Porting Guide

Sun Java™ Wireless Client Software 2.2

Java Platform, Micro Edition

Sun Microsystems, Inc.
www.sun.com

December 2008

Contents

Figures

Tables

Preface

This guide describes how to port Sun Java Wireless Client Software 2.2 to your mobile device.

Before You Read This Book

To fully use the information in this document, you must have thorough knowledge of the topics discussed in these documents:

- JSR 118 *Mobile Information Device Profile 2.0*
- JSR 139 *Connected Limited Device Configuration 1.1*
- JSR 185 *Java Technology for the Wireless Industry*
- JSR 248 *Mobile Service Architecture*
- *CLDC HotSpot™ Implementation Porting Guide*
- *Skin Author's Guide to Adaptive User Interface Technology*
- JSR 75 *Personal Information Management and File Connection API*
- JSR 82 *Java APIs for Bluetooth*
- JSR 120 *Wireless Messaging API 1.0*
- JSR 135 *Mobile Media API*
- JSR 177 *Security and Trust Services API for J2ME*
- JSR 179 *Location API for J2ME*
- JSR 205 *Wireless Messaging API 2.0*
- JSR 211 *Content Handler API*
- JSR 234 *Advanced Multimedia Supplements API for J2ME*
- JSR 256 *Mobile Sensor API*

How This Book Is Organized

This book contains the following chapters and appendices:

Chapter 1 is a basic overview of the Java Wireless Client software and how it works.

Chapter 2 is an overview of the porting process. This chapter provides a high-level strategy for porting the Java Wireless Client software to your device.

Chapter 3 describes how to port the logging subsystem of the Java Wireless Client software.

Chapter 4 describes how to port the memory allocation APIs of the Java Wireless Client software.

Chapter 5 describes how to port the time and timer APIs of the Java Wireless Client software.

Chapter 6 describes how to port the file system APIs of the Java Wireless Client software.

Chapter 7 describes how to the port the display APIs of the Java Wireless Client software.

Chapter 8 describes how to port the event handling APIs of the Java Wireless Client software.

Chapter 9 describes how to port the keypress event APIs of the Java Wireless Client software.

Chapter 10 describes how to port the runtime lifecycle events of the Java Wireless Client software.

Chapter 11 provides a further test of your porting progress by providing instructions for running an interactive, ROMized MIDlet.

Chapter 12 describes how to port of basic networking and socket communications APIs of the Java Wireless Client software.

Chapter 13 describes how to port more advanced networking and socket communications of the Java Wireless Client software.

Chapter 14 describes how to port the font system APIs of the Java Wireless Client software.

Chapter 15 describes how to port the annunciator APIs (e.g., backlight, vibrator, flash, etc.) of the Java Wireless Client software.

Chapter 16 describes how to port predictive text input support APIs of the Java Wireless Client software. (Porting of this functionality is optional.)

Chapter 17 describes how to port native image decoder APIs of the Java Wireless Client software. (Porting of this functionality is optional.)

Chapter 18 describes how to port File Connection APIs (JSR 75) of the Java Wireless Client software.

Chapter 19 describes how to port Personal Management APIs (JSR 75) of the Java Wireless Client software.

Chapter 20 describes how to port Wireless Messaging 1.0 (SMS) APIs (JSR 120) of the Java Wireless Client software.

Chapter 21 describes how to port Wireless Messaging 2.0 (MMS) APIs (JSR 205) of the Java Wireless Client software.

Chapter 22 describes how to port of Mobile Media APIs (JSR 135) of the Java Wireless Client software.

Chapter 23 describes how to port Advanced Mobile Media APIs (JSR 234) of the Java Wireless Client software.

Chapter 24 describes how to port Content Handler APIs (JSR 211) of the Java Wireless Client software.

Chapter 25 describes how to port Security and Trust Services APIs (JSR 177) of the Java Wireless Client software.

Chapter 27 describes how to port Location APIs (JSR 179) of the Java Wireless Client software.

Chapter 26 describes how to port Landmark Store APIs (JSR 179) of the Java Wireless Client software.

Chapter 28 describes how to port Bluetooth APIs (JSR 82) of the Java Wireless Client software.

Chapter 29 describes how to port Mobile Sensor APIs (JSR 256) of the Java Wireless Client software.

Chapter 30 describes how to test your completed Java Wireless Client software port.

Using Operating System Commands

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

Typographic Conventions

The following typographic conventions are used in this guide.

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

Shell Prompts

Shell	Prompt
C shell	%

Related Documentation

The following documentation is included with this release.

TABLE P-1 Related Documentation

Application	Title
All	*Release Notes*
Building	*Build Guide*
Porting Issues and Overview	*Architecture and Design Guide*
Porting Procedures and Guidelines	*Porting Guide*
Running SJWC Software and Using Tools	*Tools Guide*
Multitasking Integration and Policies	*Multitasking Guide*
Using Adaptive User Interface Technology (skins)	*Skin Author's Guide to Adaptive User Interface Technology*
Viewing reference documentation created by the Javadoc™ tool	*Java API Reference*
Viewing reference documentation created by the Doxygen tool	*Native API Reference*

In addition, you might find the following documentation helpful:

- *The Java Language Specification* (Java Series), Second Edition by James Gosling, Bill Joy, Guy Steele and Gilad Bracha. Addison-Wesley, 2000, `http://java.sun.com/docs/books/jls/index.html`.
- The Java Specification Request (JSR), (J2ME Connected, Limited Device Configuration) at `http://jcp.org/jsr/detail/30.jsp` (JSR 30)
- Mobile Information Device Profile 2.0, at `http://jcp.org/jsr/detail/118.jsp` (JSR 118)

- Java Technology for the Wireless Industry, at http://jcp.org/jsr/detail/185.jsp (JSR 185)
- A full list of JSRs for the Java Platform, Micro Edition (Java ME platform), available at http://jcp.org/jsr/tech/j2me.jsp
- KVM Debug Wire Protocol (KDWP) Specification, Sun Microsystems, Inc., available as part of the CLDC (Connected Limited Device Configuration) download package.

Accessing Sun Documentation Online

The Source for Java Developers web site enables you to access Java platform technical documentation on the web at http://java.sun.com/reference/docs/index.html.

Sun Welcomes Your Comments

We are interested in improving our documentation and welcome your comments and suggestions. Provide feedback to Sun at http://java.sun.com/docs/forms/sendusmail.html.

1

Introduction

Integrating a Java platform stack onto a small, handheld device, such as a cellular phone, is a highly complex task. A Mobile Service Architecture (MSA) ready Java platform stack can access more than 90% of the functionality of a cellular phone and is able to stress the Application Programming Interface (API) more than any other software feature of the device.

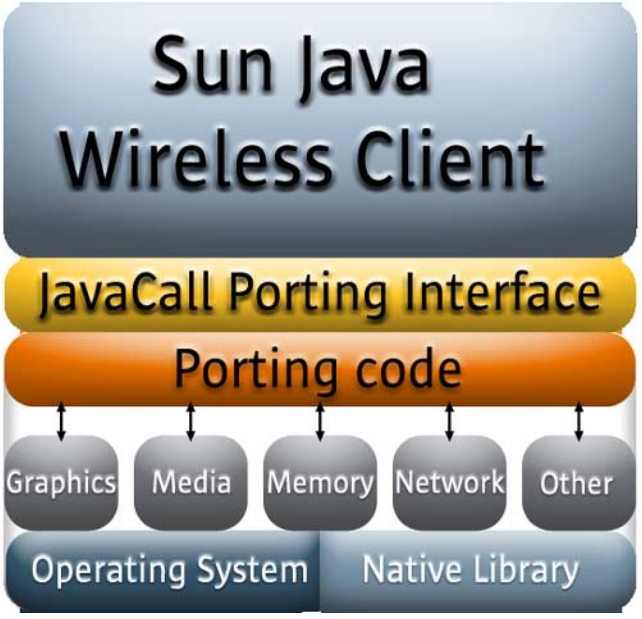
The Sun Java Wireless Client software (SJWC) provides exactly the capabilities needed for this kind of robust, small-device computing, and sets the standard for high performance and high functionality in today's wireless industry.

FIGURE 1-1 shows the SJWC JavaCall™ Porting Interface and how it is integrated into the native software stack of your device. The JavaCall API provides a set of uniform porting layer interfaces that makes mapping the Sun Java Wireless Client software to your device OS and hardware a straightforward, and much-improved experience, over previous porting arrangements.

As you will see as you work your way through this Guide, the JavaCall porting layer is used to port individual pieces of functionality one module at a time, in a progression, with each piece of functionality building on the prior one, until you have established a complete port on your device. At each step, the JavaCall porting interfaces are used to bind the functionality provided by the Sun Java Wireless Client software to the native libraries and operating system of your device.

Note – The Sun Java Wireless Client software is configured to run in Sun's Java Wireless Toolkit (WTK) Windows emulation environment. This guide provides the instructions you need to port a SJWC implementation to your handheld device.

FIGURE 1-1 The SJWC JavaCall Porting Interface, with Native Libraries and Device OS



Assuming that your device platform is stable, you will implement the JavaCall porting layer APIs one by one and Sun's Java platform stack will slot right on top. This document provides guidelines for porting Sun Java Wireless Client functionality to your device, starting with the simplest operations and proceeding through the full set of MSA-compliant optional packages provided with this release.

At the end of the Sun Java Wireless Client software porting process, you should expect to test the quality and functionality of your finished port using industry-standard Test Compatibility Kits (TCKs).

Note – It is strongly suggested that as you work through the porting process, you test each piece of functionality as you go. This testing of individual components is outside the scope of this documentation, so no instructions for conducting these tests are provided here.

Packages and Tools

The primary engineering task during the porting process is to implement the JavaCall APIs on your device platform.

The port is started by compiling and linking a "stub implementation" of these APIs for your device. This forms the skeleton of the complete port. This can be found in the subdirectory `javacall/implementation/stubs`.

2

Overview of the Porting Process

The porting process is simple in concept. Porting proceeds through a series of chapters in this guide. Each chapter has a clear objective, describes how to port closely related functionalities, and builds on the previous chapters before it.

Note – it is highly recommended that "area experts" participate in the porting of the APIs with which they are familiar.

Beginning with the simplest and most essential APIs, you will create the necessary `javacall_` functions and compile them to run on your device. At each stage, you should also plan to write enough tests to ensure that the new functionality ported to your device works as is expected.

Once a stage is fully completed, work moves on to the next stage. Links in the documentation will point you to additional resources, as needed.

The `javacall` Interfaces

The Sun Java Wireless Client software provides you with a well-documented interface for the integration of Java functionality into your device, called the `javacall` interface. JavaCall functions are grouped into subsystems. Examples of subsystems are Memory, LCD, Font, File and Directory, Sockets, and so on. Each subsystem may consist of three types of functions:

- Mandatory JavaCall functions - The device port must implement every mandatory JavaCall function. Functions that are not implemented will cause Java platform applications to fail on your device.
- Optional JavaCall functions - This group of functions can remain unimplemented, e.g. return a `JAVACALL_NOT_IMPLEMENTED` constant.

However, to expose non-mandatory device capability, relevant optional functions need to be implemented. For example, on devices that do not have audio or video recording capabilities, the function `javacall_media_start_recording()` can remain unimplemented. On the other hand, if the device supports audio or video recording, by implementing `javacall_media_start_recording()`, media recording will be supported for Java platform applications.

- Javanotify functions - Javanotify functions are C functions implemented in Sun's Java platform library that the device must call in order to communicate with the Java platform. For example, any device thread can call function `javanotify_pause()` to tell the Java platform to enter the paused state.

Getting Started

You begin your development from a "stubs" directory, in which all of the functions to be implemented exist only as stub functions that do little more than return a `JAVACALL_NOT_IMPLEMENTED` value. This is a convenient since the stubs always compile, and permit the link process to complete successfully while the code is still in development.

About one hundred `javacall` functions are needed for basic Java platform functionality. You link these javacall functions, and the resulting compiled objects, with your platform.

The chapters that follow describe each of the stages in a logical sequence. It is a good idea to follow these stages in sequence, since this process has been tried and tested on many deployments. However, if you are under severe time pressure and have many engineers, the stages after Chapter 8 can be done in parallel.

I Porting CLDC and MIDP

This part covers the following topics:

3

Porting the Logging Facility

The purpose of this chapter is to provide a means whereby runtime log messages, test output, and other text can be output.

APIs To Be Ported

The following API must ported:

```
void javacall_print (const char *s)
```

More information on the Logging APIs can be found in the following file:

```
javacall/interface/common/javacall_logging.h
```

Detailed descriptions of all of the necessary APIs can be found in the Sun Java Wireless Client Javadoc for this file.

Description

The first step in porting Java Wireless Client software to your device is to enable some basic output from the device.

The `javacall_print()` function should be mapped as simply as possible, for example, to a COM channel. The COM channel should not be shared with other parts of the device.

Quick Workaround

There are no quick workarounds for this step. Setting up logging is fundamental to beginning a successful port.

4

Porting the Memory System

The purpose of this chapter is to establish memory space within the virtual machine and provide APIs for small native allocations.

APIs To Be Ported

The APIs to be ported can be found in the following file:

`javacall/interface/common/javacall_memory.h`

Detailed descriptions of all of the necessary APIs can be found in the Sun Java Wireless Client Javadoc for this file.

Background

Porting the memory allocation APIs provides the Java Virtual Machine with a large memory block for the Java Heap (managed by the Garbage Collection subsystem) and also provides APIs for small native allocations (immovable native-code allocations, for which regular pointers can be used).

Selected API Descriptions

The following APIs must be ported:

- `javacall_memory_heap_allocate()` - This API is essential and needs to be created first. It is used to create one, large, contiguous block of memory for use by the Java platform and it will only be called once while Java is running.

When the Java platform cleans up everything and shuts down completely, the `javacall_memory_heap_deallocate()` API will be called.

- `javacall_malloc()/free()` - These API functions are needed primarily to provide small buffers to transfer information from the native context to the Java platform context. A good example is that of SMS messages passed to the Java platform while it is not running (referred to as "SMS Push").

  Most of the memory used by the Java platform comes from the single, contiguous block allocated using the `javacall_memory_heap_allocate()` API.

These APIs can be deferred to a later stage if `malloc()` and `free()` are not implemented on your platform. However, they are needed for all functions that pass events and data to and from other contexts (e.g., the event mechanism, networking, etc.) The APIs must be natively thread-safe.

Many of the other APIs are optional, and can be ignored at this stage.

Answers to Common Questions

The following are common questions asked at this stage of the porting process:

1. **What size should be used, initially?**

   Some operators require 4 MB or more, but this figure can be fine tuned at a later stage. Should not be less than 1.0MB for initial testing.

2. **Where is the memory size set?**

   The API is an "incoming" API, with Java requesting a desired memory size for the heap. The subject of memory allocation is complex and should be discussed in depth with Sun Engineering Services.

3. **Does the heap memory have to be contiguous?**

   Yes - there is no provision for a discontinuous memory heap in this version.

4. **Does the memory have to be aligned to 2, 4, 8 bytes?**

   The Java platform handles this internally. There is no need to ensure any particular alignment for this method.

Quick Workaround

If you wish to try this quickly, without actually implementing anything, do the following:

- define a static array of long
- return a pointer to it inside the function

All the other functions can remain stub implementations.

5

Setting Up Time and Timers

The purpose of this chapter is to provide a timebase for measurement and a means to "wait" for periods of time while running tests.

APIs To Be Ported

The APIs to be ported can be found in the following file:

`javacall/interface/common/javacall_time.h`

Detailed descriptions of all of the necessary APIs can be found in the Sun Java Wireless Client Javadoc for this file.

Background

Time and timer functionality is a basic requirement of many platforms, including the Java platform. In the Java platform, an application can query the current system time or schedule a time- related task, such as an alarm or a calendar event. It is a widely used and indispensable function.

The time and timer APIs define the functions for initializing a timer, cancelling a timer, getting the current time, and getting the local time zone. This chapter describes what these JavaCall APIs are, how they work, and how to port them.

Preparatory Tasks

Make sure that the accuracy of your system timer APIs are sufficient. In general, accuracy should be controlled within a few milliseconds.

Ensure that sequential calls to your timer provide values that increase monotonically. That is, each subsequent call to your timer API MUST return a value that is equal to or greater than the previous call.

Ensure that the values of time, time-zone, and Daylight Savings adjustments are properly stored and operational in your system.

Selected Timer API Descriptions

The basic mechanism for timer operation revolves around timer initialization by the Java platform (to initialize and start a regular or a cyclic timer), and callbacks that are invoked by the platform when the timer expires.

If multiple simultaneous timers are supported by your OS, the most convenient way of managing them is by using the OS-provided handle. (See the Windows implementation as an example of how this is done.) Otherwise, the JavaCall implementation needs to internally maintain a list of active timer, together with their callbacks and handles, and invoke each callback as the timers expire or cycle.

Implement the following typedef:

- `typedef void (*javacall_callback_func)(javacall_handle handle)`

  This typedef defines a function type with a single handle parameter, that returns void, and a parameter of this type is passed whenever a timer is initialized, as in the following function:

```
javacall_result javacall_time_initialize_timer(

          int wakeupInMilliSecondsFromNow,

          javacall_bool cyclic,

          javacall_callback_func   func,

          /*OUT*/ javacall_handle *handle)
```

  This creates a native timer to expire in the given time, or less. When the timer expires, the callback function must be invoked in the system context, with the correct handle. For non-cyclic timers, the platform must finalize timer resources after invoking the callback.

Selected Time API Descriptions

Ensuring accurate use of time is essential. Implement the following function:

- `char* javacall_time_get_local_timezone(void)`

  This function should return a string in the form "`GMThh:mm`". The symbol "" is either "+" or "-", and `hh:mm` is the offset from UTC time. Daylight Savings time should be handled correctly, i.e., in the United Kingdom (UK) during the Summer it will return `"GMT+01:00"`, and in San Francisco during the Winter, `"GMT-08:00"`. The string storage should be static and maintained by the function itself, and no attempt will be made to free it by the calling function.

- `javacall_time_get_milliseconds_since_1970(void)`

  Returns number of fine resolution milliseconds elapsed since midnight(00:00:00), January 1, 1970.

There are two other functions that must be implemented also:

- `javacall_time_get_seconds_since_1970(void)`

  Returns the number of seconds elapsed since midnight (00:00:00), January 1, 1970.

- `javacall_time_get_clock_milliseconds(void)`

  Returns the number of clock milliseconds.

The obvious redundancy of these three functions is to allow the Java platform to use the most suitable API and derive the other values from that API.

Note – An implementation hint: use `javacall_time_get_seconds_since_1970` and `javacall_time_get_clock_milliseconds` to get an accurate result, and then derive `"get_milliseconds_since_1970"` from these.

Answers to Common Questions

The following are common questions asked at this stage of the porting process:

1. **What accuracy is needed for the native timer?**

   It is recommended that the accuracy be better than 30 milliseconds.

2. **What is the difference between** `get_millisecond s_since_1970()` and `get_clock_milliseconds()`?

It is unusual for the system day-date clock to have millisecond accuracy and resolution. However, some systems provide an additional millisecond-accurate clock, e.g., one that measures milliseconds from system startup. The Java platform only uses `get_clock_milliseconds` for time interval counting.

Quick Workaround

The following APIs are optional:

- `javacall_time_get_seconds_since_1970()`
- `javacall_time_get_clock_milliseconds()`

If your system can't get these values directly, you can implement them using `javacall_time_get_milliseconds_since_1970()`.

6

Porting the File System APIs

The purpose of this chapter is to provide the necessary functionality for installing and executing MIDlets, by porting the file system APIs.

APIs To Be Ported

The APIs to be ported can be found in the following files:

```
javacall/interface/common/javacall_file.h
javacall/interface/common/javacall_dir.h
```

Detailed descriptions of all of the necessary APIs can be found in the Sun Java Wireless Client Javadoc for these files.

Background

In order to install, execute, and remove MIDlets the Java Virtual Machine (JVM) has to have access to the file system. During execution, MIDlets store persistent data in the Record Management System (RMS) storage, which is mapped to the file system.

The Java platform also reads and saves some internal settings, which can be loaded from the file system during JVM execution.

Preparatory Tasks

If the file system used by your platform provides POSIX-like file system functionality, you should have no problems mapping the file system APIs.

Some functions can be implemented internally if not available on the device.

Selected API Descriptions

To establish the file system functionality needed to execute MIDlets on your device, implement the following APIs:

- `javacall_file_init(void)` - This API is used only if your file system requires initialization before it can be used. In most cases this function can just return `true`.
- `javacall_file_finalize(void)` - This API is used only if your file system requires finalization after it was used. In most cases this function can just return `true`.
- `javacall_file_seek()` - This API should allow moving the implicit file pointer to a specific place in the file, but it is also used to determine where the file pointer is now. So, this function should provide current position information also. If it does not, another function that does should be implemented.
- `javacall_dir_get_free_space_for_java()` - This API is called quite a lot from the Java platform so having some kind of value-caching mechanism (or some other way to make this function light-weight) is recommended.
- `javacall_file_sizeofopenfile()` - This API can be implemented on top of `javacall_file_seek()` or `javacall_file_read()` functions. However, from a performance point of view, it is better to use a native platform counterpart.
- `javacall_file_sizeof()` - This API can be implemented on top of `javacall_file_sizeofopenfile()` if there is no relevant function, but will require opening a file, which is a heavy operation.
- `javacall_file_exist()` - This API can be implemented on top of `javacall_file_open()` if the `READ_ONLY` flag is supported for file open, but it will introduce a performance hit. Also, this function should differentiate between a directory and a file.
- `javacall_dir_get_root_path()`- Returns the root path of your Java platform's home directory. In general, all Java platform-related files are located in the root path.
- `javacall_handle javacall_dir_open()` - Opens a specified directory.
- `javacall_utf16* javacall_dir_get_next()` - Gets the next directory in a list of available directories.
- `void javacall_dir_close()` - Closes a directory.

These following functions are used by the Java platform Application Management Service (AMS) to access files related to the MIDlet installation process. In order to implement these functions, your platform must have the ability to browse a directory and access files in that directory.

- `javacall_get_file_separator()` - Returns the file separator character used by the underlying file system. This is an internal API. There are no similar APIs at the Java platform level. The accessed directory is specified by the function `javacall_dir_get_root_path()`.
- `javacall_dir_is_secure_storage()` - This API checks if the given path is located on secure storage. Secure storage is a non-removable storage that cannot be accessed by the user or overwritten by an insecure application.

Answers to Common Questions

The following is a common question asked at this stage of the porting process:

- **Should the `javacall_flush()` function be implemented?**

  If data is not flushed directly during write time and data may be lost, consider implementing the `javacall_flush()` function.

Quick Workarounds

For faster results, the following alternate APIs can be used:

- `javacall_file_truncate()`- This API is used to make a file smaller (only smaller) and can be implemented internally using a sequence such as open/read/write/close/delete/rename. However, it requires more temporary space on the file system and may introduce a performance hit.

Note – If this scheme is used, it will not be possible to use system-provided file handles. The file-handle parameter is an input-only parameter to the `javacall_file_truncate()` function and changing it here will not change any of the handles to the original file that are maintained higher up the stack. This applies to the `javacall_file_rename()` function as well.

- `javacall_file_rename()` - This API is required for MIDlet installation, but can be implemented internally using a sequence of open/write/close/delete, if required.

7

Porting the Display (LCD) APIs

The purpose of this chapter is to port the low-level graphics JavaCall primitives. After this step you will be able to view graphical output on your display.

APIs To Be Ported

The APIs to be ported can all be found in the following file:

`javacall/interface/midp/javacall_lcd.h`

Detailed descriptions of all of the necessary APIs can be found in the Sun Java Wireless Client Javadoc for this file.

Background

The JavaCall interface for low-level graphics is designed to be both high-performance and easy to port. However, it requires access to the off-screen raster buffer (backing store) and also a way to flush this buffer to the screen. Once these basic functions are ported all the graphics can be displayed.

Critical sections of the Java platform's graphics code have been written in assembly with performance as the most important criteria. It is worth taking additional time to ensure that your implementation of these graphical functions (especially the "flush" functions) is as high-performance as possible.

Preparatory Tasks

Make sure that your Java platform can access the display buffer and directly modify it.

Selected API Descriptions

To establish the functionality needed to display low-level graphics on your device, implement the following APIs:

- `javacall_lcd_get_screen()` - This API is the essential one that needs to be created first. It is used to receive the pointer to the LCD off-screen raster and the dimensions of the currently available screen. In the case that more than one display is supported on the device, the `screenType` parameter determines which screen buffer is requested.
- `javacall_lcd_init()`- This API is called during JVM startup, allowing the platform to perform device-specific initialization.
- `javacall_lcd_flush()` - This API is used for flushing the off-screen raster to the LCD display, and should be implemented as efficiently as possible, for example by using DMA.

Note – If screen raster lines are smaller than the memory addresses they occupy, the `lcd_flush()` and `lcd_flush_partial()` functions should deal appropriately with the necessary padding. Internally, the Java platform treats the screen as an image without taking any padding into account.

Answers to Common Questions

The following are common questions asked at this stage of the porting process:

1. **Why is my Display distorted?**

   Make sure that the implementation of `javacall_flush()` does not access memory areas that are not supposed to be modified, such as the status bar. See also the note below regarding pixel types. If you are using DMA to flush the screen, please make sure you are flushing the CPU data-cache before running the DMA copy.

2. **Why does the device crash every time I flush to the LCD?**

   Make sure to check the coordinates passed to `javacall_lcd_flush_partial()` are inside the LCD dimensions.

3. **Why do MIDlets that use full screen mode cause distortion in the display?**

   Make sure that after `javacall_lcd_set_full_screen_mode()` is called the correct screen dimensions are returned by `javacall_lcd_get_screen()`.

Note – `RGB2PIXELTYPE`-- Most implementations use encoded pixels as two bytes (`RGB 565`) format, as shown below.

```
-----------------------------------------

| 5 bits Red | 6 bits Green | 5 bits Blue |
-----------------------------------------
```

This macro assumes the screen memory is big-endian. If your screen memory uses little endian, as shown below, it needs to be changed.

```
-------------------------------------------------------------------------

| 3 low bits of Green | 5 bits Blue | 5 bits Red | 3 high bits of Green |

-------------------------------------------------------------------------
```

Quick Workaround

Implementing the function `javacall_lcd_flush_partial()` is mandatory and intended to increase graphical performance. If your device does not have this capability, you can ignore the coordinates given and flush the entire buffer.

8

Porting for Event Handling

The purpose of this chapter is to allow events to be delivered to the Java platform.

APIs To Be Ported

The APIs to be ported can all be found in the following file:

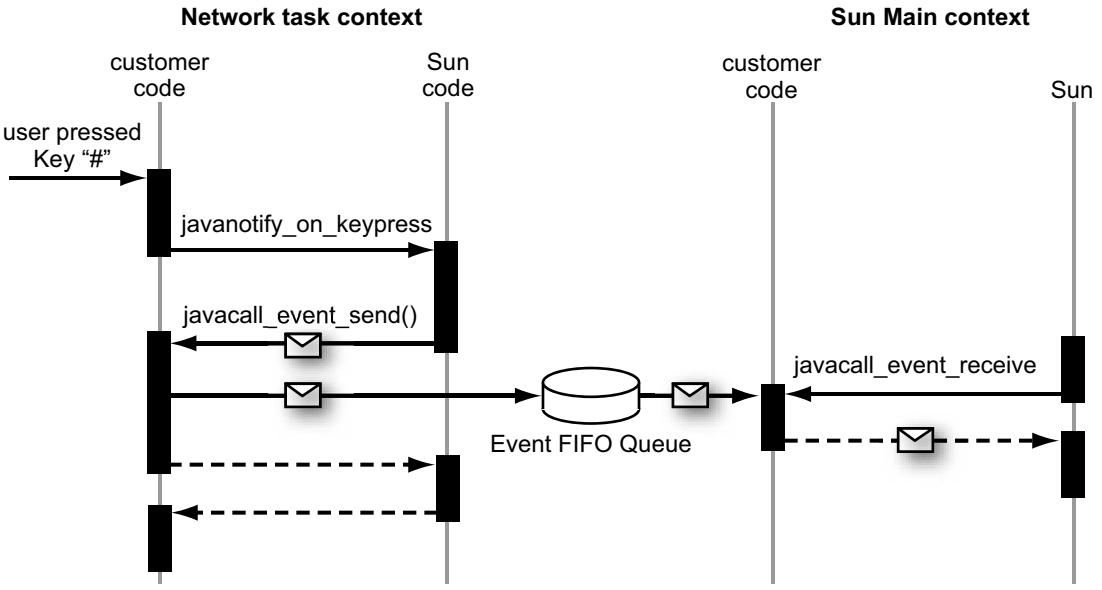
`javacall/interface/common/javacall_events.h`

Detailed descriptions of all of the necessary APIs can be found in the Sun Java Wireless Client Javadoc for this file.

Background

The Java platform needs to port the `javacall_event` APIs to receive and handle external events. See FIGURE 8-1 for details on how the JVM interacts with the Java platform via events passing.

FIGURE 8-1 Virtual Machine Events Passing



In this discussion, the terms "Sun Main Context" is used to refer to the thread of execution that is the Java platform task, and the term "Platform Context" to refer to everything else. In the JavaCall model illustrated in FIGURE 8-1, the Java platform provides an event passing mechanism (such as an OS FIFO queue) through which events can be passed safely and sequentially.

When an event such as a keypress is to be delivered to the Java platform, your system platform calls the `javanotify_on_keypress()` function. This function, which executes in the Platform Context, then invokes `javacall_event_send()`. The platform code must store this event in some thread-safe artifact such as an OS queue.

In the Sun Main Context, where the Java platform is running normally, it invokes `javacall_event_receive()` periodically to check if external events have been delivered. When all Java platform threads are sleeping (i.e., there is no need at this point in time to execute bytecode) it invokes and blocks on `javacall_event_receive()` in order to wait for external events.

If an event is available, `javacall_event_receive()` consumes the event and returns it to the Java platform. Internally, the Java platform handles the event. Usually, this is done by waking the Java platform thread that is registered as waiting for this event.

Selected API Descriptions

The event APIs require fully-functional memory/time/timer JavaCall APIs. Ensure that these subsystems are working correctly before attempting to port the JavaCall events subsystem. The underlying event queue must be thread-safe and must maintain a FIFO ordering.

Note – The messages stored in the queue are generic binary fixed-length messages.

Implement the following two functions:

- `javacall_event_receive()` - This function is called by the JVM to receive platform events such as key press, SMS, Networking, lifecycle, and multimedia events.

  The first parameter, `timeTowaitInMillisec`, decides how the JVM receives events. Event receiving can be either in polling mode (`timeTowaitInMillisec = 0`), infinite duration blocking mode (`timeTowaitInMillisec = -1`) or limited duration blocking mode (`timeTowaitInMillisec` is a positive value).

  In either of the two blocking modes:

  - The Sun Main context is blocked and is not consuming CPU cycles.
  - The display can still be active (and backlit).
  - Other native functions can still be working.
  - Battery can still be consumed.

  See Chapter 10 for discussion of pausing or shutting down the Java platform completely.

  The execution style of the Java platform is to block if possible, and poll if necessary, thus keeping CPU cycle execution to a minimum.

- `javacall_event_send()` - This function is called by the JVM to send a message to a Java platform event FIFO queue dedicated to listening for the task. The Java platform should put the message into the task event queue as a binary message. If the Sun Main execution context is blocking on `javacall_event_receive()` for events, `javacall_event_receive()` should be unblocked and return this message to the Java platform task.

Note – This discussion assumes Master Mode event handling.

Answers to Common Questions

The following are common questions asked at this stage of the porting process:

1. **Why can the Java platform not use blocking APIs?**

   Blocking APIs are very simple to use - if you are waiting for a keypress event you call an API that blocks until the event occurs. However, the Java platform allows many parallel threads of execution to occur at the same time, all running in one native context. If a native API is called that blocks, the whole Java Virtual Machine is blocked (not just the Java platform thread that made the call).

   For this reason the `javacall_event_receive()` function MUST be available in all three modes - poll for event, block with timeout, and block indefinately. Only in this manner can the Java platform threads be scheduled correctly.

2. **Why is a thread-safe FIFO event queue required for the Java platform?**

   Events always happen in the Platform Context and not in the Sun Main Context, as the Sun Main event queue is usually shared between multiple tasks. As an example, events can flow to the Java platform from the networking, keypress and sound systems independently. So, the Sun Main event queue must be thread-safe (task-safe). Meanwhile, to make sure a Java platform task will get and handle events in the correct order, the event queue must be FIFO.

3. **What are the requirements on event queue space and the size of individual Java event?**

   The Java platform event queue must have the ability to buffer at least 20 event messages. The platform should also allow the size of individual events to be up to 4K (for the data blocks, which are pointed to by the event struct).

4. **How does the JavaCall event APIs' implementation impact performance?**

   The Java platform's response time to external events (such as a keypress) is directly affected by the efficiency of the native event APIs and the underlying OS mechanism. Event JavaCall APIs are hot routines and their efficiency seriously impacts the overall performance of the JVM. In particular, `javacall_event_receive()` function must be very efficient.

Quick Workaround

Unfortunately, there are no workarounds for the two `javacall_event` APIs. `javacall_event_send()` and `javacall_event_receive()` must be implemented.

9

Porting for Keypress Events

The purpose of this chapter is to deliver all standard keypress events to the Java platform.

APIs To Be Ported

The APIs to be invoked can all be found in the following file:

`javacall/interface/midp/javacall_keypress.h`

Detailed descriptions of all of the necessary APIs can be found in the Sun Java Wireless Client Javadoc for this file.

Background

Keypress events use the events subsystem that was ported in Chapter 8. The MIDP 2.0 specification is designed for cellular phones with numeric keyboards, direction keys, game-keys, and touch-screens. These key events are described in the documentation for your device.

When a key is pressed, released, or repeated (“long-press”), an event must be delivered to the Java platform. Keypress events should only be delivered to the Java platform when it has the “foreground” or “input focus” of your system.

Preparatory Tasks

Identify the place in the system software where keypress events are delivered. Implement any logic used to determine that the Java platform is the current focus of keypress events.

Selected API Descriptions

There is one API that handles keypress events: `javanotify_key_event()`. This is a `javanotify_` function call that has already been implemented inside the Java platform and is invoked in the Platform Context whenever a key is pressed.

Note – In general, `javanotify_` APIs are invoked in the Platform Context and `javacall_` APIs are invoked in the Sun Main Context.

Every time a key is pressed, released, or enters the "repeat" mode, an event should be sent using the following API, which must be implemented:

- `javanotify_key_event(javacall_key key, javacall_keypress_type type)`

For the appropriate values for `javacall_key` and `javacall_keypress_type`, see the header file and documentation.

10

Porting Runtime Lifecycle Events

The purpose of this chapter is to port and control the runtime lifecycle events of a Java platform MIDlet, e.g. Start, Shutdown, Pause and Resume.

APIs To Be Ported

The APIs to be ported can all be found in the following file:

`javacall/interface/midp/javacall_lifecycle.h`

Detailed descriptions of all of the necessary APIs can be found in the Sun Java Wireless Client Javadoc for this file.

Background

The native platform needs to signal the Java platform to start, shutdown, pause and resume by sending some dedicated events to the JVM. The Java platform provides `javanotify()` APIs to send events to the dedicated Sun Main Context (as described in Chapter 8) and this in turn controls the Java platform start, shutdown, etc.

After an operation completes, the JVM is switched to be in the new state and notifies the native platform by invoking `javacall_lifecycle_state_changed()`. The state of the native platform is then also changed.

Preparatory Tasks

The native platform should dedicate a specific task for the Java platform at device start time and never end it until device shutdown. This task should start the "external event loop" by calling into function `JavaTask()`, which never returns.

`JavaTask()` Entry Point

The `JavaTask()` is expected to be running throughout device uptime and the native platform is required to create a dedicated task and execute the Java platform entry point function in the task. The C entry point for the Java platform task is the function `JavaTask()`.

In the Java platform implementation, this function waits for a `javanotify_start()` signal. Once this is received it will start up the Java platform. In the native test suite version of `JavaTask()`, this function can be used to run native tests.

Passing Events To the Java Platform

The native platform passes event notifications to the Java platform task by calling a set of functions called `javanotify_` functions. For example, the native platform may call the function `javanotify_pause()` to request that the JVM be paused, or `javanotify_shutdown()` to shutdown the running JVM.

`javanotify_` functions are executed in the Platform Context and use the event passing API (see Chapter 8) to pass a notification to `JavaTask()`.

Selected API Descriptions

The following APIs must be implemented:

- `javanotify_midlet_start()` - This function should be invoked by the native platform in the Platform Context to start a Java platform MIDlet. For example, this function can be invoked if a native AMS is used and the user selects a particular MIDlet to run.
- `javanotify_start()` - This function should be invoked by the native platform in the Platform Context to start the Java platform AMS. For example, this function can be invoked when the user selects the "Games and Applications" main-menu item.

- `javanotify_pause()` - This function should be invoked by the native platform in the Platform Context to pause the Java platform.
- `javanotify_resume()` - This function should be invoked by the native platform in the Platform Context to end, pause, and resume the Java platform.

11

Milestone One: Running a ROMized Interactive MIDlet

You have reached the point in the porting process where it should be possible to run one or more applications on your device. In addition to file system capability, menu mappings, and a simple display, your device also has the ability to handle keypress, lifecycle, and external events.

To test the capability and stability of your port, run an interactive MIDlet that takes advantage of these new capabilities, as shown in .

FIGURE 11-1 Running a Java Platform MIDlet

12

Porting Basic Networking and Socket Communications

The purpose of this chapter is to enable the Java platform to communicate with other devices over a socket-based connection.

APIs To Be Ported

The APIs to be ported can be found in the following files:

```
javacall/interface/midp/javacall_network.h
javacall/interface/midp/javacall_socket.h
```

Detailed descriptions of all of the necessary APIs can be found in the Sun Java Wireless Client Javadoc for these files.

Background

Networking is a critical functionality in the Java platform and is surprisingly difficult to get right. The APIs described in this section are the minimum APIs needed for basic client socket communications to function on your device.

Due to the complexity and asynchronous nature of networking stacks on all devices, platform networking APIs are often asynchronous by design. (In the rare case that the native APIs are synchronous and blocking, additional native threads must be used so that the Java platform can continue to run while the operation is in progress.)

For this reason, the networking API implementations are split into two phases, the first suffixed with `_start` and the second with `_finish`.

Preparatory Tasks

It is advisable to write and run a small native program that does the following things:

- Initializes a network connection
- Opens a socket to a fixed IP
- Writes and reads some data
- Closes the socket
- Finalizes the network connection

Such a program can be used to smoke-test the networking stack on the device and will provide much useful information for the porting effort.

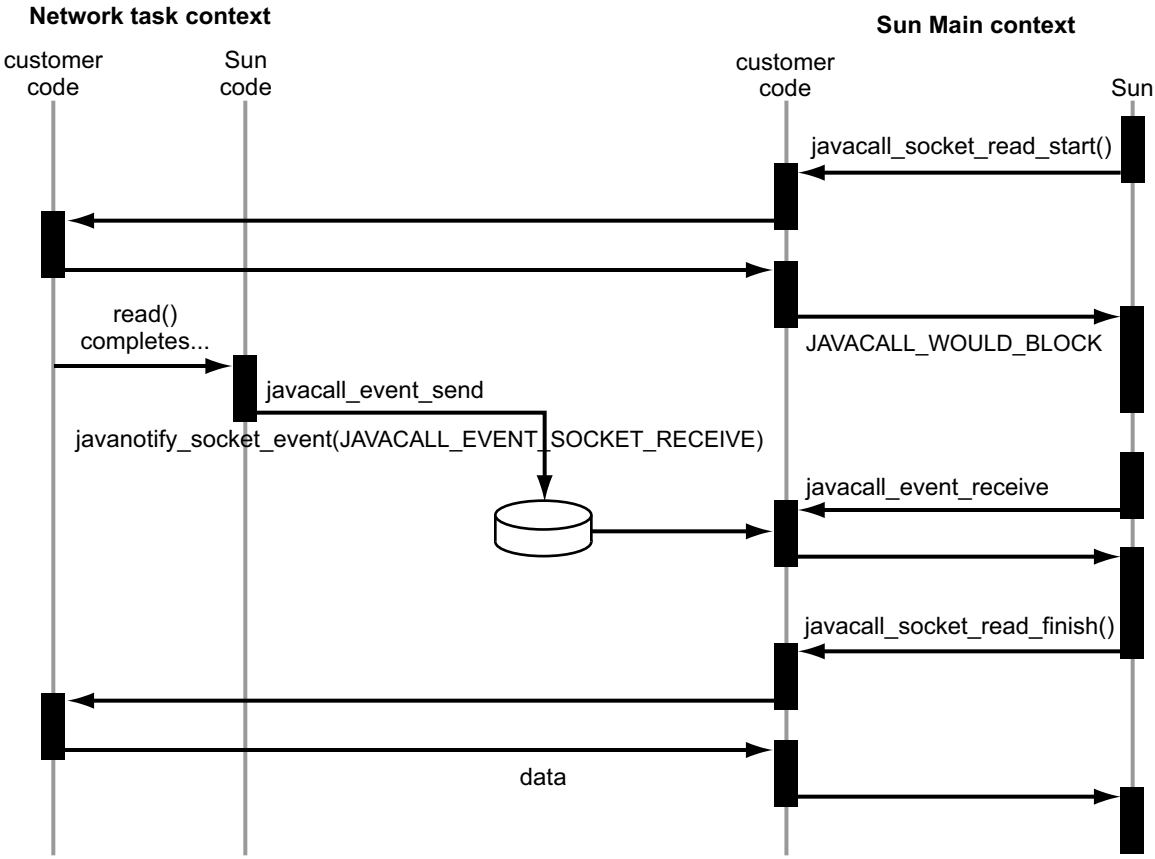
Sequence of Operation

The `javacall_socket_read_start()` call returns `JAVACALL_WOULD_BLOCK` in the case where data is not immediately available. In fact, if all data is available without need of blocking, the call can return `JAVACALL_OK` and include data. In this case, `javacall_socket_read_finish()` is not called. This behavior is described in detail in the header files.

Although `javacall_socket_read()` is given as an example, all of the networking and socket APIs in this chapter are designed to work asynchronously, so it is important to thoroughly understand the mechanism.

A typical sequence flow for `javacall_socket_read()` is shown in FIGURE 12-1.

FIGURE 12-1 Typical Sequence Flow for `javacall_socket_read()`



Common API Parameters

The following are common JavaCall API parameters:

- `javacall_handle* pHandle` - The handle is opaque data (a void*) that serves both to represent an open connection and also as the name for a pending I/O operation.

  The handle is used by the Java platform to keep track of threads that are blocked, waiting for this I/O operation to complete. The handle must be unique; that is, it must not be possible for any two currently pending I/O operations to return the same handle.

- `void **pContext` - It is recommended that implementations allocate a context structure on the C-heap in the `_start()` function and return a pointer to the caller. This context structure can contain any platform-specific information necessary to keep track of the pending I/O operation, such as a return data buffer and status codes.

  The context structure should be freed when the `_finish()` function completes. If the implementation of a particular I/O operation has no need for a context, it must set the context pointer to `NULL` in the `_start()` function.

Selected API Descriptions

The specific subset of APIs to be ported in `javacall/interface/midp/javacall_network.h` are as follows:

- `javacall_network_init_start()` – This function performs platform-specific initialization of the networking system. Is called ONCE during JVM startup before opening a network connection. For example, this API can open PPP, or some similar protocol.

  When this asynchronous operation completes, `javanotify_network_event()` should be called with `JAVACALL_NETWORK_UP`. The Java platform then invokes `javacall_network_init_finish()` to complete the transaction.
- `javacall_network_init_finish()` - This function is invoked by the Java platform to complete a network transaction, as described in the bullet above.
- `javacall_network_finalize_start()` - This function starts a network finalize operation.
- `javacall_network_finalize_finish()` - This function finishes a network finalize operation.
- `javanotify_network_event()` - The function sends a notification callback about a network event.

The `javanotify_network_event()` must be invoked by the platform as necessary. See the header file for a description of the events that should be reported using this API, and the parameters to use.

Note – All the APIs should fail cleanly; that is, if during network initialization `javacall_network_init_start()` allocates some resources, it must attempt to clean them up (e.g., deallocate buffers, release network resources) before returning a value of `JAVACALL_FAIL`. See the header file for a description of the events that should be reported using this API and the parameters to use.

The specific subset of APIs to be ported In `javacall/interface/midp/javacall_socket.h` are as follows:

- `javacall_socket_open_start()` - This function starts a network socket open operation.
- `javacall_socket_open_finish()` - This function completes a network socket open operation and signals that the socket is ready to be used.
- `javacall_socket_read_start()` - This function starts a read operation over an open network socket.
- `javacall_socket_read_finish()` - This function completes a read operation over an open network socket.
- `javacall_socket_write_start()` - This function starts a write operation over of an open network socket.
- `javacall_socket_write_finish()` - This function completes a write operation over an open network socket.
- `javacall_socket_close_start()` - This function starts a close operation on an open network socket.
- `javacall_socket_close_finish()` - This function completes a close operation on an open network socket.
- `javanotify_socket_event()` - The function sends a notification callback about a socket event.

The `javanotify_socket_event()` must be invoked by the platform as necessary. See the header file for a description of the events that should be reported using this API, and the parameters to use.

13

Porting for Advanced Networking and Socket Communications

The purpose of this chapter is to complete the implementation of networking and socket communications.

APIs To Be Ported

The APIs to be ported can be found in the following files:

```
javacall/interface/midp/javacall_network.h
javacall/interface0midp/javacall_socket.h
```

Detailed descriptions of all of the necessary APIs can be found in the Sun Java Wireless Client Javadoc for these files.

Background

Some of the advanced networking and socket communication APIs are essential if the networking stack is to operate normally. For example, if `javacall_network_gethostbyname()` is not provided, each Java platform application will need to know the IP address of every URL it attempts to open, together with any proxy information.

However, other advanced networking and socket communication APIs are optional. For example, if the device cannot support server-sockets, all of the server-socket related APIs, such as `server_socket_open()`, `server_socket_accept()`, and `server_socket_set_notifier()`, should return `JAVACALL_NOT_IMPLEMENTED`.

Selected API Descriptions

There are many more networking and socket communications APIs than the ones provided in this section. The APIs shown here the minimal ones that should be ported to ensure that networking and socket communications can be implemented on your device.

The `javacall_network.h` APIs

In `javacall/interface/midp/javacall_network.h`, the following subset of APIs must be ported:

- `javacall_network_gethostbyname_start()` - This function starts a network operation to acquire a machine host name.
- `javacall_network_gethostbyname_finish()` - This function completes a network operation to acquire a machine host name.
- `javacall_network_error()` - This function transmits the error code for a network error.
- `javacall_network_get_local_host_name()` - This function gets the name of a local host machine.
- `javacall_network_get_local_ip_address_as_string()` - This function gets the name of a local machine by IP address.
- `javacall_network_get_http_proxy()` - This function gets the name of local network proxy machine.
- `javacall_network_gethostbyaddr_start()` - This function starts an operation to get the name of one or more host machines from the network address maps.
- `javacall_network_gethostbyaddr_finish()` - This function completes an operation to get the name of one or more host machines from the network address maps.
- `javacall_network_getsockopt()` - This function gets a socket opt.
- `javacall_network_setsockopt()` - This function sets a socket opt.
- `javacall_server_socket_set_notifier()` - This function sets a notifier to be returned when an event occurs on a socket.
- `javanotify_network_event()` - This function is returned when a network event occurs on a socket.

The `javanotify_network_event()` must be invoked by the platform as necessary. See the header file for a description of the events that should be reported using this API, and the parameters to use.

The `javacall_socket.h` APIs

In `javacall/interface/midp/javacall_socket.h`, the following subset of APIs must be ported:

- `javacall_socket_available()` - This function polls a socket to see if it is available.
- `javacall_socket_shutdown_output()` - This function shuts down output from a socket connection.
- `javacall_socket_getlocaladdr()` - This function gets an address for a local socket.
- `javacall_socket_getremoteaddr()` - This function gets an address for a remote socket.
- `javacall_socket_getlocalport()` - This function gets a local port and associates it with a socket.
- `javacall_server_socket_open_start()` - This function begins an open operation on a socket.
- `javacall_server_socket_open_finish()` - This function finishes an open operation on a socket.
- `javacall_server_socket_accept_start()` - This function begins an accept operation in which it receives input from a socket connection.
- `javacall_server_socket_accept_finish()` - This functions finishes an accept operation in which it completes receiving input from a socket connection.
- `javanotify_socket_event()` - This function is returned when an event occurs on a socket.

The `javanotify_socket_event()` must be invoked by the platform as necessary. See the header file for a description of the events that should be reported using this API and the parameters to use.

14

Porting the Font System

The purpose of this chapter is to the implement the font API on your device.

APIs To Be Ported

The APIs to be ported can all be found in the following file:

`javacall/interface/midp/javacall_font.h`

Detailed descriptions of all of the necessary APIs can be found in the Sun Java Wireless Client Javadoc for this file.

Background

In MIDP, applications request fonts based on font attributes and the platform should attempt to provide a font that matches the requested attributes as closely as possible.

A font's attributes are style, size, and face. Values for the style attribute may be combined using the bit-wise OR operator, whereas values for the other attributes may not be combined.

The font API provides the information needed to map sequences of characters to sequences of glyphs, and to render sequences of glyphs to the video memory (VRAM) or to off screen memory images.

Definitions of Font Measurements

Latin-based fonts use terms such as “height,” “ascent,” “descent,” “baseline,” and “leading” (pronounced led-ing as in the metal “lead”). The following diagram shows how these terms are used.

FIGURE 14-1 Terms Used to Describe Fonts

In alphabetic fonts (based on Latin or Greek), text rises (ascends) above the baseline, and drops (descends) below the baseline. The gap between the lowest possible character and the highest possible character on the next line is the “leading.” Leading is always below the text.

Selected API Descriptions

The MIDP 2.0 Specification requires a minimum of one font, which all the combinations of style, size and face can map to. Most modern implementations will implement many more combinations, though few systems support all 72 variations of font. However, all of the various size, style, and face combinations MUST be mapped to an existing font.

Font Attributes

Fonts should be one of three sizes: Small, Medium or Large.

Fonts can have a combination of four styles: Plain, Bold, Italic and Underline. These are defined with hexadecimal values that can be logically OR-ed together.

Fonts can have one of three faces: System, Monospace, or Proportional.

The requested combination of size, style and face are called the “font attributes,” and are set by calling the `javacall_font_set_font()` function:

```
javacall_result javacall_font_set_font(javacall_font_face face,

javacall_font_style style, javacall_font_size size);
```

These attributes must persist between calls to all LCD functions and between pause and resume operations.

Font Measurements

Java platform applications need to obtain font measurements. For this, use the `javacall_font_get_info()` function, as shown here:

```
javacall_result javacall_font_get_info(

javacall_font_face face, javacall_font_style style,

javacall_font_size size, /*out*/ int* ascent,

/*out*/ int* descent, /*out*/ int* leading);
```

The three `/*out*/` parameters must provide information about the various areas of the font’s height, as shown in the diagram above. Ask your graphic designer to provide you with this information. If the information provided by this function is incorrect, it can cause layout problems. It is a common cause of misaligned text on the screen.

Reporting Font Width

The next function returns information about font width, or more precisely string width. The function is given font attributes and a string, and returns the advance width of the characters in `charArray`. The advance width is the horizontal distance that would be occupied if the characters were to be drawn using this font, including the inter-character spacing that follows the characters, which is necessary for proper positioning of any subsequent text.

Note – The `charArray` is not null terminated.

```
int javacall_font_get_width(javacall_font_face face,

javacall_font_style style, javacall_font_size size,

const char* charArray, int charArraySize);
```

Drawing the Font

The `javacall_font_draw()` function is used for drawing the font to a buffer (which can be either the screen, a back-buffer, or an image), as shown here:

```
javacall_result javacall_font_draw(javacall_pixel color,

int clipX1,

int clipY1,

int clipX2,

int clipY2,

javacall_pixel* destBuffer,

int destBufferHoriz,

int destBufferVert,

int x,

int y,

const javacall_unicode* text,

int textLen)
```

Every one of these parameters is necessary and it is up to the function to correctly clip drawn text to fit within the boundaries set by the parameters of this function. The parameters describe a color, a destination buffer, a clip-region, a starting position for the text drawing, and the text to be printed. The following diagrams show how this works.

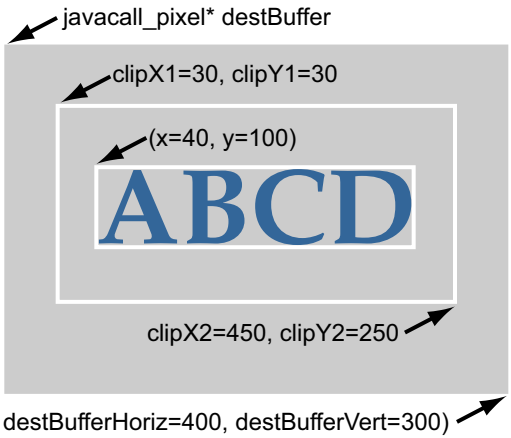
Example 1: Simple Call Where Clipping is Not in Use

In this example, as shown in FIGURE 14-2, clipping is not used and all of the text is drawn:

```
javacall_unicode abcde[]=['a', 'b', 'c', 'd', 'e'];

javacall_font_draw(color,

                   30,  //clipX1

                   30,  //clipY1

                   450, //clipX2

                   250, //clipY2
```

```
                    destBuffer
                    400, // destBufferHoriz
                    300, // destBufferVert
                    40,   //x
                    100, //y
                    abcde,
                    5);
```

FIGURE 14-2 Drawing the Font, Without Clipping



Example 2: Some Clipping Required

In this example, as shown in FIGURE 14-3, the clipping is binding and so only part of the text is shown:

```
javacall_font_draw(color,
                    30,   //clipX1
                    30,   //clipY1
                    100, //clipX2
                    250, //clipY2
```

```
destBuffer,
400, // destBufferHoriz
300, // destBufferVert
40,   //x
100, //y
abcde,
5);
```

FIGURE 14-3 Drawing the Font, With Some Clipping

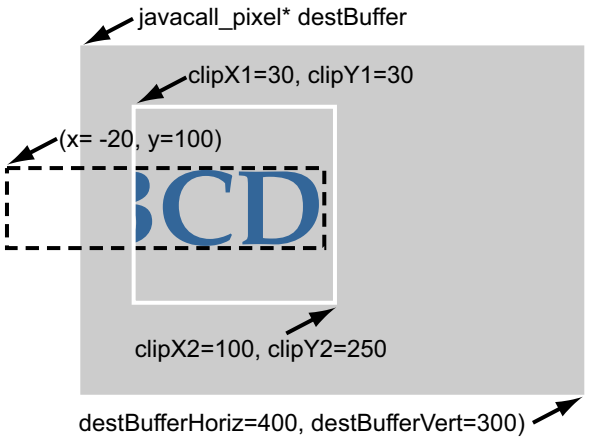
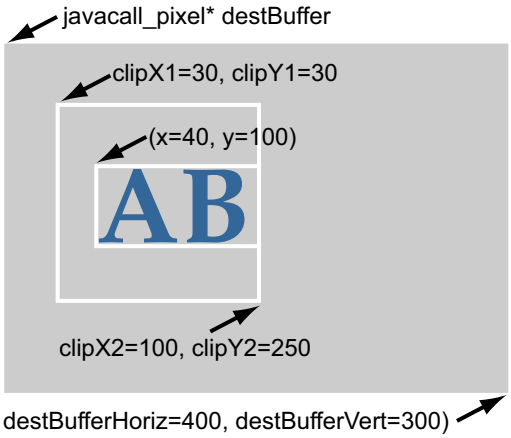


FIGURE 14-4 Drawing the Font, With More Clipping Required



Example 3: More Clipping Required

In this example, as shown in , the clipping rectangle is binding, and the x coordinate is negative.

Note – In , clipping is required in all four directions, if necessary.

Answers to Common Questions

The following are common questions asked at this stage of the porting process:

1. My Indic fonts have a different baseline. Can I use this?

No. Your font will still need to support Romanji (alphabetic) characters, not just Kanji, Hiragana, Katakana, or Gurmukhi in order to pass TCK. Note also that MIDP recognizes only the alphabetic baseline and ignores any other baselines of the font (such as a hanging Indic baseline or an Ideographic baseline).

You have the following options:

- Create your alphabetic (Romanji) font without descenders. This will look extremely unusual to native Latin/English readers, but will be MIDP 2.0 compliant.
- Base all of your Ideographic characters (such as Kanji) on the Descent line and alphabetic characters on the Baseline. Report (always) the attributes of the alphabetic characters in the APIs. If your font supports Indic glyphs (such as Devanagari, Gurmukhi, and Bengali) you should use a different hanging baseline to create the font, but still report the alphabetic baseline attributes in the APIs.

  The second option is recommended.

Quick Workaround

The Java platform has a default glyph, which can be used if no glyphs are available on the platform. To use the Java platform's default glyph, just return `JAVACALL_FAIL` for all of the Font functions.

15

Porting the Annunciator

The purpose of this chapter is to describe porting of the phone vibrator, flash_backlight, audible tone, and status icons for network, secure connection, trusted mode, and input mode.

APIs To Be Ported

The APIs to be ported can all be found in the following file:

`javacall/interface/midp/javacall_annunciator.h`

Detailed descriptions of all of the necessary APIs can be found in the Sun Java Wireless Client Javadoc for this file.

Background

The Java Technology for the Wireless Industry and MIDP 2.0 specifications require the operation of several essential types of indicators, which collectively are called “annunciators.”. The annunciators described in this chapter are:

- Vibrate
- Backlight
- Input indicator icon
- Trusted indicator icon
- Network indicator icon
- Play preset audible tone

These standard annunciators have clearly defined roles in the Java platform.

Preparatory Tasks

Since there is no programmatic way to verify that these annunciators are working correctly, it is a good idea to write small test-programs that operate the annunciators, to ensure they are working correctly. Be sure you can activate each one of the phone functions you want the API to control:

- Vibrator
- Backlight
- Input indicator icon
- Trusted indicator icon
- Network indicator icon
- Play preset audible tone

Selected API Descriptions

The following APIs must be implemented:

- `javacall_annunciator_vibrate()` - This function turns your device's Vibrate on and off.
- `javacall_annunciator_flash_backlight()` - This function sets the device backlight either bright or dim, based on the boolean value provided.

  Note that the Java platform specifies a "flashing effect," which is intended to attract the user's attention, or act as a special effect for games. However, there is no Java platform API that directly controls the state of the backlight (in Java platform terms, the backlight is either flashing or not flashing). The JavaCall API sets an absolute setting: the backlight is either bright or dim.
- `javacall_annunciator_display_trusted_icon()` - This function turns the trusted indicator icon off or on, for signed MIDlets.
- `javacall_annunciator_display_network_icon()` - This function controls the network LED or equivalent network indicator.
- `javacall_annunciator_display_secure_network_icon()` - This function controls the secure connection indicator.
- `javacall_annunciator_display_input_mode_icon()` - This function notifies the platform to show the current input mode.
- `javacall_annunciator_play_audible_tone()` - This function plays a sound of the given type.

Answers to Common Questions

The following is a common question asked at this stage of the porting process:

1. **Should my javacall implementation take the phone mode (“manner mode”) into account?**

   Yes. When porting `javacall_annunciator_vibrate()`, the programmer needs to take into consideration the current state of the phone. If the phone is in “silent all” mode, this function should do nothing.

16

Porting Predictive Text Input Support (Optional)

The purpose of this chapter is to add Predictive Text Input as a Java platform input method.

APIs To Be Ported

The APIs to be ported can all be found in the following file:

`javacall/interface/midp/javacall_pti.h`

Detailed descriptions of all of the necessary APIs can be found in the Sun Java Wireless Client Javadoc for this file.

Background

Predictive Text Input (PTI) is for small mobile devices that only have numeric keyboards. Using predictive text input makes it faster and easier to type on small mobile devices. Users can enter words with a single key press for each character. For example, to write the word “how,” user can press once “4,” “6” and “9,” eliminating the need for repeated key presses in standard text entry mode.

Predictive text input is just another text input mode, similar to upper-case text entry mode, lowercase text entry mode, numerical text entry mode, and foreign language text entry mode (Japanese, Greek etc.)

The predictive text input mechanism is not mandatory for JTWI-compliance. However, it may improve the user’s experience on most handsets.

Preparatory Tasks

Predictive Text Input APIs assume that the handset device already has a PTI dictionary and corresponding functionality. This API is used to expose the dictionary for use by the Java platform.

Selected API Descriptions

The APIs in this section assume that PTI is available and working on the handset for native programs such as SMS text-entry, etc. Consequently the API is designed to expose the native PTI functionality to the Java platform and not to implement PTI from scratch.

The PTI API uses the notion of an iterator, which supports three kinds of operations:

- Adding a keypress - A new keypress can be appended to the iterator. For example, a digit '2' can be added to the iterator by calling `javacall_t9_add_key()`.
- Clearing keypresses - A keypress can be cleared from the suffix of the iterator. Adding keys '2' '3' , and '4' and calling `javacall_t9_backspace(handle)` will change the iterator's state to contain keys '2' and '3'. All keypresses can be cleared by calling `javacall_t9_clear_all()`.
- Go Over Completion Options - The PTI API offers functions to go over all available completion options for the current keypresses.

Initialization

The `initialization` function is used to initialize Java platform predictive text input dictionaries. It uses the following APIs:

- `javacall_result javacall_pti_init(void)` - This function will be called only once, during JVM startup, and before any other PTI function is called.
- `javacall_handle javacall_pti_open(void)` - Create a new PTI iterator instance. The language should be set by default to the locale language. Only one iterator is used concurrently.
- `javacall_result javacall_pti_close(javacall_handle handle)` - This function deletes a PTI iterator.
- `javacall_result javacall_pti_set_dictionary(javacall_handle handle javacall_pti_dictionary dictionary)` - This function sets a dictionary for a PTI iterator. All newly created iterators are by default set to the default locale language. This function can be called to change the default language of the dictionary.

Keypress

A PTI iterator holds the state of all entered keypresses. A keypress is a key in the range '0'-'9' that the user presses on the handset device. The following keypress APIs are used:

- `javacall_result javacall_pti_add_key(javacall_handle handle, javacall_pti_keycode keyCode)` - This function adds a key press to a PTI iterator.
- `javacall_result javacall_pti_backspace(javacall_handle handle)` - This function removes the last keypress from the end of the list of keypresses.
- `javacall_result javacall_pti_clear_all(javacall_handle handle)` - This function clears all text from the PTI iterator.

Traversal

When traversing for all completion options for the current iterator's keypresses:

- First determine if more completions are available
- Get the next completion
- Rewind the iterator, so get next completion returns the first completion option

The following functions exist for traversing all completion options for the current iterator's keypresses.

- The function shown here returns the current T9 completion option:

```
int javacall_pti_completion_get_next(javacall_handle handle,
      javacall_utf16* outString, int outStringLen)
```

- `javacall_pti_completion_has_next(javacall_handle handle)` - This function is used to see if further completion options exist for the current PTI entry.
- `javacall_pti_completion_rewind(javacall_handle handle)` - This function is used to reset completion options for the current PTI entry.

  After this call, the function `javacall_pti_completion_get_next()` will return all completion options starting from the 1st option.

Answers to Common Questions

The following is a common question asked at this stage of the porting process:

1. **How does all this tie together?**

   The following simple coding example shows how PTI is meant to work in practice:

```
javacall_pti_init();

void* handle=javacall_call_t9_open();

if (handle!=NULL) {

    javacall_pti_add_key(handle, '2');

    javacall_pti_add_key(handle, '2');

    javacall_pti_add_key(handle, '2');

    // go over all completions for key presses '2' '2' '2' and
    // print them

    while(javacall_pti_completion_has_next(handle)) {

    char completion[256];

    javacall_pti_completion_get_next(handle, completion, 256);

    javacall_print(completion);

}

javacall_pti_close(handle);

}
```

The above program should print several strings, for example "aba" (The first three characters of the word "abandon"), "aca" (start of the word "academy") etc.

Note – The UIDemo MIDlet, freely available with the Wireless Toolkit, can be used to test predictive-input text entry.

Quick Workaround

Predictive Text Input is not a requirement and need not be supported for Java platform compliance. Just leave the stub implementations that return `JAVACALL_FAIL`.

17

Porting the Native Image Decoder (Optional)

The purpose of this chapter is to pass all image decoding operations to the platform.

This entire chapter is optional. If you choose not to implement these APIs, only the supported image formats in MIDP will be decoded and displayed.

APIs To Be Ported

The APIs to be ported can all be found in the following file:

`javacall/interface/midp/javacall_image.h`

Detailed descriptions of all of the necessary APIs can be found in the Sun Java Wireless Client Javadoc for this file.

Background

Performance is one of the most important qualities in a mobile device. By allowing direct access to a native image decoder, with optimizations, you can improve the graphical performance of your implementation.

Implementing the Native Image Decoder improves performance and supports more image formats than those supported by the Java platform image decoder in MIDP.

Selected API Descriptions

Implement the following APIs:

- `javacall_image_decode_start()` - This function is responsible for saving the memory address of the image source buffer given.
- `javacall_image_decode_finish()` - This function is the main function of the API set. It is responsible for copying the decoded image data and alpha data to the buffers provided. If the decoded JPEG image does not have alpha blending information, then the `alphaBuf` buffer must be initialized to `0xFF`.

Answers to Common Questions

The following are common questions at this stage of the porting process:

1. **If there is no error while decoding, why is no image displayed?**

   Make sure you do not return a pointer to the decoded image. You must copy the decoded information to the buffer given in `javacall_image_decode_finish()`. Please ensure that the copy routine is as efficient as possible.

2. **The decoded image is returned properly but I cannot see anything on the display.**

   Even if you do not support alpha blending, or the image does not have any alpha blending information, your implementation must initialize the `alphaBuf` in `javacall_image_decode_finish()` to `0xFF`.

Quick Workaround

Use the built-in Java platform image decoder. The built in decoder is used if the `javacall_image_decode_start()` function returns `JAVACALL_FAIL`.

II Porting Optional JSRs

This part covers the following topics:

18

Porting JSR 75: File Connection APIs

The purpose of this chapter is to implement FileConnection functionality, such as reading/writing files and directories, getting and setting their attributes, and managing removable file systems.

APIs To Be Ported

The APIs to be ported can all be found in the following files:

- `javacall-com/interface/jsr75_pim_fc/javacall_fileconnection.h`
- `javacall/interface/common/javacall_file.h`
- `javacall/interface/common/javacall_dir.h`

Detailed descriptions of all of necessary APIs can be found in the Sun Java Wireless Client software Javadoc for these files.

Background

The File Connection API (JSR-75) provides access to native file systems. Nearly all the FileConnection functionality relies on the file system of the underlying platform. A platform may not support some functionality (e.g. certain file or directory attributes) and can still be sufficient for implementing the FileConnection APIs.

When porting your FileConnection implementation, it is up to you to decide which parts of the file system on your device will be available to Java platform applications.

An important concept related to porting FileConnection functionality is that of the "virtual root," which represents the top-level directory of a file system subtree exposed to Java platform applications. The list of roots and their correspondence to real file system paths are maintained by your implementation.

Description

The FileConnection functionality of JSR 75 relies partly on JavaCall APIs for files and directories. These are used by the core components of the Java platform stack, such as MIDP. Other (more advanced) functions are not needed for any components except JSR-75, so they are located in separate header files.

Most of the API functions described below return `javacall_result` to indicate success or failure. If any additional results are required (e.g., the directory size or a file attribute value), they are returned by means of pointer-type parameters.

Preparatory Tasks

The following questions about your platform should be answered before porting FileConnection functionality:

- Which native file system directories will Java platform applications be allowed to access?
- What attributes for files and directories will be supported by your implementation?

Define the following constants in the `javacall_platform_defs.h` file for the platform being implemented:

- `JAVACALL_MAX_FILE_NAME_LENGTH` - The maximum length of a file name on the platform.
- `JAVACALL_MAX_ILLEGAL_FILE_NAME_CHARS` - The number of characters that cannot be used in a file name (a buffer of this size is provided to `javacall_fileconnection_get_illegal_filename_chars()` function).
- `JAVACALL_MAX_ROOTS_LIST_LENGTH` - The maximum size of a list of virtual roots.
- `JAVACALL_MAX_ROOT_PATH_LENGTH` - the maximum length of a native path that corresponds to a virtual root.
- `JAVACALL_MAX_LOCALIZED_ROOTS_LIST_LENGTH` - The maximum size of a list of localized root names.

- `JAVACALL_MAX_LOCALIZED_DIR_NAME_LENGTH` - The maximum length of a localized name of a special storage directory.

The values for these constants must be sufficient to make sure that no buffer overflow occurs. However, bigger values lead to bigger memory consumption at run time.

Directory Operations

Implement the following mandatory FileConnection APIs:

- `javacall_fileconnection_create_dir()` - This function creates a new directory.
- `javacall_fileconnection_delete_dir()` - This function deletes an existing empty directory.
- `javacall_fileconnection_dir_exists()` - This function is called to check whether the specified directory exists on the file system.

Optional APIs

The following APIs are optional and are not required for a basic port.

- `javacall_fileconnection_rename_dir()` - This function renames an existing directory.
- `javacall_fileconnection_dir_content_size()` - This function counts the size (in bytes) of all files contained in a directory and, if requested, in all its subdirectories.
- `javacall_fileconnection_get_free_size()` - This function is called to determine the amount of free storage in the specified directory.
- `javacall_fileconnection_get_total_size()` - This function is called to determine the total size of the storage where the specified directory is located.

File/Directory Access API

Implement the following mandatory APIs:

- `javacall_fileconnection_is_hidden()` - This function is called to determine whether a file or directory has "hidden" attribute.
- `javacall_fileconnection_is_readable()` - This function is called to determine whether a file or directory has "readable" attribute.

- `javacall_fileconnection_is_writable()` - This function is called to determine whether a file or directory has "writable" attribute.
- `javacall_fileconnection_get_last_modified()` - This function is called to determine file or directory modification time in seconds since 00:00:00 GMT, January 1, 1970.

Certain attributes may be unsupported by the file system. In this case, some meaningful behavior of the corresponding functions should be implemented. For example, there is no "readable" attribute on Win32, i.e. all files are readable.

Therefore, `javacall_fileconnection_is_readable()` should return `JAVACALL_OK` for all existing files and report that they are readable. For non-existent files, this function should return `JAVACALL_FAIL`.

Optional APIs

The following APIs are optional and are not required for a basic port.

- `javacall_fileconnection_set_hidden()` - This function sets or resets "hidden" attribute of a file or directory.
- `javacall_fileconnection_set_readable()` - This function sets or resets "readable" attribute of a file or directory.
- `javacall_fileconnection_set_writable()` - This function sets or resets "writable" attribute of a file or directory.

`javacall_fileconnection_set_readable()` should silently return `JAVACALL_OK` for all existing files. For non-existent files, it should return `JAVACALL_FAIL`.

File System Roots and Storage Directories API

Implement the following mandatory APIs:

- `javacall_fileconnection_get_mounted_roots()` - This function is called to determine currently mounted virtual file system roots.
- `javacall_fileconnection_get_path_for_root()` - This function maps each virtual root to actual file system path. For example, `/` can be mapped to `C:\My Documents` on a particular Win32 implementation if needed. This way, Java platform applications do not have a possibility to access anything above `C:\My Documents`, which means that file system access restriction is completely controlled by the JavaCall implementation.

Some of the special storage directories may be unsupported by the platform. In this case, the corresponding functions for querying directory location and localized name must return `JAVACALL_FAIL`.

Optional API

The following APIs are optional and are not required for a basic port.

- `javacall_fileconnection_get_photos_dir()` - This function is called to determine the current location of photos storage directory.
- `javacall_fileconnection_get_videos_dir()` - This function is called to determine the current location of videos storage directory.
- `javacall_fileconnection_get_graphics_dir()` - This function is called to determine the current location of clip art graphics storage directory.
- `javacall_fileconnection_get_tones_dir()` - This function is called to determine the current location of ring tones storage directory.
- `javacall_fileconnection_get_music_dir()` - This function is called to determine the current location of music storage directory.
- `javacall_fileconnection_get_recordings_dir()` - This function is called to determine the current localized name of voice recordings storage directory.
- `javacall_fileconnection_get_private_dir()` - This function is called to determine the current localized name of directory for all Java applications' private storages.
- `javacall_fileconnection_get_localized_mounted_roots()` - This function is called to get localized names for all mounted roots. The names must go in the same order and quantity as the virtual roots in `javacall_fileconnection_get_mounted_roots()`.
- `javacall_fileconnection_get_localized_photos_dir()` - This function is called to determine the current localized name of photos storage directory.
- `javacall_fileconnection_get_localized_videos_dir()` - This function is called to determine the current localized name of videos storage directory.
- `javacall_fileconnection_get_localized_graphics_dir()` - This function is called to determine the current localized name of clip art graphics storage directory.
- `javacall_fileconnection_get_localized_tones_dir()` - This function is called to determine the current localized name of ring tones storage directory.
- `javacall_fileconnection_get_localized_music_dir()` - This function is called to determine the current localized name of music storage directory.
- `javacall_fileconnection_get_localized_recordings_dir()` - This function is called to determine the current localized name of voice recordings storage directory.
- `javacall_fileconnection_get_localized_private_dir()` - This function is called to determine the current localized name for private directories.

Root Additions/Removals Notifications

Virtual roots can be added and removed (in other words, mounted and unmounted). A typical example is inserting or removing a memory card in or from a device. Whenever this happens, your implementation must have a way to notify the Java platform about the event by calling `javanotify_fileconnection_root_changed()`.

This notification should happen as soon as possible - ideally, your implementation has a mechanism to invoke the callback function immediately upon this change in root. Subsequent calls to `javacall_fileconnection_get_mounted_roots()` and `javacall_fileconnection_get_localized_mounted_roots()` must return updated root lists.

Answers to Common Questions

The following question is commonly asked at this point in the porting process:

1. **If the platform does not support directory renaming, how should `javacall_fileconnection_rename_dir()` be implemented?**

   It is okay to remove the existing directory and create a new one.

References

For more information about PDA Optional Packages for the J2ME™ Platform, see the JSR 75 Specification at:

http://jcp.org/en/jsr/detail?id=75

19

Porting JSR 75: Personal Information Management APIs

The purpose of this chapter is to implement Personal Information Management (PIM) functionality, such as accessing lists of contacts, calendar events, and "to do" items.

APIs To Be Ported

The APIs to be ported can all be found in the following file:

- `javacall-com/interface/jsr75_pim_fc/javacall_pim.h`

Detailed descriptions of all of necessary APIs can be found in the Sun Java Wireless Client software Javadoc for this file.

Background

PIM functionality allows Java applications to access personal information on a device. This includes device-specific databases used by native applications such as address book, calendar, and tasks. PIM allows Java platform applications to interact effectively with these native applications.

Here are some general points about PIM structure:

- The Java platform can access one or more PIM lists (for example, phone address book and SIM address book).
- Each PIM list consists of items (for example, the primary address book consists of contacts).

- Each item consists of fields with unique ID numbers (for example, a contact has name, address, phone number, etc.).
- Each field can have zero or more values.
- Each PIM item can belong to zero or more categories.

Description

The PIM JavaCall header file, `javacall_pim.h`, contains many structures and enumerations for PIM data representation. Your JavaCall implementation should not rely on constant values or data format that accidentally match platform-specific constants or data structures.

Most of the API functions described below return `javacall_result` to indicate success or failure. If any additional results are required (for example, PIM item contents), they are returned by means of pointer-type parameters.

Preparatory Tasks

The following platform-dependent constants should be defined in the `javacall_platform_defs.h` file for the implementation being ported:

- `JAVACALL_PIM_MAX_ARRAY_ELEMENTS` - The maximum number of values in a single field.
- `JAVACALL_PIM_MAX_ATTRIBUTES` - The maximum number of attributes supported by a PIM list.
- `JAVACALL_PIM_MAX_FIELDS` - The maximum number of fields in a PIM item.

Values for these constants must be sufficient to make sure that no buffer overflow happens. However, bigger values lead to bigger memory consumption at run time.

Lists and Items APIs

Implement the following mandatory APIs:

- `javacall_pim_list_is_supported_type()` - This function checks if the given PIM list type is supported by the platform.
- `javacall_pim_get_lists()` - This function is used to retrieve names of all lists of a specified type.

- `javacall_pim_list_open()` - This functions opens the PIM list for access. It is called prior to doing any other operations with the list.
- `javacall_pim_list_close()` - This function closes the PIM list. It should free any platform resources consumed by opening the list.
- `javacall_pim_list_get_next_item()` - This function is used for sequential access to items within an opened list.
- `javacall_pim_list_add_item()` - This function adds a new item to an opened list.
- `javacall_pim_list_remove_item()` - This function removes an existing item from an opened list.
- `javacall_pim_list_modify_item()` - This functions alters data of an existing item and the categories it belongs to. The data is provided in vCard 2.1/3.0 or vCalendar 1.0 format.

Fields and Attributes APIs

Implement the following mandatory APIs:

- `javacall_pim_list_get_fields()` - This function is called to determine what fields are supported by a PIM list. This includes both standard and extended (OEM) fields. However, it is up to your JavaCall implementation which extended PIM fields to expose to the Java platform.
- `javacall_pim_list_get_attributes()` - This function is called to determine what attributes are supported by a PIM list.

Categories APIs

The following APIs are optional and are not required for a basic port:

- `javacall_pim_list_add_category()` - This function adds a new category to a PIM list. If the specified category already exists, the call should be considered successful.
- `javacall_pim_list_remove_category()` - This function removes an existing category from a PIM list. If there is no such category, the call should be considered successful.
- `javacall_pim_list_rename_category()` - This function renames an existing category. All items that belong to the existing category are reassigned to the new one.
- `javacall_pim_list_max_categories()` - This function is called to determine the maximum number of categories in a list.

- `javacall_pim_list_max_categories_per_item()` - This function is called to determine the maximum number of categories that a list item can belong to.
- `javacall_pim_list_get_categories()` - This function is used to get all categories defined for a PIM list.

If an empty string is not a valid category name on the platform, category functions must return `JAVACALL_FAIL` when called with this name.

Quick Workaround

The quickest way to get a working implementation of PIM functionality is to specify `JSR_75_PIM_HANDLER_IMPL=java` when building JSR 75. This build setting implements PIM functionality as a pure Java platform, which stores PIM data in plain files.

This approach does not provide any access to platform-specific personal data and does not allow interaction with native applications such as the address book. It can be viewed as PIM emulation on top of FileConnection.

References

For more information about PDA Optional Packages for the J2ME Platform, see the JSR 75 Specification at:

`http://jcp.org/en/jsr/detail?id=75`

20

Porting JSR 120: Short Message Service APIs

The purpose of this chapter is to implement Short Message Service (SMS) and Cell Broadcast Service (CBS) functionality.

APIs To Be Ported

The APIs to be ported can all be found in the following files:

- `javacall/interface/jsr120_wma/javacall_sms.h`
- `javacall/interface/jsr120_wma/javacall_cbs.h`

Detailed descriptions of all of necessary APIs can be found in the Sun Java Wireless Client software Javadoc for these files.

Background

SMS is an extremely popular way to send short text messages (called "texting") between cellular subscribers. JSR 120 provides a Java platform API for sending and receiving messages via SMS. Java applications can be registered in the "Push registry," allowing a correctly addressed incoming message to launch and run a MIDlet, even if the Java platform itself is not running.

GSM, CDMA or any other underlying protocol that supports SMS is assumed. CBS is a unidirectional data service where messages are broadcast by a base station and received by every mobile station listening to that base station. From a porting perspective the only difference between CBS and SMS is that CBS does not have a `send()` method.

Preparatory Tasks

Make sure your device is connected to the network and can send/receive SMS messages. If CBS messages are supported, to test them you should have access to a CBS center.

Selected API Descriptions

Implement the following APIs:

- `javacall_sms_send(type, addr, buf, bufLen, srcPort, destPort, handle)` - This is the main function called to send a message.
  - `type` - The message encoding type: ASCII / BINARY / UNICODE_UCS2. ASCII type is supposed to be encoded to GSM 7-bit alphabet to increase useful payload size up to 160 characters per message.
  - `addr` - The phone number in `msisdn` format as shown here:

    ```
    msisdn ::== "+" digits | digits

    digit ::== "0" | "1" | "2" | "3"| "4" | "5" | "6" | "7" | "8"| "9"

    digits ::== digit | digit digits
    ```

  - `srcPort` and `destPort` - These are optional parameters. If `destPort` is `0` then a message should come to the user inbox and `srcPort` is not processed. If `destPort` is greater than zero, the `src/destPort` pair is added to the SMS message. On the recipient side, the message is passed to an application registered on the given port (`destPort`). `srcPort` could be used by the application to send a response message.
  - `handle` - Used to notify the platform layer about the results of sending SMS. See `javanotify_sms_send_completed()`.
- `javacall_sms_add_listening_port(port_number)` - This function adds a port on which to listen for the arrival of SMS messages.
- `javacall_sms_remove_listening_port(port_number)` - This function removes a port from the list of available listening ports.

  The platform layer notifies the JavaCall layer on which port it expects to receive a message. `port` is a positive value in a range from 1-65535. Messages that come to a non-registered port should be rejected.

  The JavaCall layer has a list of registered ports; any SMS message receiving a call from the JavaCall layer should check the receiving port. If it matches to one from the list, `javanotify_incoming_sms` should be called to notify the platform layer about the received SMS.

- `javacall_sms_get_number_of_segments(type, buf, bufLen, port)` - This function returns the expected number of SMS messages to send given data. The long data can be split into several messages that are concatenated on the recipient side. The JSR 120 Specification requires the ability to send at least three messages. The return value should not exceed the expected number of characters per message.
- `javanotify_sms_send_completed(result, handle)` - This function is used to notify the platform layer that a `javacall_sms_send()` call is completed. `javacall_sms_send()` should not block the thread. Instead, the non-blocking `javacall_sms_send()` implementation should call `javanotify_sms_send_completed`.
- `javanotify_incoming_sms(type, addr, buf, bufLen, srcPort, destPort, timeStamp)` - This function is used to notify the platform layer about a received SMS. The multipart message should be concatenated before calling the platform layer.

Support for Message Segments

The maximum allowable length of an SMS message is 140 symbols per message. Longer messages will be split into segments of equal length, with up to three segments per individual message.

It is assumed that segmentation and aggregation are performed on the platform side, probably inside `javanotify_incoming_sms()` or `javacall_sms_send()`.

Answers to Common Questions

The following common questions are asked at this point in the porting process:

1. **Why don't sms_open and close APIs exist**?

   It is assumed that the platform will manage the SMS subsystem, which is running at all times. Thus, in this architecture, the Java platform just checks its availability through `javacall_sms_is_service_available()`.

2. **What's the meaning of the** `sourcePort` **parameter in** `javacall_sms_send()`**?**

   The SMS protocol supports a `sourcePort` field in its message format. The Java platform uses this field as a way to route messages to specific applications in the subscriber's device. If a Java application returns a message received from a sending server (for example, `sms://:50000`), the `sourcePort` in the return message is `50000`.

Quick Workaround

As a workaround, `javacall_sms_send()` function can block the thread. However, it should call `javanotify_sms_send_completed()` before exit.

References

For more information about SMS and Java Wireless Messaging 1.0, see the JSR 120 Specification at:

http://www.jcp.org/en/jsr/detail?id=120

21

Porting JSR 205: Multimedia Message Service API

The purpose of this chapter is to implement sending and receiving of Multimedia Message Service (MMS) messages.

APIs To Be Ported

The APIs to be ported can be found in the following file:

- `javacall-com/interface/jsr205_wma20/javacall_mms.h`

Detailed descriptions of all of necessary APIs can be found in the Sun Java Wireless Client software Javadoc for this file.

Background

The Wireless Messaging 2.0 Specification (JSR 205) provides a set of APIs for sending and receiving MMS messages that is very similar to the API of Wireless Messaging 1.0 (JSR 120). That specification describes the sending and receiving of SMS messages. The important difference between MMS and SMS is that MMS is intended for passing much larger amounts of data than just text (e.g., pictures, videos, sound files, and more).

Description

Like short integer ports for SMS, MMS has virtual character ports that are called `yapped`. The javacall layer should not accept MMS and pass it to the platform layer, if the proper `appID` is not registered.

The most important mechanism of MMS is the fetching mechanism. The MMS client (for example, a cell phone) first receives a notification about the availability of a MMS message on the MMS center. The client (e.g., the phone user) can then decide to download the message or not (receiving bytes of MMS data can cost money).

If the user decides to receive the MMS, the device calls the JavaCall `fetch()` mechanism. The `fetch()` mechanism then downloads the message and the platform layer is notified.

Preparatory Tasks

Make sure your device (phone) is connected to the network, and can send and receive MMS messages.

Selected API Descriptions

Implement the following mandatory APIs:

- `javacall_mms_send(headerLen, header, bodyLen, body, toAddr, appID, handle)` - This is the main function that sends MMS message to the network.
  - `header` - This is the standard MMS header generated by JSR 205 codes.
  - `body` - This contains data to be transferred.
  - `toAddr`, `appID` - This is the destination phone number and the destination application ID string. In most cases, the JavaCall implementation can skip these parameters because all the required data is already packaged in the standard MMS header. Besides, a "to" address header could also contain "cc" and "bcc" addresses, the senders application ID string, and a number of other required parameters.
  - `handle` - This is the `id` to be passed to `javanotify_mms_send_completed(handle)` callback when sending of the MMS is completed. Note that `javacall_mms_send()` should not block the thread. Instead, it should exit and call `javanotify_mms_send_completed()`when the process is finished.

- `javanotify_incoming_mms(fromAddress, appID, replyToAppID, bodyLen, body)` - This is the main callback that reports about receiving MMS messages.
  - `fromAddress`, `appID`, `replyToAppID` are the parameters from the MMS header. The it implementation parses the header and passes to the platform layer only those parameters that it needs.
- `javacall_mms_add_listening_appID(string)` - This function adds a listener that accepts a string as an incoming value. The string is the `appID` of the sending application.

Quick Workaround

`javanotify_incoming_mms_available()` callback and the subsequent `fetch()` call could be skipped. Data could be fully downloaded and passed to the platform layer by a single `javanotify_incoming_mms()` call.

As a workaround, the `javacall_mms_send()` function can block the thread, but it should call `javanotify_mms_send_completed()` before exit.

References

For more information about Wireless Messaging API 2.0, see the JSR 205 Specification at:

http://www.jcp.org/en/jsr/detail?id=205

22

Porting JSR 135: Mobile Media API

The purpose of this chapter is to implement Mobile Media API (MMAPI), such as audio playback, video playback, and audio and video recording.

APIs To Be Ported

The API to be ported can be found in the following files:

- `javacall/interface/jsr135_mmapi/javacall_multimedia.h`
- `javacall/interface/jsr135_mmapi/javanotify_multimedia.h`

Detailed descriptions of all necessary APIs can be found in the Sun Java Wireless Client software Javadoc for these files.

Background

The Mobile Media API (JSR 135) provides a rich set of software controls for implementing multimedia functionality on your platform. Much of this functionality, such as audio and video playback, depends on the hardware and software capabilities of your platform. MMAPI treats the multimedia capabilities of your platform as a “black box” and operates this box using the JavaCall API for JSR 135.

Note – The MMAPI Specification defines a small set of necessary APIs and a large set of optional features. This purpose of this document is to present the basic APIs. The optional features you choose to implement are up to you, based on your device and platform requirements.

Beyond implementing the mandatory APIs, you can choose which optional functionality to support, based on your native media library and the level of Java platform support required by your carrier network.

Two JavaCall functions are used to query what is supported in your JavaCall implementation:

- `javacall_media_get_configuration()`
- `javacall_media_get_player_controls()`

If some MMAPI functionality is not supported by your platform, it MUST return `JAVACALL_NOT_IMPLEMENTED`.

Overview of MMAPI

In order to effectively port your platform's multimedia capabilities using the JavaCall porting layer, you must understand the basic concepts of MMAPI:

- Player - A software entity used to playback media and get access to the playback process. You interact with the player to play, pause, resume, stop, rewind, adjust volume, and carry out other audio or video actions.
- Media Format - The player needs to know the type of content to be played and how to interact with it. Media format is determined not only by MIME type, but using the media format identifier defined in the JSR 234 Specification. (For more information on media formats, see "Media Format" on page 93 and the JSR 234 Specification, as detailed in "References" on page 109.)
- Controls - The actions a player has available to take on a media format. Not all players support all possible control functions.
- Media Capabilities - The ability of your platform to handle playback of different types of content in different ways. (For more information, see "Platform Media Capabilities" on page 97.)

Besides understanding these important Mobile Media API concepts, it is also important to understand the transactional process that occurs in the lifecycle of a player. Between the time it is created by the Java platform and resources are assigned to it, and the time it is destroyed and its resources returned to the system,

there are several states a player must pass through. During this time, decisions are made by the player program as to the media format being passed in and the best ways to handle this format.

The following sections discuss these concepts in more detail.

The Player

A new Player is created with the function `javacall_media_create()` and destroyed with the function `javacall_media_destroy()`. The function `javacall_media_create()` returns a handle (of type `javacall_handle`) to the newly-created Player. Later, this handle is passed as a parameter to any function called for by the Player, until it is destroyed.

In most cases, a new Player is created from a given URL or from media data defined by a specified MIME type.

Special Player Types

In some cases, a special Player is created by passing a special value for a URL parameter to the `javacall_media_create()` function. The following are special URLs:

- `device://midi`
- `device://tone`
- `capture://video` (possibly followed by a video encoding string)
- `capture://audio` (possibly followed by an audio encoding string)
- `capture://radio`

For further explanation and details on special player types, see the description of the class `javax.microedition.media.Manager` class in the JSR-135 Specification.

Media Format

In the JSR 135 (Mobile Media API) Java platform API, MIME type is used to identify the format of the played media.

It is not always clear which MIME type should be assigned to a certain media format. For example, some customers require that MP3 format is assigned the MIME type `audio/mpeg`. Others require that it be assigned the MIME type `audio/mp3`.

In reverse, it is not always clear which media format is contained in a certain MIME type. For example, the content type `video/mpeg` can contain MPEG-1, MPEG-2 or MPEG-4 format. All the three formats require totally different decoding algorithms.

Note – The JavaCall API definition of “media format” is similar to that used by JSR 234 (Advanced Multimedia Supplements). For more information, see the description of the `FormatControl` interface in the JSR 234 Specification.

In the multimedia JavaCall API, “format” is referred to with the typedef `javacall_media_format_type`.

Supported Mime Types

The Mobile Service Architecture (JSR 248) and Mobile Media API (JSR 135) require support for a number of MIME types. Other MIME types are optional. The Sun Java Wireless Client software supports all required MIME types, as well as some optional ones.

Player Controls

Control is a subset of the Mobile Media API that may or may not be supported by a given Player. What controls a player supports can be discovered by calling the function `javacall_media_get_player_controls()`. The return value is a bit mask where a bit is set if, and only if, the corresponding Control is supported.

Each Control corresponds to a group of Mobile Media JavaCall API functions, the calling of which may or may not be supported for a given Player. The correspondence between a specific Control functionality and the JavaCall API that calls it is shown in TABLE 22-1.

Note – In TABLE 22-1, the prefix `JAVACALL_MEDIA_CTRL_` has been omitted from the Control names for simplicity.

TABLE 22-1 Player Controls and JavaCall API

Control	Affected Functions
VOLUME	`javacall_media_get_volume`
	`javacall_media_set_volume`
	`javacall_media_is_mute`
	`javacall_media_set_mute`
RECORD	`javacall_media_close_recording`
	`javacall_media_commit_recording`
	`javacall_media_get_record_content_type`
	`javacall_media_get_record_content_type_length`
	`javacall_media_get_recorded_data`
	`javacall_media_get_recorded_data_size`
	`javacall_media_pause_recording`
	`javacall_media_recording_handled_by_native`
	`javacall_media_reset_recording`
	`javacall_media_set_recordsize_limit`
	`javacall_media_start_recording`
	`javacall_media_stop_recording`
METADATA	`javacall_media_get_metadata`
	`javacall_media_get_metadata_key`
	`javacall_media_get_metadata_key_counts`
EVENT	`javacall_media_get_event_data`
STOPTIME	None

TABLE 22-1 Player Controls and JavaCall API

Control	Affected Functions
VIDEO	`javacall_media_get_video_size`
	`javacall_media_get_video_snapshot_data`
	`javacall_media_get_video_snapshot_data_size`
	`javacall_media_set_video_color_key`
	`javacall_media_set_video_full_screen_mode`
	`javacall_media_set_video_location`
	`javacall_media_set_video_visible`
	`javacall_media_start_video_snapshot`
FRAME_POSITIONING	`javacall_media_map_frame_to_time`
	`javacall_media_map_time_to_frame`
	`javacall_media_seek_to_frame`
	`javacall_media_skip_frames`
TONE	None
MIDI	`javacall_media_get_channel_volume`
	`javacall_media_get_midibank_key_name`
	`javacall_media_get_midibank_list`
	`javacall_media_get_midibank_programe`
	`javacall_media_get_midibank_program_list`
	`javacall_media_get_midibank_program_name`
	`javacall_media_is_midibank_query_supported`
	`javacall_media_long_midi_event`
	`javacall_media_set_channel_volume`
	`javacall_media_set_program`
	`javacall_media_short_midi_event`

TABLE 22-1 Player Controls and JavaCall API

Control	Affected Functions

PITCH	`javacall_media_get_max_pitch`
	`javacall_media_get_min_pitch`
	`javacall_media_get_pitch`
	`javacall_media_set_pitch`

RATE	`javacall_media_get_max_rate`
	`javacall_media_get_min_rate`
	`javacall_media_get_rate`
	`javacall_media_set_rate`

TEMPO	`javacall_media_get_tempo` `javacall_media_set_tempo`

Note – Simple tones can be played without any Player. For more information, see "Simple Tones" on page 109.

Platform Media Capabilities

To query for your platform media capabilities, the Java platform calls `javacall_media_get_configuration()`. This can happen before any Player is created and the return value should not change over time. Define the function `javacall_media_get_configuration()` accordingly and fill in the fields of the returned "configuration" structure carefully.

The returned structure defines the following items:

- The System Properties (except for `microedition.media.version`, which is determined by the Java platform. For more information, see the JSR-135 Specification).
- Which Media Formats, over which protocols, are supported by your JavaCall implementation (these formats include the pseudo-protocols "whole stream in memory" and "streaming from memory buffers")
- Which MIME types are suitable for a given Media Format (if there is more than one MIME type, the first in the list is the default)

The Java platform determines from the above information whether player content should be downloaded completely before playing (full downloading) or whether it should be downloaded and played in parallel during the download process (streaming).

In some cases, both full downloading and streaming can be handled entirely by the Java platform, without needing to hand off to the native platform via the JavaCall porting layer. For example, the Animated GIF playback is supported entirely by the Java platform. (For more information, see the description of the class `Manager` methods, the names of which start with "`getSupported...`"in the JSR-135 Specification.)

Special Players

To indicate which special Players are supported by your implementation and which are not, fill the following fields of the Configuration accordingly:

- `supportDeviceMIDI`
- `supportDeviceTone`
- `supportCaptureRadio`
- `audioEncoding`
- `videoEncoding`
- `videoSnapshotEncoding`

For more information, see "Platform Media Capabilities" on page 97,

Player Lifecycle and Player States

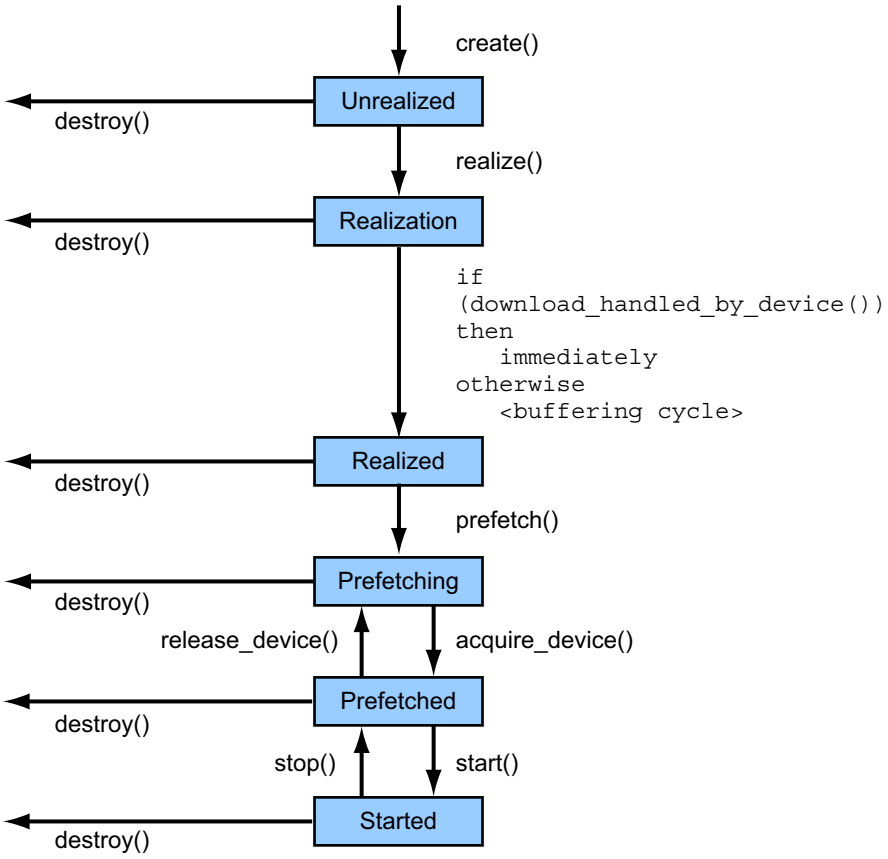
The primary purpose of a Player in the Mobile Media API is to play some kind of content, for example a song (audio) or a movie (video). In both cases, the content has a specific time duration, with an explicit beginning and an explicit ending. However, sometimes the duration cannot be known to the Player (for example, it is radio) or is undefined (for example, it is an interactive MIDI player).

When a request for content comes into the Java platform (i.e., a MIDlet calls the class `javax.microedition.media.Manager` class method `createPlayer()`), a new Player is created. After that, the Player can be closed any time the MIDlet calls the Player method `close()`. For more information, see "The Closed State" on page 108.

When the content is finished playing, the Player should generate the event END_OF_MEDIA. (For more information, see the comments in the file `javanotify_multimedia.h`.)

Between the creation of the Player and its closure, the Player can go through several states, such as unrealized, realization, and realized. The lifecycle and primary methods used for the Player are illustrated in FIGURE 22-1.

FIGURE 22-1 Player Lifecycle and Player States



The `javacall_media_create()` function creates a new Player and returns a handle to refer to it. Beside this, this function also does some initialization, but it MUST NOT try to download and examine media data in any way. However, in some cases (see "Why Are Some URLs Handled On The Native Platform Side?" on page 102), the initialization may include trying to connect to a URL in order to check its availability.

There are two special parameters passed to `javacall_media_create()`:

- application ID
- Player ID

Both parameters are integers generated by the Java platform. Later, when notifying the Java platform about any media event that occurs for the Player, you can use these values to identify the Player.

If the Player is being created from a URL, then the JavaCall porting layer may decide to reject the creation for some reason. For example, the creation may be rejected if you do not want input from a pre-defined "bad" Internet site. In this case, the function MUST return `JAVACALL_FAIL`.

The `javacall_media_destroy()` Function

The inverse of the `javacall_media_create()` function is `javacall_media_destroy()`. This function is called when the Java platform no longer needs the Player. When calling the `javacall_media_destroy()` function, be sure to de-initialize all resources initialized with `javacall_media_create()`. Also, be sure to free all resources allocated to the Player with this function. (For more information, see "The Closed State" on page 108).

The Unrealized State

The Unrealized state is the Player state immediately after it is created. In this state (after `javacall_media_create()` has just returned), the Java platform calls only one of the following functions for the Player:

- `javacall_media_get_format()` - This function queries for the Player media format. At this point, the format may still be unknown (equal to `JAVACALL_MEDIA_FORMAT_UNKNOWN)`, even if the MIME type is given by the Java platform when creating the Player.

Note – At the Realization stage (or earlier) your JavaCall porting layer implementation MUST identify the media format or reject it as not supported (`JAVACALL_MEDIA_FORMAT_UNSUPPORTED`). If the Player is refused, the function must also return `JAVACALL_MEDIA_FORMAT_UNSUPPORTED`).

- `javacall_media_download_handled_by_device()` - This function asks whether a URL passed in to the javacall_media_create() function when the Player is first initialized is to be handled by the JavaCall porting layer or by your Java platform.

Note – In some cases, a URL is not passed to `javacall_media_create()`. However, if one is, this question must be answered definitively. For more information see "Why Are Some URLs Handled On The Native Platform Side?" on page 102.

- `javacall_media_realize()` - This function initiates transition to the next Player state (Realization). Unlike the Java platform API function realize(), javacall_media_realize() MUST return immediately. The realization process may include some initialization work, and it may also include downloading and examining media data.

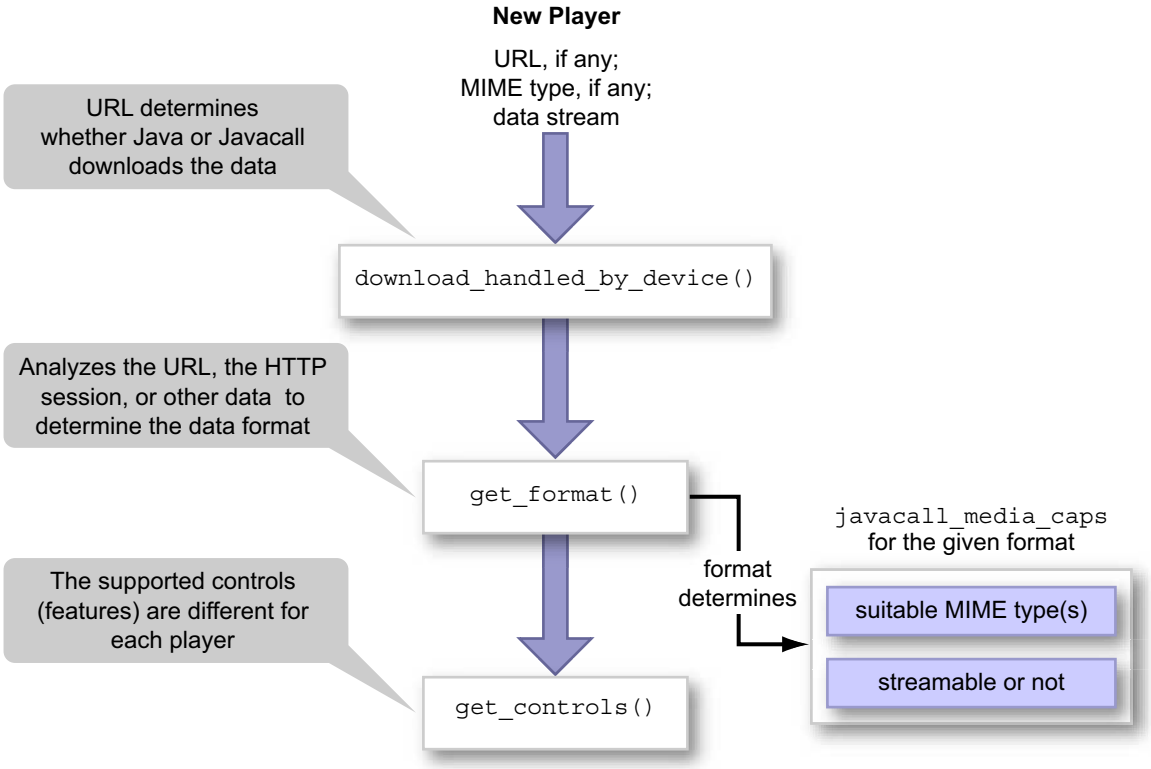
Downloading and Examining Media Data

There are two ways for a Player to download and examine media data. They are:

- Through memory buffers that are filled by the Java platform. In this case, even if a URL is available, it is connected and handled on the Java platform side.
- Directly via the URL, if available. In this case, the connection may be handled by the native platform.

The choice between the two scenarios is determined by the return value of the function `javacall_media_download_handled_by_device()`, as shown in FIGURE 22-2.

FIGURE 22-2 Downloading and Examining Media Data



Why Are Some URLs Handled On The Native Platform Side?

There are several reasons why it can be desirable for the JavaCall porting layer to connect to a URL and download or stream the data by itself:

- The limited capabilities of the Java layer. For example, in the Sun Java Wireless Client software, the Java platform does not support streaming using memory buffers. Moreover, it supports only HTTP protocol to download data. Thus, the only way to implement some functionality, for example RTSP streaming, is to connect to RTSP links on the JavaCall porting layer side.

Note – The same is true for HTTP streaming. For more information, see "Full Downloading vs. Streaming" on page 103.

- The need for special actions to be taken. For example, when connecting to some kinds of media resources, money may be charged without the Java platform being aware of the charges.
- Better performance when your Java platform handles some URLs directly. For example, transmission of data downloaded by the Java platform to a Player through memory buffers may introduce some overhead in critical resource usage, such as memory consumption and CPU load.

Full Downloading vs. Streaming

Streaming means that media is played in parallel as the content is being downloaded.

Some Players may not support this functionality. For those Players, they must have their entire data stream downloaded in full before they can begin playback.

The JavaCall porting layer MUST indicate those media types for which streaming is supported and for which types it is not. This is done by assigning the following fields to the corresponding `javacall_media_cap` structure (an array of the `javacall_media_cap` structures, indexed by media format, is a field in the structure returned by the function `javacall_media_get_configuration()`. For details, see the comments in the file `javacall_multimedia.h`.):

- `streamingProtocols`. This field reports for which protocols streaming is supported by the JavaCall porting layer for a given media format.
- `wholeProtocols`. This field reports for which protocols playback is supported, but streaming is not for a given media format.

Note – For more information, see "Platform Media Capabilities" on page 97.

In the above fields the pseudo-protocol `JAVACALL_MEDIA_MEMORY_PROTOCOL` may be used to indicate whether streaming or playback is supported from memory buffers filled by the Java platform.

Note – Streaming from memory IS NOT SUPPORTED by the Java platform in the Sun Java Wireless Client software.

The Realization State

The function `javacall_media_realize()` MUST initiate the realization process. The realization process may include some initialization work, but it may also include downloading and examining of media data.

If downloaded media data needs to be examined, the Realization state may transition media data from the Java platform to the JavaCall porting layer using the media buffering cycle. (For more information on the media buffering cycle, see "The Media Buffering Cycle" on page 104 and "The Media Buffering API" on page 105.)

One of two things must happen:

- If the result of `javacall_media_download_handled_by_device()= JAVACALL_FALSE`, the function `javacall_media_realize()` MUST return immediately and wait for the media buffering cycle to be completed. The state is considered to be changed from Realization to Realized when the media buffering cycle is finished.
- If the result of `javacall_media_download_handled_by_device()= JAVACALL_TRUE`, the function `javacall_media_realize()` must NOT return immediately. It may initialize and examine the media data using the URL that has been passed to the function `javacall_media_create()`. When the realization process is finished, the `javacall_media_realize()` function returns and the state is changed from Realization to Realized.

Examination of some portion of the media may be needed, for example, to parse the headers in order to determine the media format. However, a decision about the format may be made based on just the file extension. This is left to the discretion of your JavaCall parting layer implementation.

Beside format, other parameters MUST be recognized, for example, by parsing the headers of the media file. These are:

- Duration of the media, if available
- Height and width for video playback
- MetaData, if supported and available in the beginning of the media

The Media Buffering Cycle

During the media buffering cycle, some amount of media data ia being transferred from the Java platform to the native platform, via the JavaCall porting layer. This is done to carry out some action. The buffering cycle is finished as soon as the JavaCall porting layer signals that the action is completed and no more data is needed.

The media buffering cycle may happen in the following two Player states:

- Realization state
- Prefetching state

The media buffering cycle happens the same way in both the Realization and Prefetching states:

1. The Java platform calls the function `javacall_media_get_java_buffer_size()`. If the output parameter `first_chunk_size` returns `0`, then the cycle is finished. If it returns some other value but `0`, step 2 takes place.
2. The Java platform calls the `javacall_media_get_buffer_address()` function. Then it calls `javacall_media_do_buffering()`, using the returned address as input.
3. If the output parameter `need_more_data` returns `JAVACALL_TRUE`, Step 2 is repeated. If not, the buffering cycle is finished.

When the media buffering cycle is finished, control is passed back to the Realization or Prefetching state from which it was launched.

The Media Buffering API

The following functions are used to pass input media from the Java platform to the JavaCall porting layer during the media buffering cycle:

- `javacall_media_get_java_buffer_size()` - The Java platform calls this function to determine the optimal size for its buffers for a given Player.
- `javacall_media_get_buffer_address()` - The Java platform calls this function to get the address and size of the buffer, which it fills with a portion of input media data.
- `javacall_media_do_buffering()` - The Java platform calls this function when the current buffer is filled and can be used by the JavaCall porting layer.
- `JAVACALL_EVENT_MEDIA_NEED_MORE_MEDIA_DATA` - The JavaCall porting layer returns this event to notify the Java platform that it is ready for more input data.
- `javacall_media_clear_buffer()` - This function clears (deletes) all the internal buffers of a Player and resets the buffering system. If this function has not been called, the buffers MUST be freed by `javacall_media_destroy()`.

Note – For more information on these events, see the `javanotify_multimedia.h` file.

The Realized State

The Realized state is still not suitable to start playback. To get ready for this a Player must do a prefetch operation. However, a Player in a Realized state can be involved in other operations. (For more information on this, see the JSR-135 Specification.)

The Pre-Fetching State

The function `javacall_media_prefetch()` initiates the Prefetching state. The behavior of this function depends on the return value of the function `javacall_media_download_handled_by_device()`. There are two possible situations:

- The return value is `JAVACALL_FALSE`. In this case, the function `javacall_media_realize()` should return immediately and a buffering cycle follows.
- The return value is `JAVACALL_TRUE`. In this case, the function `javacall_media_realize()` may fill the internal buffers with media data, using the URL that has been passed to the function `javacall_media_create()`, and return only after it is done.

Note – The media buffering cycle used in the Prefetching state is the same as that used in the Realization state. For more information on the media buffering cycle and Media Buffering API, see "The Media Buffering Cycle" on page 104 and "The Media Buffering API" on page 105.

The purposes of the Prefetching state are the following:

- To fill the internal buffers with media data in order to minimize the Player start/playback latency.
- To request acquisition of scarce platform resources needed to start playback (e.g. audio device, CPU, hardware decoder, screen buffer etc.) The request can be rejected if the resources are not available or there are not enough resources to start playback because those resources are being used by other Players.

Actions in the Prefetching state include:

- Filling the Internal Buffers - It is up to your implementation to decide how much data to pre-buffer. The extreme case is the whole media.

  In order to help your implementation with the decision, the Java platform reports the size of the whole media, if available. It is done by calling the `javacall_media_set_whole_content_size()` function.

- Acquisition of Scarce Resources - This is done by the `javacall_media_acquire_device()` function, performed by the Java platform. If any of the scarce resources are not available, this function MUST return `JAVACALL_FAIL`. Otherwise the Player state becomes Prefetched.

  The inverse of the `javacall_media_acquire_device()` function is `javacall_media_release_device()`. This function MUST free scarce resources and make them available for other Players. After this, the Player becomes Prefetching, since the internal buffers are already pre-filled. (For more information, see the description of the `Player.prefetch()` method in the JSR-135 Specification.)

The PreFetched State

In the Prefetched state the user can control playback by the following functions:

- `javacall_media_start()`
- `javacall_media_stop()`
- `javacall_media_pause()`
- `javacall_media_resume()`

Their purposes correspond exactly to their names. For more details, see the file `javacall_multimedia.h`. and the JSR-135 Specification.

Reporting the Media Player Duration

A Player may or may not be aware of the total duration of the media played back. However, this information may become available only in the process of playback.

The following is used to report the duration, if available:

- `javacall_media_get_duration()` - This function MUST return `JAVACALL_NO_DATA_AVAILABLE` if the duration of playback is still unknown.
- `JAVACALL_EVENT_MEDIA_DURATION_UPDATED` - This event should be posted by the JavaCall porting layer as soon as the Player has information about the playback duration, if it was not known when `javacall_media_prefetch()` was called.

For more information on these functions, see JSR-135 Specification.

The Seek API: Rewind and Fast Forward

Some Players may support seeking within a given media time. The following functions in the JavaCall porting layer are used:

- `javacall_media_get_time()`
- `javacall_media_set_time()`

For details, see the file `javacall_multimedia.h` file.

If a Player doesn't support seek functionality, the `javacall_media_set_time()` function MUST return `JAVACALL_NOT_IMPLEMENTED`.

The Closed State

After `javacall_media_close()` has returned, the Player is not be used anymore in any way. You may free some resources using in this function, for example, memory buffers. However, alternatively, it can be done using the function `javacall_media_destroy()`.

Selected API Descriptions

The most important functions of the Mobile Media API JavaCall porting layer are considered in detail in the previous discussion, "Player Lifecycle and Player States" on page 98. However, the following supplementary functions are also important.

Media Library Initialization API

The function `javacall_media_initialize()` is called prior to any actions with the Mobile Media JavaCall API. Please place the initialization needed for the native media library in this function.

The inverse function is `javacall_media_finalize()`. It is called after all the actions with the Mobile Media JavaCall API are completed. Deinitialize in this function all things initialized in the function `javacall_media_initialize()`.

Simple Tones

It is possible to playback simple tones without any player needing to be invoked. This is done with the following functions:

- `javacall_media_play_tone()`
- `javacall_media_stop_tone()`

For more details, see the file `javacall_multimedia.h`.

Dual Tones

It is also possible to play two Simple Tones simultaneously, which is called a Dual Tone.

The `javacall_media_play_dualtone()` function is used for this.

This feature is optional and it is uncommon to require support for it. In most cases, you may just make this function return `JAVACALL_NOT_IMPLEMENTED`.

References

For more information about Mobile Media API, see the JSR 135 Specification at:

`http://jcp.org/en/jsr/detail?id=135`

For more information about the Advanced Multimedia Supplements, see Chapter 23 and the JSR 234 Specification at:

`http://jcp.org/en/jsr/detail?id=234`

For more information about the Advanced Multimedia Supplements API for J2ME Format Definitions, see:

`http://www.forum.nokia.com/main/resources/technologies/java/documentation/java_jsr.html`

23

Porting JSR 234: Advanced Multimedia API

The purpose of this chapter is to implement Advanced Multimedia Supplements API (AMMS) functionality, such as 3D audio, music effects, and image processing.

APIs To Be Ported

The API to be ported can all be found in the following file:

- `Javacall-com/interface/jsr234_amms/javacall_multimedia_advanced.h`
- `Javacall-com/interface/jsr234_amms/javanotify_multimedia_advanced.h`

Detailed descriptions of all necessary APIs can be found in the Sun Java Wireless Client software Javadoc for these files.

Background

The functionality provided by the Advanced Multimedia Supplements (JSR 234) is intended to extend and enhance the multimedia capabilities provided by the Mobile Media API (JSR 135). In MMAPI, you were given the ability to launch a single Player, then define and manage a specific set of audio and video controls for it. In AMMS, you have the ability to launch a number of Players, group them together, and then interact with them as a source of sound in 3D space (for example, by applying sound affects to them).

Although AMMS provides an extended set of controls for each individual Player, the ability to group Players also allows many new, interesting possibilities. 3D audio, music effects such as reverb and location control, and enhanced image processing for video, are all possible with AMMS. Taken together, the JSR 234 optional package extends the range and multimedia capabilities of your Java platform.

Most of AMMS functionality, such as 3D audio, depends on the hardware and software capabilities of your platform. The AMMS implementation provided by the Sun Java Wireless Client software treats the multimedia capabilities of your platform as a "black box" and operates this box using the JSR-234 JavaCall API.

Description

The Advanced Multimedia (JSR 234) JavaCall API consists of the following two parts, which differ slightly from each other:

- 3D Audio and Sound Effects (aka Music Capability)
- Image processing

Both of these features are described in detail in the subsections "Supported SoundSource3D Audio Features" on page 113 and "Image Processing" on page 120.

Some features of the Advanced Multimedia Supplements API are mandatory and others are optional. The main focus of this chapter are the mandatory APIs that must be implemented to have a working JSR 234 port.

Note – The Sun Java Wireless Client software provides support for a number of AMMS features. It is up to you to decide which of these features your hardware and software implementation will support.

Supported and Unsupported AMMS Features

The Sun Java Wireless Client software supports all JSR-234 features required by the Mobile Services Architecture (MSA 248) Specification. However, the AMMS Specification (JSR 234) provides a large set of optional features, some of which are not supported in the Sun Java Wireless Client software.

The Sun Java Wireless Client software supports the following JSR 234 features:

- Camera Control features
- Tuner Control features

For more information, see "Supported Camera Control Features" on page 116 and "Supported Tuner Control Features" on page 116.

Note – Management of the supported 3D Audio, Spectator, and Music Effects features described in the following sections are handled through the Global Manager (`javax.microedition.amms.GlobalManager`). For more information on the Global Manager, see "The Global Manager" on page 118.

Supported SoundSource3D Audio Features

SoundSource3D represents a sound source in a virtual acoustical space. The Sun Java Wireless Client software supports some 3D audio features of the Advanced Multimedia Supplements Specification. Only the following controls can be obtained from SoundSource3D:

- `DistanceAttenuationControl`
- `LocationControl`
- `ReverbSourceControl`

A Player of any media format can be added to the SoundSource3D, if it supported by your hardware and software platform. If supported, a MIDI channel can also be added to the SoundSource3D.

Note – All other features of JSR 234 3D Audio are not supported.

Supported SoundSource3D Features on Your Platform

Use the following functions to tell which SoundSource3D features are supported on your platform:

- The function `javacall_amms_local_manager_create_sound_source3d()` should return `JAVACALL_NOT_IMPLEMENTED` if SoundSource3D creation is not supported.
- Use the function `javacall_audio3d_get_supported_soundsource3d_player_types()` to indicate which media formats can be played back with SoundSource3D effects. If a player of any other format is added to a SoundSource3D with the function `javacall_audio3d_soundsource3d_add_midi_channel()`, it should return `JAVACALL_NOT_IMPLEMENTED`. For more information, see the function descriptions.
- Use the function `javacall_audio3d_soundsource3d_get_controls()` and `javacall_audio3d_soundsource3d_get_control()` to indicate which Controls (of the three in the list above) are supported for SoundSource3D. For more information, see the function descriptions.

- The function `javacall_audio3d_soundsource3d_add_midi_channel()` should return `JAVACALL_NOT_IMPLEMENTED` if addition of MIDI channels to SoundSource3D is not supported.

Supported Spectator Controls

In the Advanced Multimedia Supplements Specification, the Spectator represents the listener in a virtual acoustical space. The Sun Java Wireless Client software provides support for only the following Spectator controls:

- `LocationControl`
- `OrientationControl`

Supported Spectator Features on Your Platform

Use the following functions to tell which Spectator features are supported on your platform:

- The function `javacall_amms_local_manager_get_spectator()` should return `JAVACALL_NOT_IMPLEMENTED` if getting the Spectator is not supported.
- Use the functions `javacall_audio3d_spectator_get_control()` and `javacall_audio3d_spectator_get_controls()` to indicate which controls (from the two in the list above) are supported by your platform. For more information, see the function descriptions.

Supported Global Scope Music Effects Features

In the Advanced Multimedia Supplements Specification, "effects" provide a way for sound to be modified during audio playback. For example, adding reverb to a song gives it a wavering effect when played, but does not change the song itself. The Sun Java Wireless Client software supports the following globally-applied Music Effects features:

- `AudioVirtualizerControl`
- `ChorusControl`
- `EqualizerControl`
- `ReverbControl`
- `javax.microedition.media.VolumeControl`

Note – For globally-applied Music Effects features, the scope in which it is applied is the Java platform application.

Supported Music Effects Features on Your Platform

Use the following functions to tell which Music Effects features are supported on your platform:

- Use the functions `javacall_amms_local_manager_get_control()` and `javacall_amms_local_manager_get_controls()` to indicate which of the controls (of the five in the list above) are supported on your platform. For more information, see the function descriptions.

Supported Image Processing Features

Only the following AMMS Image Encoding and Post-Processing features are supported in the Sun Java Wireless Client software:

- A MediaProcessor can be created for raw images
- A MediaProcessor can be created for any image format if supported by your platform
- A MediaProcessor can provide:
  - ImageEffectControl
  - ImageTransformControl
  - OverlayControl
  - ImageFormatControl capable of encoding images to any format, if supported by your platform

---

Note – Other features of AMMS Image Encoding and Post-Processing *cannot* be supported.

Supported Image Processing Features on Your Platform

Use the following functions to tell which Image Processing features are supported on your platform:

- The function `javacall_image_filter_create()` should return `JAVACALL_NOT_IMPLEMENTED` if, and only if, one of the following items is true:
  - The control (from those in the list above) that corresponds to the passed `filter_type` is not supported.
  - Your platform does not support processing of the MIME type passed as `source_mime_type`.
  - Your platform does not support conversion of images to the MIME type passed as `dest_mime_type`.

- Use the function `javacall_image_filter_get_supported_dest_mime_types()` to indicate which MIME types are supported by a given Image Filter.
- Use the function `javacall_image_filter_get_supported_source_mime_types()` to indicate which MIME types are supported by a given Image Filter process.
- Use the function `javacall_image_filter_get_supported_presets()` to indicate which presets are supported by a given Image Filter.
- Use the function `javacall_image_filter_get_int_values()` to indicate which integer values are supported by a given Image Filter.
- Use the function `javacall_image_filter_get_str_values()` to indicate which string values are supported by a given Image Filter.

Supported Camera Control Features

The following optional AMMS camera controls are supported in Sun Java Wireless Client software:

- `CameraControl`
- `FlashControl`
- `FocusControl`
- `SnapshotControl`
- `ZoomControl`
- `ExposureControl`

The method `javacall_amms_camera_control_is_supported()` determines if `CameraControl` is supported for a given camera. The functionality for this is implemented by a set of `javacall_amms_camera_control_***` APIs.

Note – Only `capture://video` players are queried by the Java Wireless Client software.

Other controls (e.g., `FlashControl`, `FocusControl`, `SnapshotControl`, `ZoomControl`, and `ExposureControl`) follow similar rules as those described for `CameraControl`.

Supported Tuner Control Features

The following optional AMMS camera controls are supported in Sun Java Wireless Client software:

- TunerControl
- RDSControl

The method javacall_amms_tuner_control_is_supported() determines if TunerControl is supported for a given camera. The functionality for this is implemented by a set of javacall_amms_tuner_control_*** APIs.

Note – Only capture://radio players are queried by the Java Wireless Client software.

The RDSControl follows similar rules as those described for TunerControl.

Selected API Descriptions

According to the capabilities of your hardware platform and JavaCall API implementation, system properties can be assigned. System properties are set in the following file:

jsr234/build/cldc_application/config/properties_jsr234.xml

For more information about the properties_jsr234.xml file, see "Setting System Properties" on page 117.

Setting System Properties

The system properties file, properties_jsr234.xml, defines JSR 234 System Properties, as shown in the following example.

```
<property Key=microedition.amms.version" Value="1.1"
Scope="system"/>
.....
<property Key="audio.samplerates" Value="48000 44100 32000 22050
16000 11025 8000" Scope="system"/>
```

For each property there is a Key, which means the System Property name. The property defines the specific Value, which is the System Property value, and Scope, which should always equal to "system."

For example, for the Key=microedition.amms.version, the system is being told to reckon the JSR 234 (AMMS) version implemented on your platform as "1.1."

For a complete list of properties to assign to your JSR 234 implementation, see the JSR 234 Specification. (For the complete JSR 234 URL, see "References" on page 120.)

The Global Manager

In JSR 234, the `GlobalManager` class provides methods for the following functionalities:

- Creating SoundSource3Ds
- Creating MediaProcessors
- Getting globally-applied Controls
- Getting the Spectator

To carry out these functionalities, the following Global Manager methods are supported:

- `createSoundSource3D()`
- `createMediaProcessor(String)`
- `getControls()`
- `getSpectator()`
- `getSupportedMediaProcessorInputTypes()`
- `getSupportedSoundSource3DPlayerTypes()`

The corresponding JavaCall API methods are as follows (see the `javacall_multimedia_advanced.h` file for details).

Setting 3D Audio and Music Effects

This section describes how to map JavaCall APIs to corresponding Java platform APIs for 3D audio and music effects in your implementation of JSR 234. For example, the Java API interface `LocationControl` maps as follows.

TABLE 23-1 JavaCall API Mapping for `LocationControl`

Java API	JavaCall API
Control	javacall_amms_control_t
LocationControl	javacall_amms_location_control_t
a Control that is a LocationControl	javacall_amms_control_t, where the structure fields are as follows: -ptr is a pointer to javacall_amms_location_control_t - type equals to javacall audio_3d_eLocationControl
LocationControl.getCartesian()	javacall_audio3d_location_control_get_cartesian()
LocationControl_setCartesian()	javacall_audio3d_location_control_set_cartesian()
LocationControl.setSpherical	javacall_audio3d_location_control_set_spherical()

Here are some exceptions to this mapping structure:

- The AMMS GlobalManager maps to `javacall_amms_local_manager`. This is because, if more than one MIDlet runs in parallel, each has its own GlobalManager. The scope of each GlobalManager is not global for the native layer, because the native layer must handle the Java applications running at the same moment.
- The JSR-135 Player maps to `javacall_handle`. For more information about Mobile Media API and the JSR-135 Player, see Chapter 22.

For more information about direct mapping for this part of the AMMS, see JSR-234 Specification. (For the complete JSR 234 URL, see "References" on page 120.)

Image Processing

When it comes to image processing in your AMMS implementation, the API mapping from Java platform APIs to JavaCall porting layer APIs is not as straightforward as it is for 3D Audio and Music.

The following items make mapping image processing APIs different:

- Though `javacall_media_processor` is a mapping of the Java API MediaProcessor, it is not created from `javacall_amms_local_manager`, unlike in the Java platform API. Instead, `Javacall_media_processor` is created by the function `javacall_media_processor_create()`, which has nothing to do with the Local Manager.
- A new entity was introduced in the JSR-234 JavaCall API. It is the Image Filter (`javacall_image_filter`) as described in "The Image Filter" on page 120.

For complete information, see the `javacall_multimedia_advanced.h` file.

The Image Filter

Image Filter is an entity designed to process an image in one certain, pre-defined way. This way is determined by the Image Filter Type, the enum `javacall_amms_image_filter_type`.

A Media Processor (`javacall_media_processor`) can include one or more consecutive Image Filters, which defines exactly how an image is processed by the Media Processor.

References

For more information about Advanced Multimedia Supplements, see the JSR 234 Specification at:

`http://jcp.org/en/jsr/detail?id=234`

24

Porting JSR 211: Content Handler API

The purpose of this chapter is to port the Content Handler API (CHAPI) to your platform.

APIs To Be Ported

The APIs to be ported can all be found in the following files:

- `javacall/interface/jsr211_chapi/javacall_chapi_invoke.h`
- `javacall/interface/jsr211_chapi/javacall_chapi_registry.h`

Detailed descriptions of all of necessary APIs can be found in the Sun Java Wireless Client software Javadoc for these files.

Background

A content handler is a MIDlet designed to process a specific type of content, for example, a picture or an e-mail message. Content handlers are registered in the platform Registry. When some application in your system needs to process specific content, it calls the implementation of CHAPI and hands off one of the following things:

- A request for a specific handler, identified by handlerID
- A general request to handle a specific type of content, for example, a `.jpg` image file

The application requesting the handler passes other information to the CHAPI implementation with its request:

- The data to be processed
- The action to be taken on the data

The CHAPI implementation calls the platform Registry for the appropriate content handler, passing in a specific identifier or a general request for the type of content to be handled. The Registry returns the handler, which is passed to the Application Management System (AMS) to start up.

Once the content handler is started, the CHAPI implementation tells it what action to take on what data and the handler processes the content. When processing is finished, the handler shuts down. If results need to be returned to the calling application, they are passed back via the CHAPI implementation.

Description

The CHAPI implementation provides two kinds of JavaCall porting interfaces:

- Interfaces for porting to the platform Registry
- Interfaces for porting to the AMS

The interfaces and functions needed for both these ports are described in the following sections.

Porting to the Platform Registry

Implement the following APIs:

- `javacall_result javacall_chapi_init_registry(void)` - This function is called every time the CHAPI subsystem is initialized.
- `void javacall_chapi_finalize_registry(void)` - This function is called every time CHAPI subsystem is shut down. It is the last call to the system and frees allocated resources.
- `javacall_result javacall_chapi_register_handler()` - This function registers a new content handler in the Registry. The information passed in as parameters should be stored and returned when requested by a specific "getter" function. For more information, see `javax.microedition.content.Registry.register()`.

Enumeration Functions

The following enumeration functions should return some set of handlers that satisfy a specific condition. All these functions have a `pos_id` parameter, which provides a pointer to an integer value. Initially, the value of the parameter is a pointer to zero value.

Every `enum` function should modify this value to know what handler from the set should be returned next. If an `enum` method has the ability to extract all handlers simultaneously, it makes sense to store the result in some memory block and return its address as a value of `pos_id`.

`An enum` function should return a handler id value via the output parameters `handler_id_out` and `length`. If the length of the buffer pointed at by `handler_id_out` is less than required for storing the next handler id, the function must return an `ERROR_BUFFER_TOO_SMALL` error code. The parameter `length` must point to a required buffer size value.

Each enumeration function is called in sequence, using the output of the function as input to the next enumeration function until all functions have been called and the sequence of functions is finished.

Note – In the following `enum` functions, the comment /*OUT*/ indicates the output that is passed to the next `enum` function.

Implement the following primary enumeration functions:

- `javacall_result javacall_chapi_enum_handlers(int* pos_id, /*OUT*/ javacall_utf16* handler_id_out, int* length)` - This function begins the process of enumeration for all handlers.
- `void javacall_chapi_enum_finish(int pos_id)` - This function is called every time an enumeration of handlers is completed.

Implement the following specific enumeration functions:

- `javacall_result javacall_chapi_enum_handlers_by_suffix(javacall_const_utf16_string suffix, int* pos_id, /*OUT*/ javacall_utf16* handler_id_out, int* length)` - This function enumerates content handlers by the specified suffix.
- `javacall_result javacall_chapi_enum_handlers_by_type(javacall_const_utf16_string content_type, int* pos_id, /*OUT*/ javacall_utf16* handler_id_out, int* length)` - This function enumerates content handlers by the specified type.

- `javacall_result javacall_chapi_enum_handlers_by_action(javacall_const_utf16_string action, int* pos_id, /*OUT*/ javacall_utf16* handler_id_out, int* length)` - This function enumerates content handlers by the specified action.
- `javacall_result javacall_chapi_enum_handlers_by_suite_id( javacall_const_utf16_string suite_id, int* pos_id, /*OUT*/ javacall_utf16* handler_id_out, int* length)` - This function enumerates content handlers by the specified suite_id.
- `javacall_result javacall_chapi_enum_handlers_by_prefix(javacall_const_utf16_string id, int* pos_id, /*OUT*/ javacall_utf16* handler_id_out, int* length)` - This function enumerates content handlers by the specified prefix.

Implement the following enumeration functions:

- `javacall_result javacall_chapi_enum_suffixes(javacall_const_utf16_string content_handler_id, int* pos_id, /*OUT*/ javacall_utf16* suffix_out, int* length)` - This function enumerates all suffixes being registered with the specified content handler as the first parameter of the function (content_handler_id).
- `javacall_result javacall_chapi_enum_types(javacall_const_utf16_string content_handler_id, /*OUT*/ int* pos_id, javacall_utf16* type_out, int* length)` - This function enumerates all types being registered with the specified content handler as the first parameter of the function (content_handler_id).
- `javacall_result javacall_chapi_enum_actions(javacall_const_utf16_string content_handler_id, /*OUT*/ int* pos_id, javacall_utf16* action_out, int* length)` - This function enumerates all actions being registered with the specified content handler as the first parameter of the function (content_handler_id).
- `javacall_result javacall_chapi_enum_action_locales(javacall_const_utf16_string content_handler_id, /*OUT*/ int* pos_id, javacall_utf16* locale_out, int* length)` - This function enumerates all locales being registered with the specified content handler as the first parameter of the function (content_handler_id).
- `javacall_result javacall_chapi_enum_access_allowed_callers(javacall_const_utf16_string content_handler_id, int* pos_id, /*OUT*/`

`javacall_utf16* access_allowed_out, int* length)` - This function enumerates all allowed callers registered with the specified content handler as the first parameter of the function (`content_handler_id`).

Other Get Functions

The following functions are get functions, which return some information about a specific handler.

Implement the following functions:

- `javacall_result javacall_chapi_get_local_action_name(javacall_const_utf16_string content_handler_id, javacall_const_utf16_string action, javacall_const_utf16_string locale, /*OUT*/ javacall_utf16* local_action_out, int* length)` - This function gets a local action name.
- `javacall_result javacall_chapi_get_content_handler_friendly_appname(javacall_const_utf16_string content_handler_id, /*OUT*/ javacall_utf16* handler_friendly_appname_out, int* length)` - This function gets a friendly appname.
- `javacall_result javacall_chapi_get_handler_info(javacall_const_utf16_string content_handler_id, /*OUT*/ javacall_utf16* suite_id_out, int* suite_id_len, javacall_utf16* classname_out, int* classname_len, javacall_chapi_handler_registration_type *flag_out)` - This function gets handler info.
- `javacall_bool javacall_chapi_is_access_allowed(javacall_const_utf16_string content_handler_id, javacall_const_utf16_string caller_id)` - This function finds out if access is allowed to a certain handler.
- `javacall_bool javacall_chapi_is_action_supported(javacall_const_utf16_string content_handler_id, javacall_const_utf16_string action)` - This function finds out if a certain action is supported.
- `javacall_result javacall_chapi_unregister_handler(javacall_const_utf16_string content_handler_id)` - This function removes all information about a handler from the registry.

Porting to the AMS

Implement the following APIs:

- `javacall_result javacall_chapi_ams_launch_midlet(int suite_id, javacall_const_utf16_string class_name, /*OUT*/ javacall_bool* should_exit)` - This function starts a content handler, but should not start a second copy of the application. (This function should only be provided if a native AMS is used.)
- `javacall_result javacall_chapi_platform_invoke(int invoc_id, const javacall_utf16_string handler_id, javacall_chapi_invocation* invocation, /*OUT*/ javacall_bool* without_finish_notification, /*OUT*/ javacall_bool* should_exit)` - This function starts a native content handler.
- `void javanotify_chapi_platform_finish(int invoc_id, javacall_utf16_string url, int argsLen, javacall_utf16_string* args, int dataLen, void* data, javacall_chapi_invocation_status status)` - This function informs the CHAPI subsystem that a native handler is finished with processing some invocation.
- `void javanotify_chapi_java_invoke(const javacall_utf16_string handler_id, javacall_chapi_invocation* invocation, int invoc_id)` - This function allows an external application to create a CHAPI invocation.
- `javacall_result javacall_chapi_java_finish(int invoc_id, javacall_const_utf16_string url,int argsLen, javacall_const_utf16_string* args, int dataLen, void* data, javacall_chapi_invocation_status status, /* OUT */ javacall_bool* should_exit)` - This function signals that a content handler is finished with invocation processing.

References

For more information about Content Handler API, see the JSR 211 Specification at:

http://www.jcp.org/en/jsr/detail?id=211

25

Porting JSR-177: Security and Trust Services API

The purpose of this chapter is to implement the functionality of the Security and Trust Services API (SATSA), such as exchanging data with smart cards and other security devices.

Background

The JSR-177 (SATSA) API consists of four packages, as described in TABLE 25-1.

TABLE 25-1 Four Main Parts of the SATSA Package

Part	Description
APDU	Supports data exchange with smart cards. Used by other parts of the system to access security elements. Porting is required for any type of security element.
Java Card RMI	Provides RMI support for on-card applets.
PKI	Supports the public key infrastructure and wireless identity module.
CRYPTO	Provides interfaces for cryptographic algorithms for encoding and decoding data and generating and verifying signatures.

Note – Only one of the four SATSA packages is ported using the JavaCall porting layer. This is the Application Protocol Data Unit (APDU) package.

The SATSA Security Element

The SATSA specification introduces a Security Element (SE) as a "device" that holds the user's credentials and is used to cipher and sign messages. This device could be a smart card or a smart card reader. You can access the SE by two means:

- File access - Available for smart cards that support an on-card file system.
- Application oriented access - Available for smart cards that are compatible with Java Card™ technology or with the (U)SIM Application Toolkit specifications.

Both methods use APDU commands to communicate with the SE. The SATSA-APDU package provides data exchange with the SE.

SATSA-APDU Implementations

The SATSA-APDU package has two implementations:

- CardDevice - This implementation can support any number of devices. It provides different interfaces for card devices developed in both C and Java programming languages. Both types of card devices can be used simultaneously. This implementation is comprehensive and contains a number of classes and methods.
- Simple - This implementation can support only one device and provides only a C interface. It contains only a few classes.

APIs To Be Ported

The APIs to be ported can be found in the following file:

- `javacall/interface/jsr177_satsa/javacall_carddevice.h`

Detailed descriptions of all necessary APIs can be found in the Sun Java Wireless Client software Javadoc for this file.

The JavaCall API for SATSA consists of five functional groups:

- Initialization and finalization
- Data exchange
- Locking
- Retrieving information
- Error handling

Porting of all five functional groups is mandatory.

Initialization and Finalization API

Implement the following APIs:

- `javacall_carddevice_init()` - This function must initialize the card device. If the SATSA-APDU optional package is implemented on a device that does not have functional smart card slots this function must return `JAVACALL_NOT_IMPLEMENTED`.
- `javacall_carddevice_finalize()` - This function must finalize the card device. If no special finalization actions are required, it may do nothing.
- `javacall_carddevice_set_property()` - This function is used for setting properties. If a JavaCall implementation doesn't support a given property it must return `JAVACALL_NOT_IMPLEMENTED`.
- `javacall_carddevice_select_slot()` - This function must select a "current" slot for use. If a card device has only one slot, this method may do nothing. This method is called only when the card device is locked (see the locking group).

Data Exchange API

There are two operations with a smart card that require data exchange:

- Reset - This operation powers up the device and receives an answer-to-reset (ATR).
- Transfer data - This operation allows sending and receiving data (APDU) to and from a card.

Data exchange operations may take some time. In this case, a data exchange method must return `JAVACALL_WOULD_BLOCK`. When the operation is completed, a JavaCall implementation must call the `javanotify_carddevice_event()` callback method. Then, the corresponding `_finish()` method is called. It must either return `JAVACALL_WOULD_BLOCK` or fill given buffer with received data.

- `javacall_carddevice_reset_start()` and `javacall_carddevice_reset_finish()` reset and return an ATR.
- `javacall_carddevice_xfer_data_start()` and `javacall_carddevice_xfer_data_finish()` exchange an APDU.

Locking API

Before any data exchange operation is performed a card device must be locked. The SATSA implementation uses following scenario:

1. `javacall_carddevice_lock()` is called.
2. `javacall_carddevice_select_slot()` is called.
3. Some card operation is performed (is called).
4. `javacall_carddevice_unlock()` is called.

The `javacall_carddevice_lock()` method may return `JAVACALL_WOULD_BLOCK`. In this case, your JavaCall implementation must call `javanotify_carddevice_event()` when the device can be locked.

A JavaCall implementation must guarantee that native applications cannot access the card device when it has been locked.

- `javacall_carddevice_lock()` - This function is used to lock the card device.
- `javacall_carddevice_unlock()` - This function is used to unlock the card device.

Retrieving Information API

The following APIs must be implemented:

- `javacall_carddevice_get_slot_count()` - This function returns the number of device slots. This method is called once at initialization time.
- `javacall_carddevice_is_sat_start()` - This function may require data exchange with a smart card. If data exchange operations are used and they take time, this method must return `JAVACALL_WOULD_BLOCK`.

  When the operation is completed, a JavaCall implementation must call the `javanotify_carddevice_event()` callback method. Then, the corresponding `javacall_carddevice_is_sat_finished()` method is called. It must return `JAVACALL_WOULD_BLOCK` or return a boolean value that represents if the slot is a SAT slot.
- `javacall_carddevice_card_movement_events()`- This function must report which insertion/withdrawals are performed with a card. This method is called before and after any card operation.

Error Handling API

If an error is believed to have occurred, the SATSA implementation sends a text message into the error handling system. In that case, an error state must be set. The SATSA implementation must then retrieve all messages (if allowed by memory limitations) concatenated to one string and separated by new line characters. After receiving this message string, the error state must be cleared.

- `javacall_carddevice_clear_error()` - This function is used to clear the error state.
- `javacall_carddevice_set_error()` - This function receives a message and sets the error state.
- `javacall_carddevice_get_error()` - This function returns the concatenated message string and clears the error state.

Additional SATSA Packages

The other three SATSA packages (RMI, PKI, and SATSA-CRYPTO) are not ported using the JavaCall porting layer. Discussing their use is outside the scope of this document.

References

For more information about Security and Trust Services API for J2ME™ see the JSR 177 Specification at:

http://jcp.org/en/jsr/detail?id=177

26

Porting JSR 179: LandmarkStore API

The purpose of this chapter is to implement the LandmarkStore database.

APIs To Be Ported

The APIs to be ported can all be found in the following file:

- `javacall-com/interface/jsr179_location/javacall_landmarkstore.h`

Detailed descriptions of all of the necessary APIs can be found in the Sun Java Wireless Client Javadoc for this file.

Background

The SR-179 API provides access to a native database of known landmarks stored in the device. It is up to you to decide what parts of the LandmarkStore database is implemented via the JavaCall porting layer, and where, and in which format, the LandmarkStore database is stored on the platform.

Description

Currently there are two implementations of the LandmarkStore:

- The `java_global` implementation provides all functionality necessary for the JSR-179 LandmarkStore. This implementation is platform independent and does not have any interfaces to the platform.
- The `platform_global` implementation has special dedicated native interfaces used as a porting layer. The `platform_global` implementation is used if the LandmarkStore database is shared with the native platform.

Preparatory Tasks

Implementation of the LandmarkStore is chosen at build time by assigning appropriate values to the `JSR_179_STORE_IMPL` option in the `build/subsystem.gmk` file. Also, the following internal properties should be defined in the *<jsr179_root>*`/src/share/config/common/properties_jsr179.xml` file, where *<jsr179_root>* is the location where you have installed the JSR 179 source:

- `com.sun.j2me.location.CreateLandmarkStoreSupported` - Set the value of this to `true` if creation of the new LandmarkStore is supported.
- `com.sun.j2me.location.DeleteLandmarkStoreSupported` - Set the value of this to `true` if removal of the LandmarkStores is supported.
- `com.sun.j2me.location.CreateCategorySupported` - Set the value of this to `true` if creating a landmark Category is supported.
- `com.sun.j2me.location.DeleteCategorySupported` - Set the value of this to `true` if removal of a landmark Category is supported.

Selected API Descriptions

In case of a `platform_global` implementation of LandmarkStore, the following JavaCall functions should be implemented.

Note – In all functions below, a NULL landmarkStoreName means the default LandmarkStore.

Implement the following mandatory APIs.

- `javacall_landmarkstore_landmark_add_to_landmarkstore()`- This function should a add new landmark to an existing LandmarkStore. It returns a unique landmark identifier.
- `javacall_landmarkstore_landmark_add_to_category()` - This function adds an existing landmark to an existing landmark category.
- `javacall_landmarkstore_landmark_update()` - This function updates an existing landmark in the LandmarkStore.
- `javacall_landmarkstore_landmark_delete_from_landmarkstore()` - This function deletes a landmark from the LandmarkStore.
- `javacall_landmarkstore_landmark_delete_from_category()` - This function deletes a landmark from a category only. It does not remove a landmark from the LandmarkStore.
- `javacall_landmarkstore_list_open()` - This function provides a list of existing LandmarkStores and returns a handle to the list. Only one LandmarkStore name is returned by each call. If no more LandmarkStores are available, this call should return `NULL`.
- `javacall_landmarkstore_list_next()` - This function uses the handle provided by `javacall_landmarkstore_list_open()` to provide LandmarkStore names. The LandmarkStore name returned by `javacall_landmarkstore_list_next()` should be valid until the next `javacall_landmarkstore_list_next()` call, or until `javacall_landmarkstore_list_close()` is called.
- `javacall_landmarkstore_list_close()` - This function deallocates all allocated memory.
- `javacall_landmarkstore_landmarklist_open()` - This function returns all landmarks belonging to a provided LandmarkStore. If a `categoryName` is not `NULL`, only landmarks belonging to a provided category inside a LandmarkStore are returned.
- `javacall_landmarkstore_landmarklist_next()` - This function assumes that the returned landmark memory block is valid until the next function call. It returns `NULL` if no more landmarks are in the list.
- `javacall_landmarkstore_landmarklist_close()` - This function deallocates all allocated memory.
- `javacall_landmarkstore_categorylist_open()` - This function returns a list of category names in a specified landmark store.
- `javacall_landmarkstore_categorylist_next()` - This function assumes that the returned landmark memory block is valid until the next function call. It returns `NULL` if no more categories are in the list.
- `javacall_landmarkstore_categorylist_close()` - This function deallocates all allocated resources.

Optional API

The following optional APIs are not needed to create a basic, functional working port:

- `javacall_landmarkstore_create()` - This function must be implemented if `CreateLandmarkStoreSupported` is set to `true` in the `src/share/config/common/properties_jsr179.xml` file.
- `javacall_landmarkstore_delete()` - this function should be implemented if `DeleteLandmarkStoreSupported` is set to `true` in the `src/share/config/common/properties_jsr179.xml` file.
- `javacall_landmarkstore_category_add()` - This function should be implemented if `CreateCategorySupported` is set to `true` in the `src/share/config/common/properties_jsr179.xml` file.
- `javacall_landmarkstore_category_delete()` - This function should be implemented if `DeleteCategorySupported` is set to `true` in the `src/share/config/common/properties_jsr179.xml` file.

References

For more information about the Landmark Store API for J2ME Platform, see the JSR 179 Specification at:

http://jcp.org/en/jsr/detail?id=179

27

Porting JSR 179: Location API

The purpose of this chapter is to implement the Location API.

APIs To Be Ported

The APIs to be ported can all be found in the following file:

- `javacall-com/interface/jsr179_location/javacall_location.h`

Detailed descriptions of all of necessary APIs can be found in the Sun Java Wireless Client software Javadoc for this file.

Background

JSR-179 provides access to location services such as obtaining information about the present geographic location and orientation of a device. Location information is platform dependent, as access to specific platform libraries, or even hardware, may be required.

Description

The native platform can provide several methods to access location services, such as satellites (Standalone GPS), CellID, or Assisted GPS. Every method should be implemented as a separate `LocationProvider`.

Porting of the `LocationProvider` to the native implementation hides interactions with the location device and is intended to be a porting layer for an efficient platform-dependent implementation of the `LocationProvider`.

Additionally, there are two implementations of `atan2` math function for Location API:

- The `java_global` platform independent implementation. This implementation is platform independent and does not require porting to new platforms.
- The `platform_global` implementation has a native interface that can be used as a porting layer. This implementation is available if an optimized implementation of `atan2` is present on the platform

Preparatory Tasks

The following internal properties should be defined in the *<jsr179_root>*`/src/share/config/common/properties_jsr179.xml` file, where *<jsr179_root>* is the location where you have installed the JSR 179 source:

- `com.sun.j2me.location.ProximitySupported` - Set the value of this to `true` if proximity monitoring is supported.

Selected API Descriptions

Implement the following mandatory APIs:

- `javacall_location_property_get()` - This function returns a list of Location Providers supported by the platform. If more then one Location Provider is supported, they should be separated by commas.

  If this function is called with the `JAVACALL_LOCATION_ORIENTATION_LIST` parameter, it should return the Orientation Provider name. If returned as `NULL`, the orientation functionality is not supported.

- `javacall_location_provider_getinfo()` - This function returns information about the Location Provider.

- `javacall_location_provider_open()` - This function opens a Location Provider. In the most cases, initialization of a Location Provider requires long term operations. In this case, `javacall_location_provider_open()` should return `JAVACALL_WOULD_BLOCK` and continue initialization in the background.

  After initialization is completed, the JavaCall implementation should call the `javanotify_location_event(JAVACALL_EVENT_LOCATION_OPEN_COMPLETED)` function, an with appropriate status code. If `javacall_location_provider_open()` calls for an already opened Location Provider, the function should return it quickly, without reinitialization.

- `javacall_location_provider_close()` - This function closes a Location Provider. Your platform should calculate the number of opened Location Provider instances and, if all instances are closed, release the Location Provider.

- `javacall_location_provider_state()` -This function returns the state of a Location Provider:

  - `JAVACALL_LOCATION_AVAILABLE`
  - `JAVACALL_LOCATION_OUT_OF_SERVICE`
  - `JAVACALL_LOCATION_TEMPORARILY_UNAVAILABLE`

- `javacall_location_update_set()` - This function initiates obtaining new coordinates and should return `JAVACALL_WOULD_BLOCK`.

  As soon as new coordinates are obtained, your platform should call `javanotify_location_event()` with `JAVACALL_EVENT_LOCATION_UPDATE_ONCE`. If timeout expires before obtaining the new coordinates, `javanotify_location_event()` should be called with `JAVACALL_LOCATION_RESULT_TIMEOUT`.

- `javacall_location_update_cancel()` - This function should cancel a `javacall_location_update_set()` request. In this case, `javanotify_location_event()` should be called with `JAVACALL_LOCATION_RESULT_CANCELED`.

- `javacall_location_get()` - This function returns location information, obtained via the `javacall_location_update_set()` function.

Optional APIs

The following APIs are optional and do not need to be implemented to achieve a basic working port:

- `javacall_location_get_extrainfo()` - This function returns additional information about an obtained location, if `javacall_location_update_set()` has completed successfully, or return a reason for the failure.

Your platform should indicate ExtraInfo in the `extraInfoSize` field of the last received Location Information. There are three predefined mime-types for `ExtraInfo`:

- `application/X-jsr179-location-nmea`
- `application/X-jsr179-location-lif`
- `text/plain`

If your platform supports another mime-type for Extra Info, it should be indicated in the `outMimeTypeBuffer` field.

- `javacall_location_get_addressinfo()` - This function returns Address Info related to the latest received Location Information. Your platform should indicate `AddressInfo` presence in the `addressInfoFieldNumber` field of the received Location Information.
- `javacall_location_orientation_update()` - This function initiates obtaining of new orientation information and should return `JAVACALL_WOULD_BLOCK`. As soon as orientation is obtained, your platform should call `javanotify_location_event()` with `JAVACALL_EVENT_LOCATION_ORIENTATION_COMPLETED`.
- `javacall_location_orientation_get()` - This function returns orientation information, obtained via `javacall_location_orientation_update()`.
- `javacall_location_atan2()` - This function calculates atan from two parameters. Your platform should use this function to override the default atan2 implementation by your platform optimized version.

References

For more information about the Location API for J2ME Platform, see the JSR 179 Specification at:

`http://jcp.org/en/jsr/detail?id=179`

28

Porting JSR 82: Bluetooth API

The purpose of this chapter is to implement Bluetooth functionality.

APIs To Be Ported

The APIs to be ported can all be found in the following files:

- `javacall-com/interface/jsr82_bt/javacall_bt.h`
- `javacall-com/interface/jsr82_bt/javanotify_bt.h`

Detailed descriptions of all of the necessary APIs can be found in the Sun Java Wireless Client Javadoc for these files.

Background

The Bluetooth Specification provides a standard set of Java programming language APIs that enables low-power, handheld devices such as cell phones, pagers, PDAs, and other small devices to share functionality over a peer-to-peer wireless connection. Bluetooth wireless technology allows for heterogeneous connections between different kinds of devices (for example, between a cell phone and a PDA).

Description

Bluetooth technology supports three categories of basic functionality, including the following:

- Discovery - The process of seeking and finding other Bluetooth-enabled objects, which can include the following:
  - Devices, such as other cell phones, headsets, or PDAs.
  - Services, such as applications available from a Bluetooth server.
  - Registering services, for example, making Bluetooth-enabled applications available for discovery by other Bluetooth-enabled devices or applications.
- Communication - The process of establishing connection with another Bluetooth-enabled device and using that connection to exchange data between the devices.
- Device management - The process of managing and controlling connectivity between Bluetooth-enabled devices.

The Bluetooth Stack

This API covers the three parts that make up the Bluetooth stack:

- Bluetooth Control Center (BCC) and the BluetoothStack interface - This part provides a means for device management and security, including:
  - Device inquiry
  - Retrieval of a remote device name
  - Authentication
  - Updating the Service Discovery Database (SDDB) service records
- Service Discovery and Registration - This part provides a means to discover services that are available on a service discovery server. A service discovery server is responsible for advertising services to client devices, including:
  - Creating service records to advertise services
  - Updating service records if services change
  - Removing service records when services are no longer available
- Logical Link and Control Application Protocol (L2CAP) and Bluetooth Serial Port Profile (BTSPP) - This part provides a means to discover if data packets are in the input queue and return them for reading, using different packaging formats.

Preparatory Tasks

The major work related to the JSR82 porting lies in implementation of the JavaCall API for the target platform.

The Javacall API contains two parts:

- JavaCall functions called from the JSR level, which should be ported on the target platform. All JavaCall API Bluetooth functions and variable types have the prefix `javacall_bt_`.
- `javanotify` functions provided by the JSR, which are called by the target platform to communicate with the JSR level. All `javanotify` functions have the prefix `javanotify_bt_`.

Constants related to the Bluetooth implementation are located in the file `properties_jsr82.xml`. Configuration parameters related to the Bluetooth implementation are in the file `subsystem.gmk`.

Depending on your implementation of JSR 82, it is possible to configure the following parameters:

- The JSR 82 reference implementation contains an emulator that allows you to work without having access to a real Bluetooth device. To use this option, you must set the following variable in the `subsystem.gmk` file: `USE_JSR_82_EMULATOR=true`.
- The JSR 82 reference implementation provides the Service Discovery Data Base (SDDB) that supports the Service Discovery protocol. However, many bluetooth devices have their own implementation of SDDB.

  You can choose to use the SDDB implementation that comes with the JSR 82 reference implementation or the one that comes with the native platform. If you choose the JSR 82 SDDB reference implementation, you must set the following variable in the `subsystem.gmk` file: `USE_NATIVE_SDDB=false`. If you choose the native platform implementation, you must set `USE_NATIVE_SDDB=true`.

JavaCall API Bluetooth Variable Types and Values

Most JavaCall API Bluetooth functions return the result value. The type of it is the same as for other JSRs and it has name `javacall_result`. Most often the values are as follows:

- `JAVACALL_OK` - This means an action has finished correctly.
- `JAVACALL_FAIL` - This indicates an error occurred during execution.
- `JAVACALL_WOULD_BLOCK` - this means that the function starts an asynchronous operation and a function is called when the operation has finished.

Selected API Descriptions

Implement the following APIs:

- Notification functions (prefix `javanotify_bt_`) - Notification functions shall be called by the target platform when an appropriated event occurs. Data structures and functions headers are defined in the `javanotify_bt.h` file.
- Bluetooth Control Center (prefix `javacall_bt_bcc_`) - Please refer to the `javacall_bt.h` file for more information. Everything you need to know is well-explained in that file.
- Bluetooth Stack (prefix `javacall_bt_stack_`)
  - `javacall_bt_stack_start_inquiry()` - This function is asynchronous. A device is discovered via the `javanotify_bt_device_discovered()` call. Notification about a completed inquiry is sent via the `javanotify_bt_inquiry_complete()` call.
  - `javacall_bt_stack_ask_friendly_name()` - This function shall be asynchronous. The notification about discovered device - via the `javanotify_bt_remote_name_complete()` call.
  - `javacall_bt_stack_authenticate()` - This function is asynchronous. The notification about a discovered device - via the `javanotify_bt_authentication_complete()` call.
  - `javacall_bt_stack_encrypt()` - This function is asynchronous. The notification about discovered device - via the `javanotify_bt_encryption_change()` call.
- Service Discovery Database (prefix `javacall_bt_sddb_`)
  - `javacall_bt_sddb_update_psm()`- This function is required for the push implementation only.
  - `javacall_bt_sddb_get_records()` - This function returns not `javacall_result` but number of entries available/saved to the array, or `0` if an error occurs.
- Service Discovery Protocol (prefix `javacall_bt_sdp_`) - These functions provide an access to the native Service Discovery Protocol (SDP) implementation that is provided by the target platform. Implementation of these functions is required in case the `USE_NATIVE_SDDB` variable is set in the build system to the true.
  - `javacall_bt_sdp_request()` - This function provides transfer of the required parameters to the native SDP implementation to form a service discovery request. The notification about discovered service - via the `javanotify_bt_service_result()` call, notification about service search completed via the `javanotify_bt_service_complete()` call.
  - `javacall_bt_sdp_get_service()`- This function reads a service record from the service search result. The returned data shall be in the bluetooth PDU format.

- L2CAP protocol (prefix `javacall_bt_l2cap_`) - Please refer to the `javacall_bt.h` file for more information. Everything you need to know is well-explained in that file.
- RFCOMM protocol (prefix `javacall_bt_rfcomm_`) - Please refer to the `javacall_bt.h` file for more information. Everything you need to know is well-explained in that file.

References

For more information about the Bluetooth API 1.1, see the JSR 82 Specification at:

http://jcp.org/en/jsr/detail?id=82

29

Porting JSR 256: Mobile Sensor API

The purpose of this chapter is to implement Mobile Sensor functionality.

APIs To Be Ported

The APIs to be ported can all be found in the following file:

- `javacall-com/interface/jsr256_sensor/javacall_sensor.h`

Detailed descriptions of all of necessary APIs can be found in the Sun Java Wireless Client software Javadoc for this file.

Background

A sensor is a device designed to read specific kinds of information. For example, a sensor may read temperature, pressure, longitude, or many other kinds of data.

A sensor may consist of one or several channels. Each channel can read data of a determined type. To implement sensor functionality into your system, you need to take the following two general steps:

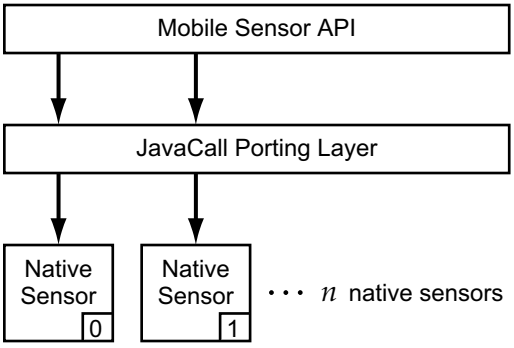
- Set the properties of each sensor and their channels
- Implement the simplest sensor functions (initialization, reading data, etc.)

Sensor functionality can be implemented on the native platform or strictly at the Java platform layer. This chapter covers primarily the native platform layer, which requires porting specific functions using the JavaCall porting layer interfaces.

Note – Information about creating sensors strictly on the Java platform layer is included in this chapter, but it is not the primary focus of this discussion.

The following figure provides an overview of the Mobile Sensor API and its relationship to the native sensors on your device. The hardware layer of your device may have one to *n* number of sensors, each designed to monitor or respond to specific conditions or data. Your Java platform JSR 256 implementation uses the JavaCall porting interfaces to call your native sensors and receive data from them.

FIGURE 29-1 Overview of Mobile Sensor API with Native Sensors



The JavaCall functions you need to implement the relationship between your native sensors and your Java platform are described in this chapter.

Description

You must set sensor properties for them to be recognized by your system. Properties are required for all sensors regardless of their interface (that is, regardless of whether they are implemented at the Java platform or native platform layer).

Mobile sensor properties are defined in the following file:

`$HOME/src/share/config/common/properties_jsr256_default.xml`

Many properties can be defined for each sensor. Properties are set first for the individual sensor, followed by property settings for each channel on the sensor. Additional sensors are defined in the same way. For example, if your device has three native sensors, you define the properties for each sensor and channels in the `properties_jsr_256_default.xml` file for sensors `0`, `1`, and `2`.

Sensor Startup Process

When your device starts up, the properties defined in the `properties_jsr_256_default.xml` file are read into memory for each sensor. The sensor class creates an instance of `sensor0` and calls the JavaCall porting layer to retrieve a `sensorID` from the first native sensor. When `sensorID` is returned, the `sensor0` instance is ready to receive data from the first native sensor.

A second instance of the sensor class is created, the JavaCall porting layer is called to retrieve a `sensorID` from the second sensor, and when the `sensorID` is returned, the `sensor1` instance is ready to receive data from the second native sensor. This process is followed for all sensors defined in the `properties_jsr_256_default.xml`.

Preparatory Tasks

When integrating your sensors with the JavaCall porting layer, the classes for sensor and channel instances are predefined. These predefined classes are:

- `NativeExampleSensor`
- `NativeExampleChannel`

There is no need to change the implementation of any listed classes.

`NativeExampleSensor` Class

Each JavaCall sensor has an unique number called `sensorType`. The `DeviceFactory.generateSensor(...)` method should contain the following code, as shown below. `numberSensor` is passed in as an argument by the `generateSensor` method.

```
        switch (numberSensor) {

        ..............
```

In the above example, several possible return cases exist. In the return function, `SENSOR_CURRENT_BEARER` is equal to the unique number of an individual sensor.

- Case 1

```
return new
NativeExampleSensor(numberSensor,channelCount,SENSOR_CURRENT_
BEARER);
```

- Case 2

```
return new
NativeExampleSensor(numberSensor,channelCount,SENSOR_BATTERY_
LEVEL);
```

- Case 3

```
return new
NativeExampleSensor(numberSensor,channelCount,SENSOR_BATTERY_
CHARGE);
```

- Case 4

```
return TestingSensor.getInstance(numberSensor,channelCount);.
..................
       default:
           }
           return null;
     }
```

The third argument of the `NativeExampleSensor` class constructor is `sensorType` declared above. The contents of the `NativeExampleSensor` class do not need to change.

`NativeExampleChannel` Class

The `DeviceFactory.generateChannel(...)` method should contain the following code, as shown below.

```
public static ChannelDevice generateChannel(int numberSensor,
int numberChannel) {

       switch (numberSensor) {

       ..............
```

In the above example, several possible return cases exist. In the return function, `SENSOR_CURRENT_BEARER` is equal to the unique number of an individual sensor.

- Case 1

```
// Bearer Sensor - javacall

      switch (numberChannel) {

         case 0: // First channel

return new NativeExampleChannel(numberSensor, numberChannel,
SENSOR_CURRENT_BEARER);

                   }

      break;
```

- Case 2

```
 switch (numberChannel) {

      case 0: // First channel

return new NativeExampleChannel(numberSensor, numberChannel,
SENSOR_BATTERY_LEVEL);

}

      break;

   ..............
```

The third argument of `NativeExampleChannel` class constructor is `sensorType` declared above. The contents of `NativeExampleChannel` class does not need to be changed.

Selected API Descriptions

Implement the following JavaCall APIs:

- `javacall_result javacall_sensor_is_available(javacall_sensor_type sensor)` - This function is called by the system and checks if a sensor is available. Returns `JAVACALL_OK` when a sensor is available else. If not available, it returns `JAVACALL_FAIL`.
- `javacall_result javacall_sensor_open(javacall_sensor_type sensor, void** pContext)` - This function is called by the system and opens a sensor. It returns:
  - JAVACALL_OK if successful.

- JAVACALL_FAIL if there's an error.
- JAVACALL_WOULD_BLOCK when the system needs notification to continue.

- javacall_result javacall_sensor_close(javacall_sensor_type sensor, void** pContext) - This function is called by the system and closes a sensor. It returns:
  - JAVACALL_OK if successful.
  - JAVACALL_FAIL if there's an error.
  - JAVACALL_WOULD_BLOCK when the system needs notification to continue.
- void javanotify_sensor_connection_completed(javacall_sensor_type sensor, javacall_bool isOpen, int errCode) - This function is called by the porting implementation and provides notification to the system that the sensor open and close process has been completed. (Notification is provided only after JAVACALL_WOULD_BLOCK has been returned.)
- javacall_result javacall_sensor_get_channel_data(javacall_sensor_type sensor, int channel, javacall_sensor_channel_data* data, int* dataCount, void** pContext) - This function is called by the system and gets channel data from a native sensor. The return values are the same as the javacall_sensor_open function.
- javacall_result javacall_sensor_start_measuring_data(javacall_sensor_type sensor) - This function is called by the system to start measuring sensor data. After this function has been called, the native platform should call javanotify_sensor_channel_data_available() to notify the JVM when there is data available.
- javacall_result javacall_sensor_stop_measuring_data(javacall_sensor_type sensor) - This function is called by the system to stop measuring sensor data. After this function is called, the VM will no longer be notified when there is sensor data available.
- javacall_result javanotify_sensor_channel_data_available(javacall_sensor_type sensor, int channel, int errCode) - This function is called by the porting implementation to notify the system that data is available or that an error has occurred.
- javacall_result javacall_sensor_start_monitor_availability(javacall_sensor_type sensor) - This function is called by the system to start monitoring a specified sensor's availability. If the monitored sensor's state is changed, the native platform should notify JVM by calling javanotify_sensor_availability().

- `javacall_result javacall_sensor_stop_monitor_availability(javacall_sensor_type sensor)` - This function is called by the system to stop monitoring a specified sensor's availability. The JVM is no longer be notified if there are state changes for the specified sensor.
- `void javanotify_sensor_availability(javacall_sensor_type sensor, javacall_bool isAvailable)` - This functions is called by the porting implementation to notify the Java platform that the monitored sensor's available state is changed.

Implementing Non-Native Sensors

A sensor can be implemented on your device without requiring porting of JavaCall methods to talk to the native hardware layer. The implementation of software sensors can be done entirely in the Java programming language.

Each sensor and channel is represented in the system by an instance of a special class. This class contains the abstract methods for initializing, reading data, etc. Implementation of these methods for any sensor is a way to plug a sensor and its channels into the Java platform.

Note – This section provides only a brief overview of the Java programming language interfaces used to implement software sensors. For more detail, see the Sun Java Wireless Client software Javadoc.

TABLE 29-1 describes the abstract methods used to implement the `com.sun.javame.sensor.SensorDevice` class.

When the Java platform receives sensor properties as input, it calls the static method `DeviceFactory.generateSensor(int numberSensor, int channelCount)`. This method returns the instance of an appropriate sensor class (or returns `null` when the input parameters are wrong).

`numberSensor` contains the base-0 number of the sensor. `channelCount` contains the number of channels for the sensor.

TABLE 29-1 The `SensorDevice` Class

Name, Return Type, and Arguments	Call When	Description
public abstract boolean initSensor();	Connector.open("sensor:...")	Implementation needs contain sensor initialization (e.g. signal exchange with sensor) or be empty when initialization is not need.
public abstract boolean finishSensor();	SensorConnection.close()	Same as above
public abstract boolean isAvailable;	SensorInfo.isAvailable()	Returns true when sensor is available else returns false.
public abstract void startMonitoringAvailability (Availability Listener listener);	SensorManager.add SensorListener(...)	Sends signal to sensor for activate availability monitoring or empty when sensor doesn't support availability monitoring.
public abstract void stopMonitoringAvailability();	SensorManager.removeSensorListener(...)	Sends signal to sensor for terminate availability monitoring or empty when sensor doesn't support availability monitoring.

TABLE 29-2 describes the abstract methods used to implement the `com.sun.javame.sensor.ChannelDevice` class.

When the Java platform receives channel properties as input, it calls the static method `DeviceFactory.generateChannel(int numberSensor, int numberChannel)`. This method returns the instance of appropriate channel class (or returns `null` when the input parameters are wrong).

`numberSensor` contains the base-0 number of the sensor. `channelCount` contains the base-0 number of the channel for this sensor.

TABLE 29-2 The `ChannelDevice` Class

Name, Return Type, and Arguments	Call When	Description
public abstract boolean initChannel();	SensorDevice.initSensor() for each channel	Implementation needs contain channel initialization (e.g. signal exchange with sensor) or be empty when initialization is not need.
protected abstract int measureData(int numberSensor, int numberChannel, int sensorType);	System tries to read data from channel (SensorConnection.getData(...), SensorConnection.setDataListener(...), etc.)	Porting engineer implements here the data reading process from channel. This method saves data in the internal storage and returns the code of reading: /** Error code: read data is OK. */ public static final int DATA_READ_OK = 0; /** Error code: channel is busy. */ public static final int CHANNEL_BUSY = 1; /** Error code: buffer is overflow. */ public static final int BUFFER_OVERFLOW = 2; /** Error code: sensor becomes unavailable. */ public static final int SENSOR_UNAVAILABLE = 3; or any other error codes.

TABLE 29-2 The `ChannelDevice` Class

Name, Return Type, and Arguments	Call When	Description
protected abstract Object[] getData(int numberSensor, int numberChannel);	Please see previous string	This method returns the array of last data measured by measureData() method. When the data type of channel is integer, the elements of returned array are instances of Integer, when data type is double, elements are Double.
protected abstract float getUncertainty(int numberSensor, int numberChannel);	Please see previous string	This method returns the uncertainty of last read data.
protected abstract boolean getValidity(int numberSensor, int numberChannel);	Please see previous string	This method returns the uncertainty of last read data.

References

For more information about the Mobile Sensor API, see the JSR 256 Specification at:

`http://jcp.org/en/jsr/detail?id=256`

30

Milestone Two: Testing Your Completed Port

At this point in the porting process, there is sufficient Sun Java Wireless Client software functionality in your device to warrant an initial run of the Java Technology Compatibility Kit (TCK) test suites.

Running the test suites for the first time is an informative process and provides much new information about the current state of your ported components.

Note – Instructions for running the Java Technology Compatibility Kit test suites are outside the scope of this document. For more information on Java ME TCKs, see `http://java.sun.com/javame/meframework/index.jsp`.

Glossary

API Application Programming Interface. A set of classes used by programmers to write applications, which provide standard methods and interfaces and eliminate the need for programmers to reinvent commonly used code.

AMS Application Management Service. The system functionality that completes tasks such as installing applications, updating applications, and switching foregrounds.

Application list The screen that lists all of the installed applications. The user gets to this screen by pressing the `Apps` soft key on the home screen. The application list uses text color to show which applications are running. It also provides a system menu that enables the user to perform application management tasks on the highlighted application.

Background An application state in which the application does not receive events from its input stream and its displayable is not rendered to the screen.

CDC Connected Device Configuration. A Java ME platform configuration for devices, it requires a minimum of 2 megabytes of memory and a network connection that is always on.

CLDC Connected Limited Device Configuration. A Java ME platform configuration for devices with less than 512 kilobytes of RAM and an intermittent (limited) network connection, it uses a stripped-down Java virtual machine called the KVM, as well as several minimalist Java platform APIs for application services.

Configuration Defines the minimum Java runtime environment (for example, the combination of a Java virtual machine and a core set of Java platform APIs) for a family of Java ME platform devices.

Foreground The application state in which the application is rendered to the device display and the input stream is passed to it.

Foreground switching Changing which application is in the foreground by shifting the focus from one application to another.

GCF Generic Connection Framework. A part of CLDC, it improves network connectivity for wireless devices.

Home screen The main screen of the application manager. This is the screen the user sees after they exit an application.

HTTP HyperText Transfer Protocol. The most commonly used Internet protocol, based on TCP/IP, which is used to fetch documents and other hypertext objects from remote hosts.

HTTPS Secure HyperText Transfer Protocol. A protocol for transferring encrypted hypertext data using Secure Socket Layer (SSL) technology.

JAD file Java Application Descriptor file. A file provided in a MIDlet suite that contains attributes used by application management software (AMS) to manage the MIDlet's life cycle, as well as other application-specific attributes used by the MIDlet suite itself.

JAR file Java Archive file. A platform-independent file format that aggregates many files into one. Multiple applications written in the Java programming language and their required components (`.class` files, images, sounds, and other resource files) can be bundled in a JAR file and provided as part of a MIDlet suite.

Java Community Process™ (JCP™) program Java Community Process program. An open organization of international developers and licensees who develop and revise Java platform specifications, reference implementations, and technology compatibility kits using a formal submission and approval process.

Java ME platform Java Platform, Micro Edition. A group of specifications and technologies that pertain to running the Java platform on small devices, such as cell phones, pagers, PDAs, and set-top boxes. More specifically, the Java ME platform consists of a configuration (such as CLDC or CDC) and a profile (such as MIDP or Personal Basis Profile) tailored to a specific class of device.

Java Specification Request (JSR) A proposal for developing new Java platform technology, which is reviewed, developed, and finalized into a formal specification by the JCP program.

Java Virtual Machine A software "execution engine" that safely and compatibly executes the byte codes in Java class files on a microprocessor.

KVM A Java virtual machine designed to run in small devices, such as cell phones and pagers. The CLDC configuration is designed to run in a KVM.

LCD Liquid Crystal Display. A common kind of screen display often used in small devices.

LCDUI Liquid Crystal Display User Interface. A user interface toolkit for interacting with LCD screens in small devices. More generally, a shorthand way of referring to the MIDP user interface APIs.

MIDlet An application written for MIDP.

MIDlet suite A way of packaging one or more midlets for easy distribution and use. Each MIDlet suite contains a Java application descriptor file (`.jad`), which lists the class names and files names for each MIDlet, and a Java Archive file (`.jar`), which contains the class files and resource files for each MIDlet.

MIDP Mobile Information Device Profile. A specification for a Java ME platform profile, running on top of a CLDC configuration, which provides APIs for application life cycle, user interface, networking, and persistent storage in small devices.

Obfuscation A technique used to complicate code by making it harder to understand when it is de-compiled. Obfuscation makes it harder to reverse-engineer applications and therefore, steal them.

Optional Package A set of Java ME platform APIs that provides additional functionality by extending the runtime capabilities of an existing configuration and profile.

PNG Portable Network Graphics. An image format commonly used with MIDP that can be compressed, transmitted, and stored without losing image quality.

Preemption Taking a resource, such as the foreground, from another application.

Preverification Due to limited memory and processing power on small devices, the process of verifying Java technology classes is split into two parts. The first part is preverification and done off-device using the preverify tool. The second part, which is verification, is done on the device at runtime.

Profile A set of APIs added to a configuration to support specific uses of a mobile device. Along with its underlying configuration, a profile defines a complete and self-contained application environment.

Provisioning A mechanism for providing services, data, or both to a mobile device over a network.

Push Registry The list of inbound connections, across which entities can push data, maintained by the Java Wireless Client software. Each item in the list contains the URL (protocol, host, and port) for the connection, the entity permitted to push data through the connection, and the application that receives the connection.

RMI Remote Method Invocation. A feature of Java SE technology that enables Java technology objects running in one virtual machine to seamlessly invoke objects running in another virtual machine.

RMS Record Management System. A simple record-oriented database that enables a MIDlet to persistently store information and retrieve it later. MIDlets can also use the RMS to share data.

SMS Short Message Service. A protocol allowing transmission of short text-based messages over a wireless network.

SOAP Simple Object Access Protocol. An XML-based protocol that allows objects of any type to communicate in a distributed environment, it is most commonly used to develop web services.

SSL Secure Sockets Layer. A protocol for transmitting data over the Internet using encryption and authentication, including the use of digital certificates and both public and private keys.

Sun Java Device Test Suite A set of Java programming language tests developed specifically for the wireless marketplace, providing targeted, standardized testing for CLDC and MIDP on small and handheld devices.

SVM Single Virtual Machine. A mode of the Java Wireless Client software, it can run only one MIDlet at a time.

task At the platform level, each separate application that runs within a single Java virtual machine is called a task. The API used to instantiate each task is a stripped-down version of the Isolate API defined in JSR 121. See the *CLDC HotSpot Implementation Architecture Guide* for more information.

TCP/IP Transmission Control Protocol/Internet Protocol. A fundamental Internet protocol that provides for reliable delivery of streams of data from one host to another.

WAE Wireless Application Environment. It provides an application framework for small devices, by leveraging other technologies such as WAP, WTP, and WSP.

WAP Wireless Application Protocol. A protocol for transmitting data between a server and a client (such as a cell phone) over a wireless network. WAP in the wireless world is analogous to HTTP in the World Wide Web.

WMA Wireless Messaging API. A set of classes for sending and receiving Short Message Service messages.

(x) button The button the user presses to end a task. On a real device this is the End key. On Windows it is the End key and sometimes the power key on the phone skin.

Index