 - JSR 139	*J2ME Connected Limited Device Configuration*
MMAPI - JSR 135	*Mobile Media API*
J2ME Web Services - JSR 172	*J2ME Web Services Specification*
SATSA - JSR 177	*Security and Trust Services APIs for J2ME*
Location API - JSR 179	*Location API for J2ME*
SIP API - JSR 180	*SIP API for J2ME*
Mobile 3D Graphics - JSR 184	*Mobile 3D Graphics API for J2ME*
JTWI - JSR 185	*Java Technology for the Wireless Industry*
WMA 2.0 - JSR 205	*Wireless Messaging API (WMA)*
CHAPI 1.0 - JSR 211	*Content Handler API*
SVG API - JSR 226	*Scalable 2D Vector Graphics API for J2ME*
Payment API - JSR 229	*Payment API*
Advanced Multimedia - JSR 234	*Advanced Multimedia Supplements*
Mobile Internationalization - JSR 238	*Mobile Internationalization API*
Java Binding for OpenGL® ES API - JSR 239	*Java Binding for OpenGL® ES API*
MSA - JSR 248	*Mobile Service Architecture*

* Available with a source license.

Although specifications are definitive, they are not always the most accessible kind of information. Check out the Mobility page for an additional viewpoint:

http://developers.sun.com/techtopics/mobility/

Accessing Documentation Online

The following sites provide technical documentation related to Java technology:

- http://developers.sun.com/
- http://java.sun.com/docs/

We Welcome Your Comments

We are interested in improving our documentation and welcome your suggestions. Email your feedback from developers.sun.com.

CHAPTER 1

Introduction

The Sun Java Wireless Toolkit for CLDC provides an emulation environment for the development of MIDP applications. This document provides instructions for customizing the toolkit in two useful ways:

- Creating new emulator skins
- Creating obfuscator plug-ins

The remainder of this chapter briefly describes each of these customizations.

1.1 Creating New Emulator Skins

You can customize the emulators in the Sun Java Wireless Toolkit for CLDC in the following ways:

1. Download third-party emulators and install them into the Sun Java Wireless Toolkit for CLDC.
2. Create a new emulator skin based on the Sun Java Wireless Toolkit for CLDC's default emulator. This process is described in Chapter 2.
3. Customize the default emulator implementation. To do this, license the Sun Java Wireless Toolkit for CLDC source code to customize the emulator implementation.

1.2 Creating Obfuscator Plug-Ins

An *obfuscator* is a tool that is used to reduce the size of an executable MIDlet suite. Smaller MIDlet suites mean lower download times, which, in the current bandwidth-starved wireless world, means less waiting and possibly lower airtime charges for users.

The Sun Java Wireless Toolkit for CLDC includes support for the ProGuard obfuscator (`http://proguard.sourceforge.net/`), but it includes a flexible architecture that allows for any type of obfuscator.

Chapter 3 provides the technical details.

CHAPTER 2

Skinning the Emulator

This chapter describes how emulator skins are defined. You can modify existing skins or create new skins for the emulator. This process is known as *skinning* the emulator.

2.1 Skin Property File

Emulator skins are defined by a single property file. Each skin property file is contained in its own subdirectory of *toolkit*`\wtklib\devices`, where *toolkit* is the installation directory of the Sun Java Wireless Toolkit for CLDC. The name of the property file matches the directory name.

For example, the `DefaultColorPhone` skin is defined by `DefaultColorPhone.properties` in the *toolkit*`\wtklib\devices\DefaultColorPhone` directory.

The skin property file defines the appearance and behavior of the emulator skin. It includes pointers to images and sounds that may or may not reside in the same directory. For example, the `DefaultColorPhone` directory contains images for the phone itself, but the icons and sounds for `DefaultColorPhone` are defined in `wtklib\devices\Share`.

The remainder of this chapter describes the contents of the skin property file. The property file is a plain text file. You can use any text editor to modify it. In general, entries in the property file have a property name followed by a value. A colon or equals sign separates the name and value. Lines that begin with a hash mark (#) are comments.

The simplest way to create a new skin is to copy an existing one and modify it, for example:

1. **Copy the `DefaultColorPhone` directory.**
2. **Name the new directory with the name of your new skin.**
3. **Rename the properties file to match the directory name.**

   If you named the directory `NewSkin`, rename its property file to `NewSkin.properties`.

2.2 Skin Appearance

The overall appearance of the emulator skin is determined by a variety of factors, each of which is described in this section:

- Skin images
- Screen bounds and paintable area
- Screen characteristics
- Icons
- Fonts
- Commands
- Sounds

2.2.1 Skin Images

Much of a skin's appearance is determined by three images:

1. The *default* image shows the device in a neutral (normal) state.
2. The *highlighted* image shows the device with all the buttons highlighted, as they are when the user moves the mouse over the buttons.
3. The *pressed* image shows the device with all its buttons pressed.

Each of these images shows the entire device. The toolkit uses portions of these images to show button highlights and button presses.

For example, the three images from `DefaultColorPhone` are shown in FIGURE 2-1.

FIGURE 2-1 Normal, Highlighted, and Pressed Images for `DefaultColorPhone`

A close-up of a portion of each keypad is shown in FIGURE 2-2 so you can see the differences in the three images.

FIGURE 2-2 Normal, Highlighted, and Pressed Emulator Skin Image Details

In the skin property file, the three image files are specified with the following properties:

- `default_image=`*image-file-name*
- `pressed_buttons_image=`*image-file-name*
- `highlighted_image=`*image-file-name*

The image files can be PNG, GIF, or JPEG. Use the same dimension for all image files. For example, `DefaultColorPhone.properties` has the following entries:

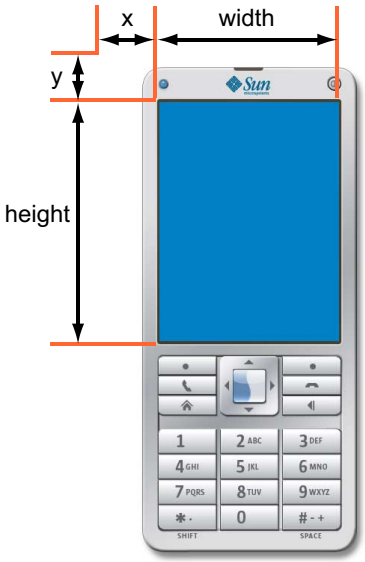
- `default_image=DefaultColorPhoneNormal.png`
- `pressed_buttons_image=DefaultColorPhonePressed.png`
- `highlighted_image=DefaultColorPhoneHiLite.png`

2.2.2 Screen Bounds and Paintable Area

The screen represents the display of a real device. It is defined by the overall screen bounds, the *paintable* bounds, and other parameters that determine factors like the number of colors.

The overall screen bounds are the total area of the display. They are defined in pixel measurements relative to the origin of the image files, which is in the upper left corner (0,0).

FIGURE 2-3 Screen Bounds



The screen bounds are specified in the property file as follows:

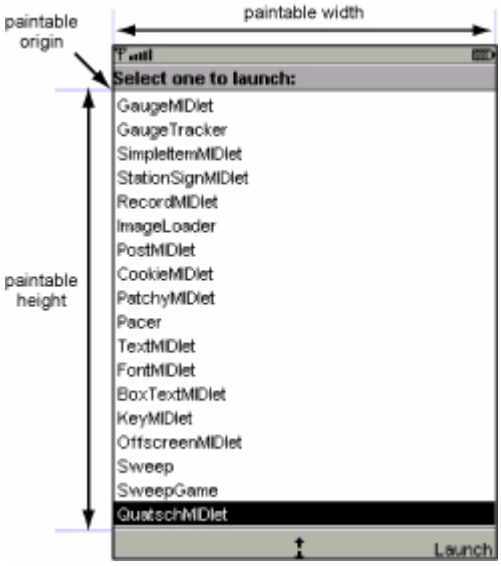
- `screen.x=`*x-coordinate*
- `screen.y=`*y-coordinate*
- `screen.width=`*width*
- `screen.height=`*height*

For example:

- `screen.x=37`
- `screen.y=54`
- `screen.width=240`
- `screen.height=320`

Most devices do not make their full display area available to MIDP applications. The remainder of the screen is generally reserved for icons and indicators of various kinds. Similarly, the Sun Java Wireless Toolkit for CLDC emulator enables you to define a subset of the full screen, called the *paintable* area, that is available for MIDP applications. The origin of the paintable area is expressed in coordinates relative to the upper left corner of the display. For example, the `DefaultColorPhone` emulator skin uses a top bar for icons and a bottom bar for soft labels and other icons, as shown in FIGURE 2-4.

FIGURE 2-4 Paintable Screen Area in `DefaultColorPhone`



In the emulator skin property file, the paintable area is expressed as follows.

- `screenPaintableRegion.x=`*x-coordinate*
- `screenPaintableRegion.y=`*y-coordinate*
- `screenPaintableRegion.width=`*width*
- `screenPaintableRegion.height=`*height*

Note, the x and y coordinates are relative to the screen position. Also the width and height values for `screenPaintableRegion` cannot exceed the corresponding values for `screen.width` and `screen.height`.

For example:

- `screenPaintableRegion.x=0`
- `screenPaintableRegion.y=10`
- `screenPaintableRegion.width=240`
- `screenPaintableRegion.height=290`

Note – For full-screen mode (in MIDP 2.0 and higher), the emulator uses the area beginning at the paintable area origin and extending through the bottom right corner of the screen. In `DefaultColorPhone`, this is the entire screen region with the exception of the top bar.

2.2.3 Screen Characteristics

The emulator skin property file determines the number of colors supported by the screen and the aspect ratio of the pixels. `iscolor` specifies whether the emulator skin uses color or grayscale.

2.2.3.1 `isColor=`*[true|false]*

Specify *true* for color or *false* for grayscale.

2.2.3.2 `colorCount=`*number*

`colorCount`, specifies the number of available colors. For grayscale devices it specifies the number of gray levels:

For example, `DefaultColorPhone` has a color screen with 4096 colors:

```
isColor=true
colorCount=0x1000
```

2.2.3.3 `enableAlphaChannel=`*[true|false]*

This option determines the emulator's handling of alpha (transparency).

2.2.3.4 gamma=*number*

This value determines gamma correction. A value of 1 means no error correction.

2.2.3.5 screenDoubleBuffer=[*true*|*false*]

`true` enables double buffering, `false` disables it.

2.2.3.6 screenBorderColor=*color*

`screenBorderColor` specifies the background color that is used for the non-paintable areas of the screen. For example, `DefaultColorPhone` uses the following color:

```
screenBorderColor=0xb6b6aa
```

2.2.3.7 screenBGColor=*color*

Use this property to set the background color of the screen on grayscale devices.

2.2.4 Icons

The Sun Java Wireless Toolkit for CLDC emulator supports the use of icons, which are small images that convey information to the user. Usually, icons are placed on the display but outside the paintable area. The emulator implements a fixed set of icons which are described in TABLE 2-1.

TABLE 2-1 Emulator Icons

Name	Description
battery	Shows battery state
domain	Indicates the protection domain of the running MIDlet
down	Indicates that scrolling is possible
inmode	Indicates the input mode: lower case, upper case, numbers
internet	Shows Internet activity
left	Indicates that scrolling is possible

TABLE 2-1 Emulator Icons

Name	Description
reception	Shows wireless signal strength
right	Indicates that scrolling is possible
up	Indicates that scrolling is possible

Icons are defined with a location (measured relative to the origin of the screen), a default state, and a list of images that correspond to the possible states. For example, here is the definition of the `down` icon in `DefaultColorPhone`, which is a downward-pointing arrow that appears when a list or form is shown that is taller than the available screen space:

```
icon.down: 113, 314, off
icon.down.off:
icon.down.on: ../Share/down.gif
```

The first line specifies the location where the icon is shown, which for `DefaultColorPhone` is a location in the center of the bottom bar, outside the paintable screen area. The default state is `off`.

No image file corresponds to the `off` state, but the `on` state uses the image `down.gif` from the `wtklib\devices\Share` directory.

Another interesting example is the `inmode` icon, which includes seven states with six corresponding image files:

```
icon.inmode: 113, 2, off
icon.inmode.off:
icon.inmode.ABC: ../Share/ABC.gif
icon.inmode.abc: ../Share/abc_lower.gif
icon.inmode.123: ../Share/123.gif
icon.inmode.kana: ../Share/kana.gif
icon.inmode.hira: ../Share/hira.gif
icon.inmode.sym: ../Share/sym.gif
```

Another aspect of the emulator that is similar to an icon is the network indicator. Instead of being located in the screen, the network indicator is shown on the emulator skin. In `DefaultColorPhone`, the network indicator is shown as a small green light in the upper left of the emulator skin. The network indicator is defined using two properties:

- `netindicator.image:` *image*
- `netindicator.bounds:` *x*, *y*, *width*, *height*

For example, in `DefaultColorPhone`, the network indicator looks like this:

- `netindicator.image: net_indicator.png`
- `netindicator.bounds: 53, 27, 30, 30`

The width and height should match the width and height of the network indicator image.

2.2.5 Fonts

The fonts used by the emulator are defined in the skin property file. In essence, you can define a font for each of the faces, styles, and sizes that are available in MIDP's `Font` class. The format is as follows:

`font.`*face*`.`*style*`.`*size*`:` *font-specifier*

You can surmise the fact, style, and size parameters from the MIDP `Font` API, except the identifiers are lower case in the emulator skin property file. The font face is `system`, `monospace`, or `proportional`, the style is `plain`, `bold`, or `italic`, and the size is `small`, `medium`, or `large`.

The font specifier follows the convention laid out in the Java Platform, Standard Edition (Java SE) `java.awt.Font` class library. The following example from `DefaultColorPhone` defines the proportional italic fonts in all three sizes:

```
font.proportional.italic.small: SansSerif-italic-9
font.proportional.italic.medium: SansSerif-italic-11
font.proportional.italic.large: SansSerif-italic-14
```

You need to specify a default font that is used in case no other definition is available. In `DefaultColorPhone`, a 10-point SansSerif font is used for the default:

```
font.default=SansSerif-plain-10
```

Fonts can also be underlined. By default, this is supported by the MIDP implementation, but you can disable for specific fonts like this:

`font.`*face*`.`*style*`.`*size*`.underline.enabled=false`

If you wish, you can disable underlining for all fonts like this:

```
font.all.underline.enabled=false
```

Instead of using system fonts, you have an additional option of using a *bitmap* font. A bitmap font is simply an image that contains character shapes for a font. The bitmap font image is a single line of text containing one of each character shape. To define a bitmap font, use the following property:

`font.`*name=font-property-file*

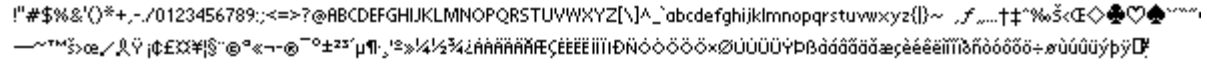
The font property file contains the following property definitions:

- `font_image =` *image-file*
- `font_height =` *font-height*

- `font_ascent = font-ascent`
- `font_descent = font-descent`
- `font_leading = font-leading`

The image file can be in PNG, GIF or JPEG format. It must contain a *single* row of characters. FIGURE 2-5 shows two rows so that the characters are easier to see.

FIGURE 2-5 Bitmap Font Image



The height, ascent, descent, and leading are all specified in pixels. If you are unfamiliar with these font terms, refer to the Java SE documentation for `java.awt.FontMetrics`.

The font property file must also contain a list of mappings between ASCII character codes and horizontal pixel offsets into the image. In the following example, the ASCII code 65 is mapped to the horizontal offset 124:

```
ascii_x-65=124
```

Once a bitmap font is defined, its name can be used as a font specifier.

2.2.6 Soft Button Labels

Soft buttons are buttons without a fixed function. They are fully discussed later in this chapter. Labels for the soft buttons are shown on the screen. The emulator skin property file determines where and how the soft button labels are shown.

The fonts for the soft button labels are defined using font aliases, which are short names that you assign to a font. Each soft button label is described by a property:

`softbutton.`*n*`=`*x*`,` *y*`,` *width*`,` *height*`,` *font-alias*`,` *alignment*

Valid values for *alignment* are `left`, `right`, and `center`.

For example, the following properties tell the toolkit to use a Courier 12-point font for the soft button labels:

- `font.softButton=Courier-plain-12`
- `softbutton.0=1,306,78,16, softButton, left`
- `softbutton.1=160,306,78,16, softButton, right`

First the font alias `softButton` is defined. The first label is left justified, while the second is right justified.

2.2.7 Sounds

MIDP alerts have associated sounds. In the Sun Java Wireless Toolkit for CLDC emulator, sounds are defined using files, one for each type enumerated in the MIDP `AlertType` class. The emulator can use any sound file type that is supported by the underlying Java SE implementation. In Java SE Development Kit 1.5, this includes AIFF, AU, WAV, MIDI, and RMF. For example, here are the definitions in `DefaultColorPhone`:

```
alert.alarm.sound:    ../Share/mid_alarm.wav
alert.info.sound:     ../Share/mid_info.wav
alert.warning.sound: ../Share/mid_warn.wav
alert.error.sound:    ../Share/mid_err.wav
alert.confirmation.sound: ../Share/mid_confirm.wav
```

A default sound is played if no sound is defined for a specific alert type:

`alert.confirmation.sound:` *sound-file*

In addition, you can define a sound that is played to simulate a phone's vibration. In `DefaultColorPhone`, it looks like this:

```
vibrator.sound: ../Share/vibrate.wav
```

2.3 Mapping User Input

Describing an emulator skin is done in two parts. The first part is the appearance, which is described above. The second part defines how user input is mapped in the emulator.

2.3.1 Keyboard Handler

A keyboard handler takes button presses and performs an appropriate action in the emulator. For example, if you use the mouse to press one of the soft buttons, it is the keyboard handler that makes the appropriate action happen in the emulator.

The keyboard handler defines a set of standard button names, which you will use when you define buttons. You just have to tell the emulator where the buttons are located in the skin and the keyboard handler takes care of the rest.

The Sun Java Wireless Toolkit for CLDC emulator includes two keyboard handlers, one for phone devices with an ITU-T keypad (`DefaultKeyboardHandler`) and one for devices with a full Qwerty keyboard. For example, `DefaultColorPhone` includes this keyboard handler property:

```
keyboard.handler = com.sun.kvem.midp.DefaultKeyboardHandler
```

`DefaultKeyboardHandler` recognizes the following standard button names: `0` through `9`, `POUND`, `ASTERISK`, `POWER`, `SEND`, `END`, `LEFT`, `RIGHT`, `UP`, `DOWN`, `SELECT`, `SOFT1`, `SOFT2`, `SOFT3`, `SOFT4`, `USER1` through `USER10`.

In `QwertyDevice`, the keyboard handler looks like this:

```
keyboard.handler = com.sun.kvem.midp.QwertyKeyboardHandler
```

`QwertyKeyboardHandler` supports the same buttons as `DefaultKeyboardHandler` and also includes buttons found on a standard keyboard like alphabetic keys, shift, and alt.

2.3.2 Buttons

Buttons are defined using a name and a set of coordinates. If two sets of coordinates are supplied, a rectangular button is defined. If more than two sets of coordinates are present, a polygonal area is used for the button.

The button region is defined relative to the device skin image. When the user moves the mouse over a defined button region, the corresponding region from the skin highlight image is shown. If the user presses a button, the corresponding region from the skin pressed image is shown.

By themselves, buttons aren't very interesting. They just associate a button name with a rectangular or polygonal region. It's the keyboard handler's job to map the button name to a function in the emulator. Section 2.3.3, "Assigning Desktop Keyboard Keys to Buttons" on page 2-13 explains how keys on your desktop computer's keyboard can be mapped to buttons.

The following property shows how to define a rectangular region for the 5 button. Its origin is `140, 553`, with a width of `84` and a height of `37`.

```
button.5 = 140, 553,  84,  37
```

Here is an example polygonal definition for the asterisk button:

```
button.ASTERISK = 66, 605, 110, 606, 140, 636, 120, 647, 70, 637
```

This polygon is defined using straight line segments connecting the listed points:

```
66, 605
110, 606
```

```
140, 636
120, 647
70, 637
```

2.3.3 Assigning Desktop Keyboard Keys to Buttons

Buttons can have one or more associated desktop keyboard keys. This means that you can use your desktop keyboard to control the emulator instead of having to move the mouse over on the device skin and press the mouse button.

For example, `DefaultColorPhone` enables you to press F1 on your desktop keyboard to simulate the left soft button. The left soft button is defined as `SOFT1` in the property file, as follows:

```
button.SOFT1 = 78, 417, 120, 423, 126, 465, 74, 440
```

The desktop keyboard shortcut is defined like this:

```
key.SOFT1 = VK_F1
```

The actual key definitions are *virtual key codes*, which are defined in the Java SE `java.awt.event.KeyEvent` class library. See the Java SE documentation for details.

You can assign multiple desktop keyboard keys to a button, if you wish. In the following example from `DefaultColorPhone`, the 5 key or the number pad 5 key on your desktop keyboard are both defined as shortcuts for the 5 button on the emulator skin:

```
key.5 = VK_5 VK_NUMPAD5
```

2.3.4 Mapping Game Keys

Game actions are already defined in `DefaultKeyboardHandler`, but you can specify your own game actions with `QwertyKeyboardHandler`. Use lines of this form:

`game.`*function* `=` *button-name*

The function can be one of `LEFT`, `RIGHT`, `UP`, `DOWN`, and `SELECT`. Standard button names are described earlier in this chapter.

The default settings are as follows:

- `game.UP = UP`
- `game.DOWN = DOWN`

- `game.LEFT = LEFT`
- `game.RIGHT = RIGHT`
- `game.SELECT = SELECT`

2.3.5 Mapping Keys to Characters

With `QwertyKeyboardHandler`, you can specify which character is generated by a button press either alone or in combination with the Shift or Alt keys.

Use a line of this form:

`keyboard.handler.qwerty.`*button* `= '`*base-character*`' '`*shift-character*`' '`*alternate-character*`'`

The base character is the character the button normally generates, shift character is the character used when the button is pressed at the same time as shift, and alternate character is the character generated when the button is pressed at the same time as alt.

You emulate a button press at the same time as pressing Shift or Alt in two ways:

- Map the buttons to the keyboard, as in the previous section, and press the key associated with the button at the same time as the Shift or Alt keys.
- Press the button shift-lock or alt-lock and then do the button press. Press Shift-lock or Alt-lock again to revert to the initial state.

For example:

```
keyboard.handler.qwerty.A = 'a' 'A' '?'
```

2.3.6 Mapping Commands to Soft Buttons

Commands are part of the MIDP specification. They are a flexible way to specify actions that need to be available to the user without mandating how a particular device makes them available.

In general, MIDP devices use soft buttons to invoke commands. The command text is shown on the display somewhere physically near to the soft buttons. If more commands are available than available soft buttons, the implementation shows one soft button label as a menu. Pressing the menu soft button brings up a menu of available commands.

The Sun Java Wireless Toolkit for CLDC emulator enables you to specify where you want certain types of commands to appear, based on the command types specified in `javax.microedition.lcdui.Command`. For example, on an emulator skin with two soft buttons, you might prefer that `BACK` and `EXIT` commands always appear on the left soft button, and that the `OK` commands appear on the right soft button.

You can specify these types of preferences in the emulator skin property file, using lines like the following:

`command.keys.`*`command-type=button`*

For example, `DefaultColorPhone` defines command preferences this way:

```
command.keys.BACK = SOFT1
command.keys.EXIT = SOFT1
command.keys.CANCEL = SOFT1
command.keys.STOP = SOFT1

command.keys.OK = SOFT2
command.keys.SCREEN = SOFT2
command.keys.ITEM = SOFT2
command.keys.HELP = SOFT2
```

By specifying additional button names, you can specify other preferred buttons for a particular command type. For example, this line tells the emulator to map `BACK` commands to `END`, if it is available, or `SOFT1` otherwise.

```
command.keys.BACK = END SOFT1
```

Finally, if you wish, you can specify that a soft button is only used for specific command types. The following definition restricts the `SOFT1` key to only the command types `BACK`, `EXIT`, `CANCEL`, and `STOP`.

```
command.exclusive.SOFT1 = BACK EXIT CANCEL STOP
```

2.3.7 Command Menu

When you have fewer available soft buttons than commands, commands are placed in a menu. The Sun Java Wireless Toolkit for CLDC emulator offers control over the command menu. You can choose the button that is used to show the menu, the buttons that are used to traverse the items in the menu, and the text labels that are shown for the menu.

The following property, from `DefaultColorPhone`, tells the emulator skin to use the second soft button to show or hide the menu.

```
command.menu.activate = SOFT2
```

By default, the UP and DOWN buttons are used to traverse the menu, while SELECT is used to choose a command. You can change these assignments using the following properties:

- command.menu.select = *button*
- command.menu.up = *button*
- command.menu.down = *button*

2.3.8 Pausing and Resuming

The MIDP specification allows applications (MIDlets) to be paused at any time, for example, in response to other phone events like incoming calls.

You can use the emulator skin property file to define desktop keyboard shortcuts for pausing and resuming MIDlets. DefaultColorPhone, for example, uses F6 for pausing (suspending) and F7 for resuming:

```
midlet.SUSPEND_ALL = VK_F6
midlet.RESUME_ALL = VK_F7
```

2.3.9 Pointer Events

A single property, as follows, determines whether the emulator skin has a touch screen:

```
touch_screen=[true|false]
```

If the emulator skin does have a touch screen, pointer events are delivered to Canvases.

2.4 Locale and Character Encoding

A locale is a geographic or political region or community that shares the same language, customs, or cultural convention. In software, a locale is a collection of files, data, and code, which contains the information necessary to adapt software to a specific geographical location.

Some operations are locale-sensitive and require a specified locale to tailor information for users, such as the following:

- Messages displayed to the user

- Cultural information, such as dates and currency formats

In the Sun Java Wireless Toolkit for CLDC emulator, the default locale is determined by the platform's locale.

To define a specific locale, use the following definition:

`microedition.locale:` *locale-name*

A locale name is comprised of two parts separated by a dash (-), for example, `en-US` is the locale designation for English-United States and `en-AU` is the designation for English-Australia.

The first part is a valid ISO Language Code. These codes are the lower-case two-letter codes defined by ISO-639. You can find a full list of these codes at a number of sites, such as
http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt.

The second part is a valid ISO Country Code. These codes are the upper-case two-letter codes defined by ISO-3166. You can find a full list of these codes at a number of sites, such as:
http://www.chemie.fu-berlin.de/diverse/doc/ISO_3166.html.

The input and output APIs in CLDC use named character encodings to convert between 8-bit characters and 16-bit Unicode characters. Specific MIDP implementation might make only a small set of encodings available for MIDlets to use.

In the emulator, the default encoding is default encoder of the platform you are running on. Your emulator might use other encodings, such as UTF-8 and UTF-16, providing they are available in the Java SE platform.

To define the character encoding used by an emulator skin, use the following definition:

`microedition.encoding:` *encoding*

To define the set of all available encodings, use the following definition:

`microedition.encoding.supported:` *list of encodings*

For example:

```
microedition.encoding: UTF-8
microedition.encoding.supported: UTF-8, UTF-16, ISO-8859-1,
ISO-8859-2, Shift_JIS
```

To support all encodings supported by the Java SE platform, leave the `microedition.encoding.supported` definition blank, as in:

```
microedition.encoding.supported:
```

Note – The encoding ISO-8859-1 is always available to applications running on emulated devices, whether or not it is listed in the `microedition.encoding.supported` entry.

CHAPTER 3

Creating an Obfuscator Plug-in

The Sun Java Wireless Toolkit for CLDC allows you to use a bytecode obfuscator to reduce the size of your MIDlet suite JAR file. The toolkit comes with support for ProGuard, as described in the *Sun Java Wireless Toolkit for CLDC User's Guide*.

If you want to use a different obfuscator, you can write a plug-in for the Sun Java Wireless Toolkit for CLDC.

3.1 Writing the Plug-in

Obfuscator plug-ins extend the `com.sun.kvem.environment.Obfuscator` interface. The interface itself is contained in *toolkit*`\wtklib\kenv.zip`.

The `Obfuscator` interface contains two methods that you must implement:

- `public void createScriptFile(File jadFilename File projectDir);`
- `public void run(File jarFileObfuscated, String wtkBinDir, String wtkLibDir, String jarFilename, String projectDir, String classPath, String emptyAPI) throws IOException;`

To compile your obfuscator plug-in, make sure to add `kenv.zip` to your `CLASSPATH`.

For example, here is the source code for a very simple plug-in. It doesn't actually invoke an obfuscator, but it shows how to implement the `Obfuscator` interface.

```
import java.io.*;
public class NullObfuscator
    implements com.sun.kvem.environment.Obfuscator {
  public void createScriptFile(File jadFilename, File projectDir) {
```

```
    System.out.println("NullObfuscator: createScriptFile()");
  }

  public void run(File jarFileObfuscated, String wtkBinDir,
    String wtkLibDir, String jarFilename, String projectDir,
    String classPath, String emptyAPI) throws IOException {
      System.out.println("NullObfuscator: run()");
    }
}
```

Suppose you save this as *toolkit*\wtklib\test\NullObfuscator.java. Then you can compile it at the command line like this:

```
set classpath=%classpath%;toolkit\wtklib\kenv.zip
javac NullObfuscator.java
```

3.2 Configuring the Toolkit

Once you've written an obfuscator plug-in, you must tell the toolkit where to find it. To do this, edit *toolkit*\wtklib\Windows\ktools.properties. Edit the obfuscator plug-in class name and tell the toolkit where to find the class. If you're following along with the example, edit the properties as follows:

```
obfuscator.runner.class.name: NullObfuscator
obfuscator.runner.classpath: wtklib\\test
```

Restart the toolkit and open a project. Now choose Project > Package > Create Obfuscated Package. In the console, the output of NullObfuscator displays, as follows:

```
Project settings saved
Building "Tiny"
NullObfuscator: createScriptFile()
NullObfuscator: run()
Wrote C:\WTK251\apps\Tiny\bin\Tiny.jar
Wrote C:\WTK251\apps\Tiny\bin\Tiny.jad
Build complete
```

Index

T