



Optional APIs Porting Guide Volume II: Non-Sun APIs

Sun Java™ Wireless Client Software 2.0
Java Platform, Micro Edition

Sun Microsystems, Inc.
www.sun.com

May 2007

Copyright © 2007 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A. All rights reserved.

Sun Microsystems, Inc. has intellectual property rights relating to technology embodied in the product that is described in this document. In particular, and without limitation, these intellectual property rights may include one or more of the U.S. patents listed at <http://www.sun.com/patents> and one or more additional patents or pending patent applications in the U.S. and in other countries.

THIS PRODUCT CONTAINS CONFIDENTIAL INFORMATION AND TRADE SECRETS OF SUN MICROSYSTEMS, INC. USE, DISCLOSURE OR REPRODUCTION IS PROHIBITED WITHOUT THE PRIOR EXPRESS WRITTEN PERMISSION OF SUN MICROSYSTEMS, INC.

U.S. Government Rights - Commercial software. Government users are subject to the Sun Microsystems, Inc. standard license agreement and applicable provisions of the FAR and its supplements.

This distribution may include materials developed by third parties.

Sun, Sun Microsystems, the Sun logo, Java, JavaCall, Javadoc, J2ME, J2SE, J2EE, Java Community Process, JCP, HotSpot, and the Java Coffee Cup logo are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd.

Intel is a trademark or registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

OpenGL is a registered trademark of Silicon Graphics, Inc.

The Adobe logo is a registered trademark of Adobe Systems, Incorporated.

Products covered by and information contained in this service manual are controlled by U.S. Export Control laws and may be subject to the export or import laws in other countries. Nuclear, missile, chemical biological weapons or nuclear maritime end uses or end users, whether direct or indirect, are strictly prohibited. Export or reexport to countries subject to U.S. embargo or to entities identified on U.S. export exclusion lists, including, but not limited to, the denied persons and specially designated nationals lists is strictly prohibited.

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

Copyright © 2007 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, Etats-Unis. Tous droits réservés.

Sun Microsystems, Inc. détient les droits de propriété intellectuelle relatifs à la technologie incorporée dans le produit qui est décrit dans ce document. En particulier, et ce sans limitation, ces droits de propriété intellectuelle peuvent inclure un ou plus des brevets américains listés à l'adresse <http://www.sun.com/patents> et un ou les brevets supplémentaires ou les applications de brevet en attente aux Etats - Unis et dans les autres pays.

CE PRODUIT CONTIENT DES INFORMATIONS CONFIDENTIELLES ET DES SECRETS COMMERCIAUX DE SUN MICROSYSTEMS, INC. SON UTILISATION, SA DIVULGATION ET SA REPRODUCTION SONT INTERDITES SANS LAUTORISATION EXPRESSE, ECRITE ET PREALABLE DE SUN MICROSYSTEMS, INC.

Droits du gouvernement des États-Unis – logiciel commercial. Les droits de l'utilisateur du gouvernement des États-Unis sont soumis aux termes de la licence standard Sun Microsystems et aux conditions appliquées de la FAR et de ces compléments.

Cette distribution peut comprendre des composants développés par des tierces parties.

Sun, Sun Microsystems, le logo Sun, Java, JavaCall, Javadoc, J2ME, J2SE, J2EE, Java Community Process, JCP, HotSpot, et le logo Java Coffee Cup sont des marques de fabrique ou des marques déposées de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays.

UNIX est une marque déposée aux Etats-Unis et dans d'autres pays et licenciée exclusivement par X/Open Company, Ltd.

Intel est une marque déposée de Intel Corporation ou de sa filiale aux Etats-Unis et dans d'autres pays.

OpenGL est une marque déposée de Silicon Graphics, Inc.

Le logo Adobe. est une marque déposée de Adobe Systems, Incorporated.

Les produits qui font l'objet de ce manuel d'entretien et les informations qu'il contient sont regis par la législation américaine en matière de contrôle des exportations et peuvent être soumis au droit d'autres pays dans le domaine des exportations et importations. Les utilisations finales, ou utilisateurs finaux, pour des armes nucléaires, des missiles, des armes biologiques et chimiques ou du nucléaire maritime, directement ou indirectement, sont strictement interdites. Les exportations ou réexportations vers des pays sous embargo des États-Unis, ou vers des entités figurant sur les listes d'exclusion d'exportation américaines, y compris, mais de manière non exclusive, la liste de personnes qui font objet d'un ordre de ne pas participer, d'une façon directe ou indirecte, aux exportations des produits ou des services qui sont régi par la législation américaine en matière de contrôle des exportations et la liste de ressortissants spécifiquement désignés, sont rigoureusement interdites.

LA DOCUMENTATION EST FOURNIE "EN L'ETAT" ET TOUTES AUTRES CONDITIONS, DECLARATIONS ET GARANTIES EXPRESSES OU TACITES SONT FORMELLEMENT EXCLUES, DANS LA MESURE AUTORISEE PAR LA LOI APPLICABLE, Y COMPRIS NOTAMMENT TOUTE GARANTIE IMPLICITE RELATIVE A LA QUALITE MARCHANDE, A L'APTITUDE A UNE UTILISATION PARTICULIERE OU A L'ABSENCE DE CONTREFAÇON.

Contents

Preface vii

1. Overview 1

2. Porting the Personal Information Management and File Connection Optional Package 3

Package Design 3

Porting Notes 5

Building JSR 75 6

Testing JSR 75 7

JavaCall Porting Layer 7

FileConnection 7

Personal Information Management 9

JavaCall API Implementation Notes 9

Memory Allocation 10

Special Storage Locations 10

Localized Storage Directory Names 10

Platform-specific Constants 11

Callback functions 11

JavaCall API Porting Steps 11

File Connection Implementation 12

PIM Implementation 12

3. Porting the Bluetooth Optional Package 13

Bluetooth Overview 13

Bluetooth Profiles 14

Bluetooth Functionality 14

Bluetooth Control Center 15

BluetoothStack Interface and the Bluetooth Control Center 15

Porting the BluetoothStack Interface 16

Assumptions About the Implementation 16

Implementation Requirements 17

▼ Porting the Bluetooth Stack 17

Porting the BCC Interface 20

L2CAP Protocol and BTSP Profile 23

BlueZ Library 23

Making Connections in L2CAP and BTSP 24

▼ Porting L2CAP and BTSP 26

Porting the OBEX Interface 27

JavaCall Porting Layer 28

JavaCall API Bluetooth Variable Types and Values 28

Definition of JavaCall API Bluetooth Variable Types, Values and Functions
28

JavaCall API Bluetooth Function Groups 28

Memory Allocation 29

Porting steps 29

4. Location API 31

Introduction 31

Implementation Description 31

LandmarkStore Implementation 32

LocationProvider implementation	33
Atan2 Implementation	33
Location API Code Structure	33
Porting layer functionality	34
Native Porting Layer	35
Accessing Landmarks	35
Getting the Current Location	36
Acquisition of Terminal Orientation	37
Implementation Notes	38
Asynchronous Operation	38
Buffer Allocation	38
Mandatory LandmarkStore Functions	39
Optional LandmarkStore Functions	39
Mandatory LocationProvider Functions	39
Optional LocationProvider Functions	40
Optional Atan2 Function	40
Callback functions	40
Porting JSR 179	41
Implementing LocationProvider	41
Implementing LandmarkStore	41
Implementing atan2	41
JavaCall Porting Layer	42
Accessing Landmarks	42
Getting the Current Location	43
Acquisition of Terminal Orientation	44
Implementation Notes	44
Asynchronous Operation	45
Buffer Allocation	45

Mandatory LandmarkStore Functions	45
Optional LandmarkStore Functions	46
Mandatory LocationProvider Functions	46
Optional LocationProvider functions	46
Optional Atan2 Function	47
Callback Functions	47
Porting JSR 179	47
Implementing LocationProvider	47
Implementing LandmarkStore	48
Implementing atan2	48
5. Integrating the Scalable 2D Vector Graphics Optional Package	49
6. Mobile Internationalization API	51
Porting java_global	52
Porting with the JavaCall API	54
Glossary	57
Index	61

Preface

This guide describes how to port Sun Java™ Wireless Client Software 2.0 software to your mobile device.

Note – Sun Microsystems has simplified the naming schemes for the various Java platforms. Java Platform, Enterprise Edition (Java EE) was formerly Java 2 Platform, Enterprise Edition (J2EE™). Java Platform, Standard Edition (Java SE) was formerly Java 2 Platform, Standard Edition (J2SE™). Java Platform, Micro Edition (Java ME) was formerly Java 2 Platform, Micro Edition (J2ME™).

References in this guide to specific documents, specifications, and products that were released when the old naming scheme was in use retain their original names. General references in this guide to Java platforms use the new, simplified naming scheme.

Before You Read This Book

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

- *CLDC 1.1 Specification*
- *MIDP 2.0 Specification*
- *JTWI Specification*
- *CLDC HotSpot™ Implementation Porting Guide*
- *Skin Author's Guide to Adaptive User Interface Technology*
- *Scalable 2D Vector Graphics API for J2ME Specification*
- *Wireless Messaging API 2.0 Specification*

- *Mobile 3D Graphics API for J2ME Specification*
- *Mobile Media API Specification*
- *Java APIs for Bluetooth Specification*
- *PDA Optional Packages for the J2ME Platform Specification*
- *J2ME Web Services Specification*
- *Security and Trust Services API for J2ME Specification*

How This Book Is Organized

This book contains the following chapters and appendices:

[Chapter 1](#) describes the contents of the rest of the book.

[Chapter 2](#) describes how to port the FileConnection and PIM APIs that are defined by JSR 75.

[Chapter 3](#) covers porting the Bluetooth and OBEX APIs from JSR 82.

[Chapter 4](#) contains information about porting the JSR 179 Location API.

[Chapter 5](#) describes porting the JSR 226 SVG API.

[Chapter 6](#) covers porting the JSR 238 Mobile Internationalization API.

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

TABLE P-1

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

Shell Prompts

TABLE P-2

Shell	Prompt
C shell	%

Related Documentation

The following documentation is included with this release:

TABLE P-3

Application	Title
All	<i>Release Notes</i>
Building Java Wireless Client software	<i>Build Guide</i>
Porting Java Wireless Client software	This book
Viewing reference documentation created by the Javadoc™ tool	<i>Java API Reference</i>
Viewing reference documentation created by the Doxygen tool	<i>Native API Reference</i>

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 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>.

Overview

This book describes how to port optional APIs that are part of the Java Wireless Client software. This book covers the optional APIs that do not belong wholly or partly to Sun Microsystems, Inc. The *Optional APIs Porting Guide, Volume I* covers optional APIs that do belong to Sun.

The exact porting process varies from API to API. In general, each optional API consists of three types of source code:

- Java platform code
- Platform-independent C code
- Platform-dependent C code

The platform-independent C code is bound to some lower layer like PCSL and works as long as the lower layer is ported correctly.

The part you must port to use an optional API on your device is the platform-dependent code.

Some optional APIs are missing one or more types of source code. APIs that don't have any platform-dependent C code can be used "for free" without any porting.

If you are using the JavaCall™ API, then no porting in the optional APIs themselves is necessary. All the platform-dependent C code is bound to the JavaCall API, which essentially makes it platform-independent C code. Your entire porting effort is concentrated on the JavaCall layer.

The following table lists the non-Sun optional APIs and their porting requirements:

Optional API	Porting Requirements
JSR 75	See Chapter 2 .
JSR 82	See Chapter 3 .
JSR 179	See Chapter 4 .

Optional API	Porting Requirements
JSR 180	The implementation is almost entirely Java platform code. Platform-independent C code hooks up the push registry. No porting is necessary.
JSR 205	This API (WMA 2.0) is already covered in Volume I of the <i>Optional APIs Porting Guide</i> in the chapter about WMA.
JSR 226	See Chapter 5 .
JSR 229	The implementation consists of Java platform code and platform-independent C code. No porting is necessary.
JSR 238	See Chapter 6 .

Porting the Personal Information Management and File Connection Optional Package

This chapter discusses the design and porting of the PDA Optional Packages for the J2ME Platform (JSR 75). The JSR 75 specification defines the Personal Information Management (PIM) and File Connection (FC) optional packages. The customized version included with Java Wireless Client software is based on the Java Wireless Toolkit implementation. See <http://www.jcp.org/en/jsr/detail?id=75> for more information about JSR 75.

Package Design

The file `midpFileConnection.c` contains a native implementation of the `DefaultFileHandler` class. It uses PCSL (see the *Sun Java Wireless Client Porting Guide*) to abstract from most system-dependent features. It calls the functions defined in `midpNativeMounts.h` to get native mount points. Use this API to initiate the native file system monitor, request currently mounted roots, and resolve human-friendly file root names into native file path nomenclature.

The following performance optimizations are done in `midpFileConnection.c`:

- The `connect()` method converts the `java.lang.String` `path` parameter to a `MidpString` and stores it in memory allocated by the `midpMalloc` function. A pointer to this string is passed to the Java platform and saved there as a long integer object. It is used for all subsequent file operations with `FileConnection` objects. The memory allocated for this string is freed by the `FileConnection.finalize` code.

- During start up, `DefaultFileHandler.initialize()` is called. The native implementation (in `jsr75/src/kvem/native/share/midpFileConnection.c`) obtains `jfieldID` values for all `DefaultFileHandler` fields accessed from native code. It determines whether they were retrieved, if they were, it saves them into native static variables.

The list of supported devices is defined by the `validDisks` array in `midpMountedRoots.cpp`. For this release, the `CFCard`, `SDCard`, and `MMCCard` devices are supported for the Texas Instruments OMAP730 development board, and the `Floppy` device is supported for the PC. The following special file roots are available for all targets: `PIMdb`, `Storage`, and `Private`. These roots are stored in dedicated subdirectories of the `appdb` directory. These directories are automatically created by the build system. The first one is used to store the PIM database, the second one can be useful if no removable devices (such as flash cards) are connected, and the third is a container for MIDlet private working directories.

The `jsr75_suite_remove_cleanup()` function is called when MIDlet suites are removed. It deletes the suites's private working directory and all its content.

The `ROMizer` is used to improve performance and security.

Porting Notes

- The following table lists `java.lang.System` properties that must be supported by the `FileConnection` implementation.

TABLE 2-1 Properties That Must be Supported by a JSR 75 implementation

Property	Description
<code>microedition.io.file.FileConnection.version</code>	Specifies the version of the <code>FileConnection</code> API. Example: 1.0.
<code>microedition.pim.version</code>	Specifies the version of the PIM API. Example: 1.0.
<code>file.separator</code>	Returns a string that represents the file separator characters for the underlying platform. Because <code>pcsl_file_getpathseparator</code> gets the value of <code>file.separator</code> , if PCSL is ported to the target system, the system properties work correctly with no additional work.

- The native file system monitor class and the file roots proxy class must be implemented for the target platform to port the JSR 75 File Connection optional package to it. These classes are required to get the mount list and to be notified of mount and unmount events. The default implementation uses polling to detect the addition and removal of file systems. If the target platform provides a mechanism that notifies you about the addition and removal of disks, it is better to customize the existing `MidpFileSystemMonitor` and `MidpMountedRoots` classes to use this mechanism.
- The `getMountedRoots` function must return the list of mounted roots as a string where `\n` is a separator. According to the JSR 75 specification based on the Uniform Resource Locators specification (RFC 1738), the file root must *not* start with the (`/`) character, but must finish with the (`/`) character. The (`/`) character is the cross-platform separator used in a URL to separate directories in a path.
- The current implementation posts the `FC_DISKS_CHANGED_EVENT` event in the Java Wireless Client software event queue after the mount list is changed. This event contains no information about which file root was mounted or unmounted. If an operating system provides details about which file root is mounted or unmounted, this information can be passed to Java Wireless Client software through the `FC_DISKS_CHANGED_EVENT` parameters.

Building JSR 75

To build the JSR 75 module, the following variables must be defined:

- Set `JSR_75_DIR` to point to the JSR 75 workspace source directory.
- Set `USE_JSR_75` to `true` in the Java Wireless Client software build system.

The makefiles contain default settings that assume that `USE_JSR_75` is set to `false`. The typical mechanism to override the default settings is to use command line settings when invoking the `make` command as shown in the following file fragment.

```
make \  
USE_JSR_75=true \  
JSR_75_DIR=local-path-to-JSR75 workspace \  
...
```

Testing JSR 75

Use the following steps to configure Java Device Test Suite software version 1.4 to test the JSR 75 module:

1. **Run the `run_admin` configuration script.**
2. **Open an existing profile or create a new profile that includes PDA OP (JSR 75) TestSuite (File->Open Profile or File->New Profile).**
3. **Select Configure Test Suites->PDA OP (JSR 75) TestSuite.**
4. **On the FileConnection tab specify the root directory name (for example, `Storage/`).**
5. **Select Security Permissions:**
 - a. **On the `read_user_data_access` tab, enable the `javax.microedition.io.Connector.file.read` checkbox.**
 - b. **On the `write_user_data_access` tab, enable the `javax.microedition.io.Connector.file.write` checkbox.**

See the *Sun Java Device Test Suite Administration Guide* for more information.

JavaCall Porting Layer

FileConnection JavaCall API functions play the same role as PCSL functions do for non-JavaCall API platforms.

Personal Information Management JavaCall API functions are much like `jsr75_pim_xxx()` native porting layer functions.

FileConnection

FileConnection JavaCall API functions are defined in `javacall_fileconnection.h` header file.

All API functions listed below are mandatory. In addition to these, JSR 75 FC requires some of core MIDP JavaCall API functions (defined in `javacall_file.h` and `javacall_dir.h`) to be implemented.

The porting layer includes:

- Base functions
 - javacall_fileconnection_init()
 - javacall_fileconnection_finalize()
 - javacall_fileconnection_get_illegal_file_name_chars()
- File access functions
 - javacall_fileconnection_is_hidden()
 - javacall_fileconnection_set_hidden()
 - javacall_fileconnection_is_readable()
 - javacall_fileconnection_set_readable()
 - javacall_fileconnection_is_writable()
 - javacall_fileconnection_set_writable()
 - javacall_fileconnection_get_last_modified()
- Directory functions
 - javacall_fileconnection_create_dir()
 - javacall_fileconnection_delete_dir()
 - javacall_fileconnection_rename_dir()
 - javacall_fileconnection_dir_exists()
 - javacall_fileconnection_is_directory()
 - javacall_fileconnection_dir_content_size()
 - javacall_fileconnection_get_free_size()
 - javacall_fileconnection_get_total_size()
- Information on file system roots and special storage locations
 - javacall_fileconnection_get_mounted_roots()
 - javacall_fileconnection_get_path_for_root()
 - javacall_fileconnection_get_localized_mounted_roots()
 - javacall_fileconnection_get_photos_dir()
 - javacall_fileconnection_get_videos_dir()
 - javacall_fileconnection_get_graphics_dir()
 - javacall_fileconnection_get_tones_dir()
 - javacall_fileconnection_get_music_dir()
 - javacall_fileconnection_get_recordings_dir()
 - javacall_fileconnection_get_localized_private_dir()
 - javacall_fileconnection_get_localized_photos_dir()
 - javacall_fileconnection_get_localized_videos_dir()
 - javacall_fileconnection_get_localized_graphics_dir()
 - javacall_fileconnection_get_localized_tones_dir()
 - javacall_fileconnection_get_localized_music_dir()
 - javacall_fileconnection_get_localized_recordings_dir()
 - javacall_fileconnection_get_localized_private_dir()

Personal Information Management

Personal Information Management JavaCall API functions are defined in `javacall_pim.h`.

Implementations can provide access to PIM databases that are not resident on the device but at well known locations, located on SIM cards attached to the device, or to PIM databases created and managed by the Java platform VM itself.

All API functions listed below are mandatory, unless the Java platform implementation of PIM (based on high-level File Connection interface) is used.

The porting layer includes:

- Accessing PIM list and manipulating its items
 - `javacall_pim_list_is_supported_type()`
 - `javacall_pim_get_lists()`
 - `javacall_pim_list_open()`
 - `javacall_pim_list_close()`
 - `javacall_pim_list_get_next_item()`
 - `javacall_pim_list_add_item()`
 - `javacall_pim_list_remove_item()`
 - `javacall_pim_list_modify_item()`
- Accessing fields and attributes
 - `javacall_pim_list_get_fields()`
 - `javacall_pim_list_get_attributes()`
- Managing categories
 - `javacall_pim_list_add_category()`
 - `javacall_pim_list_remove_category()`
 - `javacall_pim_list_rename_category()`
 - `javacall_pim_list_max_categories()`
 - `javacall_pim_list_max_categories_per_item()`
 - `javacall_pim_list_get_categories()`

JavaCall API Implementation Notes

This section describes general principles of the FC and PIM JavaCall API.

Memory Allocation

Buffers for all output parameters are allocated by the caller, unless specified otherwise.

In case of strings and arrays, there is a parameter for the size of the buffer. If output data does not fit into the buffer of the given size, the JavaCall API function must return `JAVACALL_FAIL` value.

Special Storage Locations

All `javacall_fileconnection_get_xxx_dir()` functions return paths to the directories where files of certain types are located. Each directory corresponds to one system property accessible on Java level:

- `fileconn.dir.photos` (photos and other images)
- `fileconn.dir.videos` (video clips)
- `fileconn.dir.graphics` (clip art graphics)
- `fileconn.dir.tones` (ring tones)
- `fileconn.dir.music` (music files)
- `fileconn.dir.recordings` (voice recordings)
- `fileconn.dir.private` (private storage for MIDlet suites)

If any of the directories are absent (not supported) on the platform, the respective JavaCall API function should return `JAVACALL_FAIL` value. This will mean null value of the corresponding system property in the Java platform.

Localized Storage Directory Names

For any of the special storage locations that is supported, a localized name must also be provided. In other words, if a `javacall_fileconnection_get_xxx_dir()` function returns `JAVACALL_OK`, then the corresponding `javacall_fileconnection_get_localized_xxx_dir()` must also return `JAVACALL_OK` and provide a meaningful localized string describing the storage directory. These functions provide values for the following system properties:

- `fileconn.dir.photos.name`
- `fileconn.dir.videos.name`
- `fileconn.dir.graphics.name`
- `fileconn.dir.tones.name`
- `fileconn.dir.music.name`
- `fileconn.dir.recordings.name`
- `fileconn.dir.private.name`

Platform-specific Constants

JSR 75 FC uses several platform-dependent constants that should be properly defined when implementing JavaCall API. These constants are used by the JSR code for buffers pre-allocation when calling JavaCall API functions, so their values must be sufficient as buffer sizes to hold file names, root lists, and other native file system data:

- `JAVACALL_MAX_FILE_NAME_LENGTH` – maximum length of a file name allowed the file system, located in the `javacall_file.h`
- `JAVACALL_MAX_ILLEGAL_FILE_NAME_CHARS` – maximum number of characters that cannot be used in file names, located in the `javacall_file.h`
- `JAVACALL_MAX_ROOTS_LIST_LENGTH` – maximum length of a string with a list of all roots provided by the device, located in the `javacall_dir.h`
- `JAVACALL_MAX_ROOT_PATH_LENGTH` – maximum length of native (file system-specific) path to any of the roots, located in the `javacall_dir.h`
- `JAVACALL_MAX_LOCALIZED_ROOTS_LIST_LENGTH` – maximum length of a string with a list of localized names of all file system roots, located in the `javacall_fileconnection.h`
- `JAVACALL_MAX_LOCALIZED_DIR_NAME_LENGTH` – maximum length of localized name of a special directory (e.g. `private`, `music`, `photos...`), located in the `javacall_fileconnection.h`

Callback functions

- `javanotify_fileconnection_root_changed()`

This function must be called by the platform every time a file system root gets mounted or unmounted. The function body is platform-independent (implemented by Sun) and does not require any porting.

JavaCall API implementation must notify Java of mount/unmount events as soon as possible – ideally, platform may have an efficient mechanism to invoke the callback function immediately upon root change.

JavaCall API Porting Steps

This section explains how the JSR 75 code can be ported to a new platform using the JavaCall API. The new platform is represented here as `<new_platform>`.

File Connection Implementation

1. Create `<new_platform>/jsr75_pim_fc` directory in the `javacall-com/implementation` directory.
2. Copy `fileconnection.c` from `javacall-com/implementation/stubs/jsr75_pim_fc` to the `<new_platform>/jsr75_pim_fc` directory.
3. Rewrite functions in your new `fileconnection.c` according to the platform specifics.
4. Adjust platform-specific file system constants in `javacall_file.h`, `javacall_dir.h`, and `javacall_fileconnection.h` as necessary.

PIM Implementation

1. Create `<new_platform>/jsr75_pim_fc` directory in the `javacall-com/implementation` directory.
2. Copy `pim.c` from `javacall-com/implementation/stubs/jsr75_pim_fc` to the `<new_platform>/jsr75_pim_fc` directory.
3. Rewrite functions in your new `pim.c` according to the platform specifics.
4. Set the appropriate values for `PIMRootDir` and `file.linebreak` in `jsr75/src/config/javacall/properties_jsr75.xml`.
5. Set value for `JSR_75_PIM_HANDLER_IMPL` to `platform` in `jsr75/src/config/common/config.gmk` file.

Porting the Bluetooth Optional Package

This appendix discusses the design and porting of the Bluetooth optional package (JSR 82) for the Java ME platform. 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).

For a complete description of Bluetooth functionality, see the Bluetooth Specification 1.1 at the following URL:

<http://jcp.org/aboutJava/communityprocess/final/jsr082/index.html>

Bluetooth Overview

Bluetooth technology (the Bluetooth stack) is built upon the Connected Limited Device Configuration and can be loosely divided into two components:

- **Bluetooth controller.** The baseband layer that enables the physical Radio Frequency (RF) link up between two Bluetooth-enabled devices, using the underlying Bluetooth Radio module.
- **Bluetooth host.** The higher level software stack that includes a number of Bluetooth profiles. A Bluetooth profile is a standardized way of defining a protocol and its features. Each profile enables a particular usage model between devices able to share that protocol.

Communication between the Bluetooth controller and the Bluetooth host is managed by the Host Controller Interface (HCI).

Bluetooth Profiles

Bluetooth technology supports a number of profiles, including the following generic profiles:

- **Generic Access Profile (GAP)** - An umbrella profile that includes the other three generic profiles.
- **Bluetooth Serial Port Profile (BTSP)** - A profile used to establish RS-232 serial port connectivity between devices that support it (for example, a cell phone and a headset), using an Radio Frequency Communication (RFCOMM) or Logical Link and Control Application Protocol (L2CAP) transport.
- **Service Discovery Application Profile (SDAP)** - A profile used by client applications to discover services available through a Bluetooth server.
- **Generic Object Exchange Profile (GOEP)** - An underlying profile used for file transfer, synchronization, object push, and other object transactions.

The generic profiles shown here and other functionality available with Bluetooth technology are found in the `javax.bluetooth` package, which depends upon `javax.microedition.io`.

In addition, Bluetooth technology supports the Object Exchange Protocol (OBEX), which is a separate optional package outside the core Bluetooth APIs, it is also used to establish connectivity between Bluetooth clients and Bluetooth servers and can be used to “push” or “pull” objects from one to the other.

OBEX functionality is available in the `javax.obex` package and also depends upon `javax.microedition.io`.

Bluetooth Functionality

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.

Bluetooth Control Center

The Bluetooth Control Center (BCC) is a set of capabilities that allows a user, device manufacturer, or Bluetooth service provider to define specific values for configuration parameters in a Bluetooth stack. The Bluetooth Control Center also makes it possible to manage and resolve conflicting requests made by wireless applications to a Bluetooth implementation, for example, by setting a specific security policy or defining a list of trusted devices.

BluetoothStack Interface and the Bluetooth Control Center

The JSR 82 optimized implementation of Java Wireless Client software uses the `BluetoothStack` interface whenever communication with the underlying Bluetooth stack is required. The `BluetoothStack` interface provides an API for device management and security. Typical scenarios include the following:

- Device inquiry
- Retrieval of a remote device name
- Authentication
- Updating Service Discovery Database (SDDB) service records

The BCC interface is used to provide access to services implemented by a dedicated application, which serves as a central authority for local Bluetooth device settings. Those services include retrieval of cached and preknown devices, pairing, authorization, and so on. The BCC application is usually already available on the target platform and is implemented in native code.

Some functionality of the `BluetoothStack` interface and BCC interface is overlapping. For example, both interfaces have methods for doing the following things:

- Retrieving a local Bluetooth address and friendly name
- Setting and retrieving device and service classes and access codes
- Changing authentication and encryption

In most cases, the default implementation of the BCC interface forwards these requests to the corresponding `BluetoothStack` interface methods. The idea is that the BCC interface provides a higher level of operation, while the `BluetoothStack` interface is used for low-level communication. For example, the `authenticate()` method of the BCC interface might request the user to enter a PIN code before asking

the `BluetoothStack` interface to perform an actual authentication. The overlapping of APIs is also required for the emulation build, where the `BluetoothStack` interface is not used at all.

Note – The native part of the `BluetoothStack` interface is currently implemented in C++, while plain C is used for the native BCC implementation.

Porting the `BluetoothStack` Interface

The `BluetoothStack` interface is presented by an abstract `BluetoothStack` Java programming language class and a corresponding abstract `CBluetoothStack` C++ class. The Java programming language class handles parameter checking, exception handling, synchronization and other higher-level tasks, while the native class takes care of communication with the underlying Bluetooth stack. During porting on the target platform, it may be required to subclass the Java programming language class, native class, or both.

Assumptions About the Implementation

The following assumptions have been made in regard to the underlying stack implementation:

- The Bluetooth stack API provides a non-blocking operation so that the virtual machine is allowed to run during prolonged operations, for example, a device inquiry.
- Certain Bluetooth events, such as an inquiry result or a remote name result, can be polled for, and when available, extracted from an event queue.
- Access to the local Service Discovery Database is provided.

Often, polling for and extraction of events is not directly supported by the Bluetooth stack API. Instead, a Bluetooth stack implementation can provide a callback mechanism for delivering notifications and events. However, it must also be possible to implement polling for a callback-oriented API (for example, a callback function can save event data, which can be retrieved later during a polling call).

For passing Bluetooth events from the native to the Java programming language side, a binary serialization of events is used. To minimize your porting effort, serialize event data in the format used for representing corresponding HCI events, as it enables you to reuse event construction code provided by the `GenericBluetoothStack` interface. However, any other serialization method is also acceptable.

Implementation Requirements

The underlying Bluetooth stack implementation must be able to do the following:

- Check if the Bluetooth radio is enabled, and turn it on if disabled.
- Retrieve local Bluetooth address and friendly name.
- Retrieve device class and set service classes.
- Get and set inquiry access code.
- Start an inquiry process (or report failure if the device is currently busy).
- Cancel an ongoing inquiry.
- Perform remote name retrieval (or enqueue the request if the device is busy).
- Perform asynchronous authentication.
- Request encryption change of all connections with a remote device.

The Bluetooth stack implementation must report the following events:

- Authentication completion, containing connection handle and a flag indicating whether the procedure succeeded
- Encryption change, containing a connection handle and a flag indicating whether the change succeeded

▼ Porting the Bluetooth Stack

1. Determine whether your Bluetooth stack provides access to HCI.

For stacks providing primitives to access HCI layer, the `GenericBluetoothStack` implementation is the best starting point for the port. The HCI specifies a uniform command interface to the Bluetooth controller and covers all the functionality represented in the `BluetoothStack` porting interface. If the target Bluetooth stack provides an API for the HCI, it can be beneficial to use that API, so that the porting effort can be reduced to a minimum. When using this approach, it is often enough to just implement the HCI transport, as all higher-level methods are already provided by the `GenericBluetoothStack` interface implementation.

However, some stack implementations might lack a usable API to access the HCI layer, or provide other reasons to use a high-level API instead. In such cases, use the `BluetoothStack` as a base class for the implementation.

2. Subclass `CGenericBluetoothStack` C++ class.

Usually, the `GenericBluetoothStack` Java programming language class does not need to be subclassed or modified, as it already implements all abstract methods of the `BluetoothStack` accordingly. In the `CGenericBluetoothStack` C++ class, the following three HCI-related abstract methods are declared, which need to be implemented in a subclass:

- `SendHCICommand()` method sends an HCI command to the underlying stack implementation. See the JSR 82 specification for a description of the method arguments.
- `WaitHCICommandComplete()` and `WaitHCICommandStatus()` wait until `CommandComplete` or `CommandStatus` events are received. These methods block virtual machine execution, which does not present a problem because the time between issuing a command and receiving the corresponding event is negligible for the commands used with these methods.

Also, the following event-related methods declared in `CBluetoothStack` class need to be defined:

- `CheckEvents()` method checks HCI events are pending. Only the following HCI events are of interest:
 - Command complete
 - Command status
 - Inquiry complete
 - Inquiry result
 - Remote name
 - Request complete
 - Authentication complete
 - Encryption change

All other HCI events are filtered out.

- `ReadData()` method extracts a single pending event and writes event data into the buffer provided. HCI events are presented in binary format according to the Bluetooth Specification.

3. Subclass the `BluetoothStack` Java programming language class and the `CBluetoothStack` C++ class.

When using higher-level API provided by the Bluetooth stack, it is required to make appropriate API calls in the following C++ class methods:

- `StartInquiry()`
- `CancelInquiry()`
- `AskFriendlyName()`
- `Authenticate()`

- `Encrypt()`
 - All of these methods must not block and must perform asynchronously. Also, the two event-related methods declared in `CBluetoothStack` class need to be defined.
- `CheckEvents()` method checks if Bluetooth events are pending. The following events are of interest:
 - Inquiry complete
 - Inquiry result
 - Remote name result
 - Authentication complete
 - Encryption change complete
- `ReadData()` method extracts a single pending event and writes event data into the buffer provided. The format of event data is arbitrary and can be stack specific.

Bluetooth events are usually provided using callback functions in the stack API. The usual practice is to store events when a callback function is invoked and access the events stored from `CheckEvents()` and `ReadData()` methods.

It is also required to subclass the `BluetoothStack` Java programming language class and override the `retrieveEvent()` method, which converts binary data of an event extracted with `ReadData()` into the corresponding Java programming language class. See the `GenericBluetoothStack` implementation for an example.

4. Implement other device management methods.

The following methods need to be implemented:

- `isEnabled()`
- `enable()`
- `getLocalAddress()`
- `getLocalName()`
- `getDeviceClass()`
- `setServiceClasses()`
- `getAccessCode()`

- `SetAccessCode()`

See the Doxygen-generated documentation for details on these methods and their arguments.

All of these methods are also present in the `BCC` interface and the default `BCC` implementation forwards calls to the `BluetoothStack` interface. However, in some cases, the `BCC` itself is responsible for all or some of the functionality provided by these methods.

Even though all of the methods mentioned can be implemented using the `HCI`, the `GenericBluetoothStack` currently does not provide such an implementation. None of the methods access remote devices and all of them must perform synchronously, so a higher-level API usually can be used. Still, future versions of `GenericBluetoothStack` might include an `HCI`-based implementation, which reduces the porting efforts of an `HCI`-compatible API even further.

5. Implement SDDB functionality.

The Service Discovery Database functionality is contained in two methods:

- `updateRecord()`
- `removeRecord()`

Both methods take a record handle as their first parameter. These handle values are not related to the `ServiceRecordHandle` attributes associated with the service records. Instead, these handles are only used to uniquely identify service records in the upper-level code, and might or might not match the `ServiceRecordHandle` attribute values used by the SDDB internally.

The `updateRecord()` method replaces the existing SDDB entry identified by the given handle with a new one. The new record is passed in a byte array, containing attribute-value pairs in the format identical to the one used in the `AttributeList` parameter of the `SDP_ServiceAttributeResponse` protocol data unit (PDU). See the Bluetooth specification for details. This method returns the new handle value (or the same handle value, if it has not changed).

The `removeRecord()` method permanently deletes an existing SDDB entry identified by the given handle. The handle value specified can then be reused by new SDDB entries.

Porting the BCC Interface

The `BCC` is an abstract Java programming language class. The class currently has two implementations, `JavaBCC` for use with the emulation build and `NativeBCC` for working with Bluetooth radio. Ensure the latter class is instantiated in the `BCC.getInstance()` method.

Following is a list of NativeBCC methods and possible actions you might need to perform when porting:

- `native boolean confirmEnable()`

Before enabling Bluetooth, the user must to be prompted. If the prompt dialog is to be displayed by the native BCC application, the native `bt_bcc_confirm_enable()` function needs to be implemented. If the Java programming language-based dialog is expected, the native modifier is dropped and the method is implemented using the Java programming language modifier.

- `native boolean isConnectedable()`

This method is used to check if the local device is available for connection. The native `bt_bcc_is_connectable()` function returns the connectable status, according to the native BCC application policy.

- `native boolean isPaired()`

This method determines if the local and remote device can proceed with authentication without requesting a PIN from the user. The corresponding native `bt_bcc_is_paired()` function needs to be implemented accordingly.

- `native boolean isAuthenticated()`

This method determines if the remote device has previously been authenticated. The corresponding `bt_bcc_is_authenticated()` function needs to be implemented accordingly. If the authentication status is not known, this function returns `false`.

Note – At least one connection to the remote device must be present. When the last connection to the remote device is closed, clear the “previously authenticated” state.

- `native boolean isTrusted()`

This method determines if the remote device is a trusted device. If the concept of trusted devices is supported by the native BCC application, the corresponding native `bt_bcc_is_trusted()` function needs to be implemented accordingly, otherwise this method always returns `false`.

- `native boolean isEncrypted()`

This method determines if all connections to the specified remote device are encrypted. The corresponding native `bt_bcc_is_encrypted()` function needs to be implemented accordingly.

- `native String getPasskey()`

This method retrieves the PIN code associated with a specified remote device. If the PIN code is not known, the user is prompted to enter one. The corresponding native `bt_bcc_get_passkey()` function needs to be implemented accordingly. In some cases, PIN code retrieval and bonding (pairing) cannot be separated. If

this is the case, the `getPasskey()` method returns any non-null value. A separate method for PIN code retrieval is provided for convenience, or if the UI dialog is implemented in the Java programming language instead.

- `native boolean bond()`

This method performs bonding with the specified remote device using the given PIN code. The corresponding native `bt_bcc_bond()` function needs to be implemented accordingly.

- `native boolean authorize()`

This method asks the user whether to allow a connection from the specified remote device to the given service. The corresponding native `bt_bcc_authorize()` function needs to be implemented accordingly.

- `native String getCachedDevices()` and `native String getPreknownDevices()`

These two methods retrieve a list of Bluetooth addresses of cached and previously known devices from the native BCC application. Separate the addresses by a specific character (the default is a colon). The corresponding native `bt_bcc_get_cached_devices()` and `bt_bcc_get_preknown_devices()` functions need to be implemented accordingly.

- `native int getHandle()`

This method retrieves the default ACL connection handle for the specified remote device. This handle is used for authentication and encryption in the BluetoothStack interface. The corresponding native `bt_bcc_get_handle()` function needs to be implemented accordingly.

- `native int getConnectionCount()`

This method retrieves the number of connections to the specified remote device, including connections from native applications, if any. The corresponding native `bt_bcc_get_connection_count()` function needs to be implemented accordingly.

- `native boolean setEncryption()`

This method increases or decreases the counter of encrypted connection requests for the specified device, including connections from native applications, if any. Once the counter makes a transition from 0 to 1, indicating that the encryption mode needs to be changed, the method must return `true`. The corresponding native `bt_bcc_set_encryption()` function needs to be implemented accordingly.

Additionally, two native methods might need to be implemented:

- `bt_bcc_add_connection()` is called whenever a new logical connection is established between the local and a remote device. The handle argument of this method specifies the default ACL connection handle for the connection.

- `bt_bcc_remove_connection()` is called whenever an existing connection is closed.

L2CAP Protocol and BTSP Profile

The descriptions of the Logical Link and Control Application Protocol (L2CAP) and the Bluetooth Serial Port Profile (BTSP) and have much in common. The major difference is that L2CAP is a packet protocol while BTSP is a stream protocol.

They differ in the following ways:

- L2CAP data is trimmed if it is outside the packet length. It's also possible to send empty packets (without data).
- The L2CAP `getReceiveMTU()` and `getTransmitMTU()` methods return the input and output Maximum Transmission Unit (MTU) sizes negotiated by the underlying stack while establishing connections. BTSP has no such methods.
- The L2CAP protocol has the `ready()` method that determines whether a packet is in the input data queue. In the BTSP protocol, the available method explicitly returns the number of bytes available for reading.

The set of Java programming language classes for each protocol consists of two major classes: One represents the `Connection` class and the other represents the `Notifier` class. Currently, the L2CAP and BTSP protocols are implemented on the Linux platform using the BlueZ library, which provides a GNU socket-based interface.

BlueZ Library

Although the BlueZ library has been in development for many years and is in wide use, it has major bugs, especially in non-blocking mode. Comments in the `bt*Connection.c` files mark workarounds for bugs found in the BlueZ 2.19 library. To install the BlueZ library, you can either build it from the sources downloaded from the official site or get the binaries using the package manager for your version of Linux.

You must also build the Bluetooth kernel modules for your Linux kernel.

Although the library is undocumented, some information can be found in the various forums and mail lists.

The `hcidump` utility analyzes Bluetooth HCI packets and is very useful for debugging.

Making Connections in L2CAP and BTSP

The typical name for the connection class is `*ConnectionImpl`, where (*) is the name of the protocol. `*ConnectionImpl` classes are inherited from the `BluetoothConnectionImpl` class. This is a base class for both protocols. It keeps the things associated with the connection, including the following items:

- URL
- I/O mode
- `RemoteDevice` class
- Security settings (encryption and authorization)

Each `*ConnectionImpl` contains the native methods `initialize()` and `finalize()`. The `initialize()` method retrieves `fieldIDs` used to access the Java platform class members from native code. The `finalize()` method releases native resources. That is, it closes the native connection handle if it is not closed explicitly.

Each connection implementation consists of three parts, where (*) represents L2CAP or BTSP for the equivalent protocol:

- `*ConnectionImpl.java`

The mode-specific functionality is put in separate methods, whose names end with a "0".

Long-running native code is not suitable for protocol operation because it blocks the VM. Therefore, for asynchronous operations (for example, `connect`, `accept`, `send`, and `receive`) use the Java Wireless Client software `midp_thread_wait` and `midp_thread_signal` mechanism. This is a common mechanism utilized by all Java Wireless Client software network protocols.

The following differences exist between the JSR implementation of GNU sockets, `QSocket`s and event-oriented (callback-oriented) stacks.

- If the GNU socket API is used and data is not available, the `bt_na_register_for_write` and `bt_na_register_for_read` functions add the socket into a special array. This array is accessed using `GetRegisteredBtSocketHandles()` declared in `bt_generic.h`. The sockets from the array are put into a common set of sockets checked on every VM time slice. For more information see the `checkForSocketAndKeyboardSignal` function in `midp/src/events/mastermode_port/linux_fb/native/mastermode_check_signal.c`.
- If the QT library is used and data is not available, calling `bt_na_register_for_write` and `bt_na_register_for_read` enables the read/write `QSocket` notifier. The notifier calls the Java Wireless Client software `NotifySocketStatusChanged()` function to make an up-call to unblock the thread.

- If an event-oriented (callback-oriented) stack is used (for example, Broadcomm) and data is not available, the stack must call the Java Wireless Client software `bt_notify_protocol_event()` callback function to unblock the thread.
- `*ConnectionGlue.c`, `*NotifierGlue.c`

`*ConnectionGlue.c` contains implementations for the native methods of `*ConnectionImpl` class. To avoid race conditions, native connection handles stored in the fields of the `*ConnectionImpl` and `*NotifierImpl` Java platform classes are accessed only from native code. All KNI oriented code is located in these two files and the porting layer is utilized to call native platform-specific connection methods.

There are no global native variables except `jfieldID` variables that are set once by static initializers and can be safely used by multiple tasks.

This information holds true for the `*NotifierGlue.c` as well, except for the `*NotifierImpl` class.
- `*Connection.h`, `*Connection.c`

`*Connection.h` is the porting interface declaration for the protocol, while `*Connection.c` contains an implementation for the specific platform (the OS plus the underlying bluetooth stack). This API has GNU style sockets but can also be used with other sockets underlying the Bluetooth stacks. The API contains the client connection and server connection. Some functions can be applied only for a dedicated entity, but other functions are common to all.

Be aware of the following issues when using the ROMizer to tune the `jsr82_rom.config` file:

- Renaming is disabled for the fields accessed using `KNI_GetFieldID`.
- Internal packages are made hidden. Non-ROMized code is prevented from accessing even public methods and fields of classes in these packages.
- Public JSR APIs are restricted. Non-ROMized code cannot load new classes into JSR API packages.

The Java Wireless Toolkit stores JSR public properties as well as internal properties in Java platform files.

The Java Wireless Client software has its own technology to manipulate properties. It uses the following XML configuration files for this purpose. The configuration files are `share`, `emul`, `real`, and `linux`.

▼ Porting L2CAP and BTSP

In the following steps, *platform* is the name of the target OS (for example, Linux) and *protocol* is the name of the protocol being used (the possible values are `bt12cap`, `btsp`, and `irdaobex`).

Note – Do not change files located in directories named `common` during porting.

The following steps describe how to port the Bluetooth optional package to your platform.

1. **Create the `jsr82/src/config/platform/properties_jsr82.xml` file and fill it with valid property values.**

Use `jsr82/src/config/share/properties_jsr82.xml` as a template.

2. **Create the `jsr82/src/protocol/protocol/native/platform/*Connection.c` file that contains an implementation of the protocol's porting layer (`*Connection.h`) for your target platform.**

3. **Create a customized `subsystem.gmk` for the target platform, which is similar to the Linux `subsystem.gmk`.**

4. **Build your target platform executable using the newly-created `subsystem.gmk` file.**

A common `subsystem.gmk` is available for emulation and real modes. Currently `linux_qte` and `linux_fb` are supported in real mode.

This customized `subsystem.gmk` file can include the following declarations:

- `DOXYGEN_INPUT_LIST` - Adds native headers in order to generate Doxygen documentation for them.
- `JAVADOC_ALL_SOURCEPATH` - Adds Java programming language headers to generate documentation using the Javadoc tool.
- `vpath` - Adds paths to new directories containing native source `.c` and `.cpp` files.
- `SUBSYSTEM_JSR_82_NATIVE_FILES` - Contains the names of native files to include them into the build process.
- `SUBSYSTEM_JSR_82_EXTRA_INCLUDES` - Contains paths to newly added header files.
- `SUBSYSTEM_JSR_82_JAVA_FILES` - Java platform source files to be included.
- `SUBSYSTEM_CONFIGURATION_CONFIGURATOR_INPUT_FILES` - Includes properties to the configurator tool.
- `LIBS` - Adds a native library (the key format is platform-specific).

The current release includes its own trace system for debugging. The trace system is located in `btMacros.h`. The following macros are declared: `PRINT_INFO`, `PRINT_ERROR`, and `EXCEPTION_MSG`.

You can independently switch each of the macros on and off. You can switch the `EXCEPTION_MSG` macro on during development in order to catch problems, and switch it off in the production release to reduce the code footprint.

TCK and Java Device Test Suite software can be used to test the implementation. Note that the following permissions must be enabled in the Java Device Test Suite software:

- `Connector.http`
- `Connector.comm`
- `Connector.bluetooth.client`
- `Connector.bluetooth.server`
- `Connector.obex.client.tcp`
- `Connector.obex.server.tcp`
- `Connector.obex.client`
- `Connector.obex.server`
- `Connector.socket`
- `Connector.serversocket`

Porting the OBEX Interface

OBEX is a protocol developed by the Infrared Data Association (IrDA) for pushing and pulling objects to or from clients and servers.

The `irdaobex` protocol is used for devices that make connections using only an infrared port and is implemented in two Java programming language classes: `IrNativeConnection` and `IrNativeNotifier`. The `Native` prefix is used to identify that these classes are mostly implemented in native code and are intended to be used with a non-emulation build. No corresponding implementation is available for an emulation build at the moment.

The porting interface is defined in the `IrConnection.h` header file. Most of the functions declared there must be implemented during the porting process.

In some cases, when the connection is known to be synchronous, `ir_connect_complete` is not required.

JavaCall Porting Layer

All JavaCall API Bluetooth functions and variable types have prefix `javacall_bt_` or `javanotify_bt_`.

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 values are as follows:

- `JAVACALL_OK` means an action has finished correctly.
- `JAVACALL_FAIL` indicates an error occurred during execution.
- `JAVACALL_WOULD_BLOCK` means that the function starts an asynchronous operation and a function will be called on finishing this operation.

Some JavaCall API Bluetooth functions have boolean arguments or return a boolean value (i.e. `isDeviceAuthenticated`). The JavaCall API boolean type is `javacall_bool` and it can be `JAVACALL_TRUE` or `JAVACALL_FALSE`.

The most important type is `javacall_bt_address` which is a pointer to a six byte Bluetooth address. For more information on Bluetooth addresses, see the Bluetooth specification at <http://www.bluetooth.org/spec>.

Definition of JavaCall API Bluetooth Variable Types, Values and Functions

The file `javacall_bt.h` contains the definition of all variable types, values and function prototypes of the JavaCall API Bluetooth interface.

JavaCall API Bluetooth Function Groups

- Bluetooth Control Center (prefix `javacall_bt_bcc_`)
- Bluetooth Stack (prefix `javacall_bt_stack_`)
- Service Discovery Database (prefix `javacall_bt_sddb_`)
- L2CAP protocol (prefix `javacall_bt_l2cap_`)
- RFCOMM protocol (prefix `javacall_bt_rfcomm_`)
- Notification functions (prefix `javanotify_bt_`)

Memory Allocation

Buffers for all output parameters are allocated by the caller, unless specified otherwise.

In case of strings and arrays, a parameter represents the size of the buffer. If output data does not fit into the buffer of the given size, the JavaCall API function must return `JAVACALL_FAIL`.

Porting steps

This section explains how the JSR 82 code can be ported to a new platform using the JavaCall API .

1. All functions contain empty bodies. When a function returns `javacall_result` value it should return `JAVACALL_FAIL`. Boolean return value should be `JAVACALL_FALSE`. Be sure that the build contains no mistakes. Insert trace into each function for debugging.
2. Implement functions from the first three groups (Bluetooth Control Center, Bluetooth Stack, Service Discovery Database).
3. Implement L2CAP protocol functions and run available L2CAP tests.
4. Implement RFCOMM protocol functions and run available RFCOMM tests.
5. (Optional) Implement notification functions if needed.

Location API

This chapter describes the JSR 179 Location API implementation in the Java Wireless Client software. It covers the structure of the source code, available implementations of its components, and a description of the porting steps both for the native layer and the JavaCall layer.

This chapter provides high level information about porting the Location API. For detailed information about each function in the porting layers, check out the API reference documentation that is included with the Java Wireless Client software.

Introduction

The JSR 179 Location API enables mobile applications to determine their current location. This API covers obtaining information about the present geographic location and orientation of the terminal and accessing a database of known landmarks stored in the device.

This API is in a single package, `javax.microedition.location`.

Implementation Description

In this chapter, *jsr179-root* is the root directory of the Location API source code and *javacall-root* is the root directory of the JavaCall API source code.

The Location API has two major parts corresponding to the public classes in `javax.microedition.location`:

- Access to persistent landmark store (`LandmarkStore` classes)

- Access to location information of terminal (`LocationProvider` classes)

These classes use some internal classes from `com.sun.j2me.location`.

`LandmarkStore` currently has two implementations:

- The `java_global` implementation provides all functionality necessary for `LandmarkStore` using Java platform classes. This implementation is platform independent and does not have any native interfaces.
- The `platform_global` implementation has a special dedicated layer of native interfaces that are ported to a specific platform.

The `LandmarkStore` implementation is chosen at compilation by assigning an appropriate value to `JSR_179_STORE_IMPL` option in `jsr179-root/src/config/subsystem.gmk`. See the makefile for the allowed values.

`LocationProvider` has only a `platform_global` implementation. A special dedicated layer of native interfaces is the porting layer.

The `atan2` math function has two implementations:

- The `java_global` platform-independent implementation does not require porting to new platforms.
- The `platform_global` implementation has a native interface that needs to be ported to a specific platform.

The implementation of `atan2` is chosen at compilation by assigning an appropriate value to `JSR_179_ATAN2_IMPL` in `jsr179-root/src/config/subsystem.gmk`.

LandmarkStore Implementation

Several internal properties in `jsr179-root/src/config/common/properties_jsr179.xml` define limitations of `LandmarkStore`:

Name (starts with <code>com.sun.j2me.location.</code>)	Value
<code>CreateLandmarkStoreSupported</code>	true if landmark stores can be created, false otherwise
<code>DeleteLandmarkStoreSupported</code>	true if landmark stores can be removed, false otherwise
<code>DeleteCategorySupported</code>	true if landmark categories can be removed, false otherwise

The `java_global` implementation provides all functionality necessary for the Location API via some Java platform classes. This implementation does not have any interfaces to a platform. RMS record stores are used for the `LandmarkStore` implementation.

The `platform_global` implementation has a native porting layer for an efficient platform-dependent implementation of `LandmarkStore`. The `platform_global` implementation is appropriate if the `LandmarkStore` needs to be accessible from both Java platform applications and native applications.

LocationProvider implementation

The `LocationProvider` implementation has a dedicated native layer that hides all interactions with the device and is a porting layer for an efficient platform-dependent implementation.

Several internal properties in `jsr179-root/src/config/common/properties_jsr179.xml` define limitations of **LocationProvider** functionality:

Name (starts with <code>com.sun.j2me.location.</code>)	Value
<code>OrientationSupported</code>	true if the device can sense its orientation, false otherwise
<code>ProximitySupported</code>	true if the device supports proximity monitoring, false otherwise
<code>ResetTimeout</code>	Timeout, in seconds, for <code>LocationProvider.reset()</code>

Atan2 Implementation

The `java_global` implementation is platform independent.

The `platform_global` implementation is appropriate if your platform has its own optimized implementation.

Location API Code Structure

The Location API source code is as follows:

```

jsr179-root/src
  common      - common classes
    classes
      com      - implementation independent internal classes
      javax    - JSR 179 API package
    native    - native code for common part of JSR 179
      share    - platform independent parts of LocationProvider
      linux    - linux implemetation of LocationProvider
      javacall - JavaCall API implemetation of LocationProvider
      stubs    - empty implementation of LocationProvider
  config      - files for build configuration
  include     - Native porting API header
  java_global
    classes
      com      - LandmarkStore implementation
      native
        math   - atan2 implementation
  platform_global
    classes
      com      - LandmarkStore internal classes
      native
        javacall - JavaCall API implemetation of LandmarkStore and atan2
        share    - platform independent native source code
  tests
    i3test    - the JSR 179 i3 tests

javacall-root
  interface
    jsr179_location - JavaCall API porting API headers
  implementation
    stubs
      jsr179_location empty implementations
    win32
      jsr179_location win32 implementations

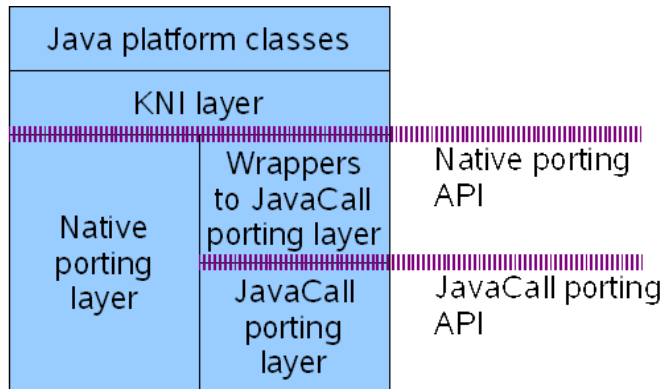
```

Porting layer functionality

The major purpose of the porting API is an interaction with your device's location hardware to access location information a platform database of landmarks.

The `platform_global` implementation could be presented as the following set of layers:

FIGURE 4-1 Location API code structure



This document naturally focuses on the native porting API and the JavaCall porting API. All the platform-dependent functionality is placed in this code layer. To make the Location API work on a new platform, you must implement all the functions below the native or JavaCall porting API for the given target.

Native Porting Layer

This API covers accessing a database of known landmarks stored in the terminal and the present geographic location and orientation of the terminal.

Accessing Landmarks

Here are optional features whose availability depends on the landmark store implementation of the device and its possible relation to landmark stores shared with native applications:

- Creating and deleting landmark stores
- Adding and removing landmark categories

The porting layer includes the following:

- Gets landmark stores:
 - `jsr179_landmarkstorelist_open()`
 - `jsr179_landmarkstorelist_close()`
 - `jsr179_landmarkstorelist_next()`
- Gets landmarks from native landmark store:

- `jsr179_landmarklist_open()`
- `jsr179_landmarklist_close()`
- `jsr179_landmarklist_next()`
- Gets categories from native landmark store:
 - `jsr179_categorylist_open()`
 - `jsr179_categorylist_close()`
 - `jsr179_categorylist_next()`
- Adds or deletes a landmark to a landmark store or category:
 - `jsr179_landmark_add_to_landmarkstore()`
 - `jsr179_landmark_add_to_category()`
 - `jsr179_landmark_delete_from_landmarkstore()`
 - `jsr179_landmark_delete_from_category()`
- (Optional) Creates or deletes a landmark store or category:
 - `jsr179_landmarkstore_create()`
 - `jsr179_landmarkstore_delete()`
 - `jsr179_category_add()`
 - `jsr179_category_delete()`

Getting the Current Location

To calculate distance or azimuth, a platform-independent `atan2` function implementation is provided. However, if your device can calculate `atan2` by means of a native math library, it is efficient to use it.

The acquired location includes the following information:

- Only Latitude and Longitude are mandatory
- Altitude (`Float.NaN` if unknown)
- HorizontalAccuracy and VerticalAccuracy in meters (`Float.NaN` if unknown)
- LocationMethod (zero if unknown)
- Speed, Course (`Float.NaN` if unknown)
- AddressInfo (provided if supported)
- ExtraInfo (provided if supported)

For the specified location provider, the following information is provided:

- Default time related properties of a location provider
- The default interval, which specifies how often updates are provided for listener

- The default maximum age, which specifies the acceptable age for cached information for listener
- The default timeout, which specifies how late update can be from interval for listener
- The average time of receiving next location
- The recommended time interval for asking a new state
- The other properties for Criteria

Following are the optional features whose availability depends on the used location methods:

- Altitude information
- Accuracy of altitude
- Course and speed information
- Textual address information related to the location
- Proximity monitoring

The porting layer includes the following functionality:

- Opens a location provider and gets information
 - `jsr179_provider_open()`
 - `jsr179_provider_close()`
 - `jsr179_provider_getinfo()`
 - `jsr179_notify_location_event()`
- Gets the current status of a location provider
 - `jsr179_provider_state()`
- Gets the location info on request basis
 - `jsr179_update_set()`
 - `jsr179_update_cancel()`
- (Optional) Gets the proximity events
 - `jsr179_proximity_monitoring_add()`
 - `jsr179_proximity_monitoring_cancel()`
 - `jsr179_notify_location_proximity()`

Acquisition of Terminal Orientation

The following features are optionally supported:

- Compass azimuth of the terminal orientation
- Pitch and roll 3D terminal orientation information

If the implementation chooses to support orientation, it must provide the azimuth information. Providing pitch and roll is optional.

The porting layer includes the following functionality:

- Opens and closes an orientation provider
 - `jsr179_provider_open()`
 - `jsr179_provider_close()`
- Gets orientation information
 - `jsr179_orientation_start()`
 - `jsr179_orientation_finish()`
 - `jsr179_notify_location_event()`

Implementation Notes

This section describes general principles of the Location API.

Asynchronous Operation

Some functions operate asynchronously. In this case, the function returns quickly with `JSR179_STATUSCODE_WOULD_BLOCK`. When the operation completes, an event is required to be sent from the platform. See section on events for related information.

The following are the asynchronous functions:

- Opening location provider
- Orientation update
- Location update once
- Proximity update

Buffer Allocation

The following rules apply to buffer allocation:

- The buffer for an output parameter must be allocated by the caller.
- A parameter for the size of the buffer must be present if no predefined maximum size exists.

The exceptional cases are enumeration functions like `*list_next()`, which have the following rules:

- The platform allocates memory and returns its pointer in this case.
- The allocated memory is valid until the enumeration is closed.

Mandatory LandmarkStore Functions

- `jsr179_landmark_add_to_landmarkstore()`
- `jsr179_landmark_add_to_category()`
- `jsr179_landmark_update()`
- `jsr179_landmark_delete_from_landmarkstore()`
- `jsr179_landmark_delete_from_category()`
- `jsr179_landmarkstorelist_open()`
- `jsr179_landmarkstorelist_close()`
- `jsr179_landmarkstorelist_next()`
- `jsr179_landmarklist_open()`
- `jsr179_landmarklist_close()`
- `jsr179_landmarklist_next()`
- `jsr179_category_add()`
- `jsr179_categorylist_open()`
- `jsr179_categorylist_close()`
- `jsr179_categorylist_next()`

Optional LandmarkStore Functions

If `com.sun.j2me.location.CreateLandmarkStoreSupported` is set to true, implement `jsr179_landmarkstore_create()`.

If `com.sun.j2me.location.DeleteLandmarkStoreSupported` is set to true, implement `jsr179_landmarkstore_delete()`.

If `com.sun.j2me.location.DeleteCategorySupported` is set to true, implement `jsr179_category_delete()`.

Mandatory LocationProvider Functions

- `jsr179_property_get()`
- `jsr179_provider_getinfo()`

- `jsr179_provider_open()`
- `jsr179_provider_close()`
- `jsr179_provider_state()`
- `jsr179_update_set()`
- `jsr179_update_cancel()`
- `jsr179_location_get()`

Optional LocationProvider Functions

The following functions can be implemented to obtain additional location information:

- `jsr179_get_extrainfo()`
- `jsr179_get_addressinfo()`

The following functions shall be implemented if `com.sun.j2me.location.OrientationSupported` is set to true:

- `jsr179_orientation_start()`
- `jsr179_orientation_finish()`

The following functions shall be implemented if `com.sun.j2me.location.ProximitySupported` is set to true:

- `jsr179_proximity_monitoring_add()`
- `jsr179_proximity_monitoring_cancel()`

Optional Atan2 Function

`jsr179_atan2()` should be implemented if the `platform_global` implementation of `atan2` is chosen in `jsr179-root/src/config/subsystem.gmk`.

Callback functions

You must implement `jsr179_notify_location_event()`.

If `com.sun.j2me.location.ProximitySupported` is set to true, implement `jsr179_notify_location_proximity()`.

Porting JSR 179

This section explains how the JSR 179 code can be ported to a new platform. The new platform is represented here as *new-platform*.

Implementing LocationProvider

1. Create *new-platform* directory in *jsr179-root/src/common/native*.
2. Create *jsr179_locationProvider_impl.c* file in *jsr179-root/src/common/native/new-platform* and implement `LocationProvider` functions according to your platform.
3. Set values for `com.sun.j2me.location.OrientationSupported` and `com.sun.j2me.location.ProximitySupported` in *jsr179-root/src/config/common/properties_jsr179.xml*.

Implementing LandmarkStore

1. Create *new-platform* directory in *jsr179-root/src/platform_global/native*.
2. Create *jsr179_landmarkStore_impl.c* in *jsr179-root/src/platform_global/native/new-platform* and implement `LandmarkStore` functions according to your platform.
3. Set values for `com.sun.j2me.location.CreateLandmarkStoreSupported`, `com.sun.j2me.location.DeleteLandmarkStoreSupported` and `com.sun.j2me.location.DeleteCategorySupported` in *jsr179-root/src/config/common/properties_jsr179.xml* file.
4. Set `JSR_179_STORE_IMPL` to platform in *jsr179-root/src/config/subsystem.gmk*.

Implementing atan2

1. Create *new-platform* directory in *jsr179-root/src/platform_global/native*.
2. Create *jsr179_math_kni.c* in *jsr179-root/src/common/native/new-platform* and implement `jsr179_atan2()` for your platform.
3. Set `JSR_179_ATAN2_IMPL` to platform in *jsr179-root/src/config/subsystem.gmk*.

JavaCall Porting Layer

The location functions in the JavaCall API are much like the native porting functions, but porting occurs in a different layer.

Accessing Landmarks

The following optional features' availability depends on the landmark store implementation of the device and its possible relation to landmark stores shared with native applications:

- Creating and deleting landmark stores
- Adding and removing landmark categories

The porting layer includes the following functionality:

- Gets landmark stores
 - `javacall_landmarkstore_list_open()`
 - `javacall_landmarkstore_list_close()`
 - `javacall_landmarkstore_list_next()`
- Gets landmarks from native landmark store
 - `javacall_landmarkstore_landmarklist_open()`
 - `javacall_landmarkstore_landmarklist_close()`
 - `javacall_landmarkstore_landmarklist_next()`
- Gets categories from native landmark store
 - `javacall_landmarkstore_categorylist_open()`
 - `javacall_landmarkstore_categorylist_close()`
 - `javacall_landmarkstore_categorylist_next()`
- Adds or deletes a landmark to landmark store or category
 - `javacall_landmarkstore_landmark_add_to_landmarkstore()`
 - `javacall_landmarkstore_landmark_add_to_category()`
 - `javacall_landmarkstore_landmark_delete_from_landmarkstore()`
 - `javacall_landmarkstore_landmark_delete_from_category()`
- (Optional) Creates or deletes a landmark store or category
 - `javacall_landmarkstore_create()`
 - `javacall_landmarkstore_delete()`

- `javacall_landmarkstore_category_add()`
- `javacall_landmarkstore_category_delete()`

Getting the Current Location

To calculate distance or azimuth, the platform independent `atan2` function implementation is provided. However, if your platform provides `atan2` functionality by means of a native math library, it is more efficient to use it.

The acquired location should include following information:

- Only Latitude and Longitude are mandatory
- Altitude (`Float.NaN` if unknown)
- HorizontalAccuracy and VerticalAccuracy in meters (`Float.NaN` if unknown)
- LocationMethod (zero if unknown)
- Speed, Course (`Float.NaN` if unknown)
- AddressInfo (provided if supported)
- ExtraInfo (provided if supported)

For the specified location provider, the following information should be provided:

- Default time related properties of a location provider
- The default interval, which is how often updates are generated for listener.
- The default maximum age, which is the acceptable age for cached info for listener
- The default timeout, which specifies how late update can be from interval for listener
- The average time of receiving next location
- The recommended time interval for asking a new state
- The other properties for Criteria

The following optional features' availability depends on the used location methods.

- Altitude information
- Accuracy of altitude
- Course and speed information
- Textual address information related to the location
- Proximity monitoring

The porting layer includes the following functionality:

- Opens a location provider and gets information
 - `javacall_location_provider_open`

- javacall_location_provider_close
- javacall_location_provider_getinfo
- javanotify_location_event
- Gets the current status of a location provider.
 - javacall_location_provider_state
- Gets the location info on request basis.
 - javacall_location_update_set
 - javacall_location_update_cancel
- (Optional) Gets the proximity events
 - javacall_location_proximity_monitoring_add
 - javacall_location_proximity_monitoring_cancel
 - javanotify_location_proximity

Acquisition of Terminal Orientation

The following features are optionally supported:

- Compass azimuth of the terminal orientation
- Pitch and roll 3D terminal orientation information

If the implementation chooses to support orientation, it must provide the azimuth information. Providing pitch and roll is optional.

The porting layer includes the following functionality:

- Opens and closes an orientation provider
 - javacall_location_provider_open
 - javacall_location_provider_close
- Gets orientation information
 - javacall_location_orientation_start
 - javacall_location_orientation_finish
 - javanotify_location_event

Implementation Notes

This section describes general principles of the Location API.

Asynchronous Operation

Some functions operate asynchronously. In this case, the function returns quickly with `JAVACALL_WOULD_BLOCK`. When the operation completes, an event is required to be sent from the platform. See the section on events for related information.

The asynchronous functions are as follows:

- Opening location provider
- Orientation update
- Location update once
- Proximity update
- Unicode string representation

All unicode strings used in this API are `NULL` terminated.

Buffer Allocation

The following rules apply to buffer allocation:

- The buffer for an output parameter must be allocated by the caller.
- A parameter for the size of the buffer must be present if no predefined maximum size exists.

The exceptional cases are enumeration functions like `*list_next()`, which have the following rules:

- The platform allocates memory and returns its pointer in this case.
- The allocated memory is valid until the enumeration is closed.

Mandatory LandmarkStore Functions

- `javacall_landmarkstore_landmark_add_to_landmarkstore()`
- `javacall_landmarkstore_landmark_add_to_category()`
- `javacall_landmarkstore_landmark_update()`
- `javacall_landmarkstore_landmark_delete_from_landmarkstore()`
- `javacall_landmarkstore_landmark_delete_from_category()`
- `javacall_landmarkstore_list_open()`
- `javacall_landmarkstore_list_close()`
- `javacall_landmarkstore_list_next()`
- `javacall_landmarkstore_landmarklist_open()`

- `javacall_landmarkstore_landmarklist_close()`
- `javacall_landmarkstore_landmarklist_next()`
- `javacall_landmarkstore_category_add()`
- `javacall_landmarkstore_categorylist_open()`
- `javacall_landmarkstore_categorylist_close()`
- `javacall_landmarkstore_categorylist_next()`

Optional LandmarkStore Functions

If `com.sun.j2me.location.CreateLandmarkStoreSupported` is set to true, implement `javacall_landmarkstore_create()`.

If `com.sun.j2me.location.DeleteLandmarkStoreSupported` is set to true, implement `javacall_landmarkstore_delete()`.

If `com.sun.j2me.location.DeleteCategorySupported` is set to true, implement `javacall_landmarkstore_category_delete()`.

Mandatory LocationProvider Functions

- `javacall_location_property_get()`
- `javacall_location_provider_getinfo()`
- `javacall_location_provider_open()`
- `javacall_location_provider_close()`
- `javacall_location_provider_state()`
- `javacall_location_update_set()`
- `javacall_location_update_cancel()`
- `javacall_location_get()`

Optional LocationProvider functions

Implement the following functions to obtain additional location information:

- `javacall_location_get_extrainfo()`
- `javacall_location_get_addressinfo()`

If `com.sun.j2me.location.OrientationSupported` is set to `true`, implement `javacall_location_orientation_start()` and `javacall_location_orientation_finish()`.

If `com.sun.j2me.location.ProximitySupported` is set to `true`, implement `javacall_location_proximity_monitoring_add()` and `javacall_location_proximity_monitoring_cancel()`.

Optional Atan2 Function

Implement `jsr179_atan2` if `platform_global` implementation of `atan2` is chosen in `jsr179-root/src/config/subsystem.gmk`.

Callback Functions

You must implement `javanotify_location_event()`.

If `com.sun.j2me.location.ProximitySupported` is set to `true`, implement `javanotify_location_proximity()`.

Porting JSR 179

This section explains how the JSR 179 code can be ported to a new platform using the JavaCall API. The new platform is represented here as *new-platform*.

Implementing LocationProvider

1. Create `new-platform/jsr179_location` directory in `javacall-com/implementation` directory.

Copy `location.c` from `javacall-com/implementation/win32/jsr179_location` to the `new-platform/jsr179_location` directory.

2. Rewrite functions in your new `location.c` according to the platform specifics.
3. Set the appropriate values for `com.sun.j2me.location.OrientationSupported` and `com.sun.j2me.location.ProximitySupported` properties in `jsr179-root/src/config/common/properties_jsr179.xml`.

Implementing LandmarkStore

1. Create *new-platform/jsr179_location* directory in `javacall-com/implementation` directory.
2. Copy `landmarkstore.c` from `javacall-com/implementation/win32/jsr179_location` to the *new-platform/jsr179_location* directory.
3. Rewrite functions in your new `landmarkstore.c` according to the platform specifics.
4. Set the appropriate values for `com.sun.j2me.location.CreateLandmarkStoreSupported`, `com.sun.j2me.location.DeleteLandmarkStoreSupported` and `com.sun.j2me.location.DeleteCategorySupported` in *jsr179-root/src/config/common/properties_jsr179.xml*.
5. Set value for `JSR_179_STORE_IMPL` to `platform` in *jsr179-root/src/config/subsystem.gmk* file.

Implementing atan2

1. Create *new-platform/jsr179_location* directory in `javacall-com/implementation` directory.
2. Copy `location.c` from `javacall-com/implementation/win32/jsr179_location` to the *new-platform/jsr179_location* directory.
3. Rewrite `javacall_location_atan2()` in your new `location.c` according to the platform specifics.
4. Set value for `JSR_179_ATAN2_IMPL` variable to `platform` in *jsr179-root/src/config/subsystem.gmk* file.

Integrating the Scalable 2D Vector Graphics Optional Package

The Scalable 2D Vector Graphics API optional package is included with the Java Wireless Client software. This optional package implements the API defined in the Scalable 2D Vector Graphics Specification (JSR 226). See <http://jcp.org/en/jsr/detail?id=226> for more information.

This optional package requires no additional porting effort because the native porting layer uses existing MIDP porting APIs. The only component in this optional package that interfaces with MIDP is the software renderer.

The software renderer requires only that the `pisces_drawRGB()` function be ported. This function copies a sequence of pixels rendered by the Pisces software to the native peer of a MIDP `Graphics` object.

The Java Wireless Client software implementation of the `pisces_drawRGB()` function is located at `installDir/pisces/src/native/midp/src/JGraphicsSurfaceDestination.c`.

The Java Wireless Client software version of this function is implemented during your port of MIDP by calling `gx_draw_rgb()`, which is declared by MIDP in the header file `gx_graphics.h`.

Mobile Internationalization API

The JSR 238 API consists of three main classes:

- `ResourceManager` provides the plumbing your application needs to retrieve resource strings, images, and other objects.
- `Formatter` knows how to represent numbers, currency, dates, and times for a specific language or region.
- `StringComparator` understands how to sort strings (collate) based on the rules for a particular language.

The Java Wireless Client software provides two implementations for each of these classes.

- The `java_global` implementation is written entirely in the Java programming language. You can use the `java_global` implementation without porting.
- The `platform_global` implementation is a combination of Java platform code and C code.

To select the implementation for each class, set the corresponding variable in `jsr238/src/config/config.gmk`. The value for each is either `java` or `platform`. By default, all three classes use the `java_global` implementation.

```
JSR_238_COLLATION_IMPL = java
JSR_238_FORMAT_IMPL = java
JSR_238_RESOURCES_IMPL = java
```

Note that the `java_global` implementation of `Formatter` depends on the `java_global` implementation of `ResourceManager`.

Porting java_global

When porting the `java_global` implementation to a new platform, keep in mind the set of supported locales for each of the main classes. Each class can have different set of locales.

For `Formatter`, the supported locales are determined by the presence of corresponding resource files, containing formatting strings, symbols, and patterns.

You can add a locale by creating `common.res` and placing it in the `global` directory, in a subdirectory named for the locale. In addition, update the metafile `_common`. You can use a JSR 238 resource file editor, like the one in the Sun Java Wireless Toolkit.

The file `common.res` has a standard format used by `ResourceManager` and described in JSR 238. It must contain resources with following types and IDs:

```
TYPE_NUMBER_FORMAT_SYMBOLS = (byte) 0xfe;
Constants.NUMBER_FORMAT_SYMBOL_RESOURCE_ID=7FFFFFFD
```

```
TYPE_DATETIME_FORMAT_SYMBOLS = (byte) 0xfd;
Constants.DATETIME_FORMAT_SYMBOL_RESOURCE_ID=7FFFFFFE
```

The format of these resources can be seen from the following methods:

```
write(java.io.OutputStream out) of class
    com.sun.j2me.global.NumberFormatSymbols
```

```
write(java.io.OutputStream out) of class
    com.sun.j2me.global.DateFormatSymbols
```

The `java_global` implementation of `Formatter` supports the following locales:

- cs
- cs-CZ
- en
- en-US
- he
- he-IL
- ja
- ja-JP
- sk
- sk-SK
- zh
- zh-CN

In the current implementation, the behavior of a language-only locale is identical to the corresponding language-with-country locale.

Usually, it is not a good idea to have language-only locales for `Formatter`, because it is not clear what values to use as default currencies.

The `java_global` implementation of `StringComparator` supports the following locales:

- `en`
- `en-US`
- `he`
- `he-IL`
- `sk`
- `sk-SK`
- `cs`
- `cs-CZ`
- `es`
- `es-ES`

The number of string comparator locales is determined in `jsr238/src/config/properties_jsr238.xml`.

Tailored rules are applied to the Unicode Collation Algorithm table located in `_coltable.bin` and containing “hot fixes” of general collation rules applied for some specific locale.

The `java_global` implementation has some limitations on supported locales. Locales that uses “backward ordering” or “context” in collation rules (French or Russian, for example) probably do not work correctly.

The `java_global` implementation of `ResourceManager` supports the same locales as `Formatter` because it relies on same resource files (`common.res`) existing for each supported locale.

JSR 238 does not define what resources need to be available as platform `DEVICE` resources and what IDs they need to have. This means `DEVICE` resources are useful only for a certain implementation. The current `java_global` implementation provides the following string resources for each locale:

- ‘Edit’ with ID 0x65
- ‘Next’ with ID 0x66
- ‘Back’ with ID 0x67

This list can be expanded by adding a new resource to `common.res` for each supported locale. Document the IDs you add to make them available for other developers in your organization.

Porting with the JavaCall API

The JavaCall API for `Formatter` relies on the ability of the underlying platform to format dates and numbers for different locales.

To format dates and numbers, the implementation asks the platform to convert an integer or float number to a text representation without any formatting. Two JavaCall API functions, `javacall_mi18n_double_to_ascii()` and `javacall_mi18n_long_to_ascii()` do the work.

If the native platform is `libc`-compliant, the functions `ltoa()` and `ecvt()` can be used.

The text representation of the number is then passed to formatting functions to apply locale rules. These functions use formatting patterns for different format types and local symbols for digits and separators. It is likely that the platform has a direct API to do this, but if not, it must at least have access to local formatting symbols and patterns. The existing `win32` JavaCall API implementation can be reused to format a number string according to this information.

You might encounter problems with the TCK on an implementation that works with Arabic or Indic locales. The TCK expects the implementation to use only regular digits 0 through 9 in formatting numbers and data, and also it always expects a dash (-) as the minus sign symbol and a percent sign (%) as the percent sign symbol used. Further, it relies on the minus sign going before a number. While it is not correct for some locales, the TCK will fail if it is done another way. To fulfill these requirements, two build variables (`NOT_USE_NATIVE_DIGITS`, `MINUS_ALWAYS_INFRONT`) must be defined (set to 1) when building the JavaCall API to satisfy the TCK but compromise correct behavior.

Converting platform locales to JSR 238 locales might be a little tricky. JSR 238 locales consist of an ISO 639 language code, an optional (but recommended) ISO 3166 country code, and an optional variant. Platform locales can consist of more parts, like language, region, script, currency variant, collation variant, calendar variant, and more. Your implementation must extract locales for pure languages and languages with countries, and provide other locales as variants to them. Here is an example:

- `uz`
- `uz-UZ`
- `uz-UZ-Latin`
- `uz-UZ-Cyrillic`

Part of the JavaCall layer `StringComparator` API is a function that asks the platform to compare strings using a specific JSR 238 level. The main problem is to convert platform collation flags to JSR 238 collation levels. These are the main rules:

- **Level 1** - Compares strings in alphabetic order ignoring case, accents and non-letter symbols
- **Level 2** - Same as level 1 but takes accent signs (non-spacing marks) in account
- **Level 3** - Takes letter case into account
- **Identical** - Takes all aspects into account (like scripting, width, and decorations)

Before comparing, strings must be put to Normalized Canonical Decomposition Form (NFD). Usually, this is done transparently by collation functions on Unicode-supported platforms.

If string normalization and collation is not supported by platform, you can choose to use the `java_global` implementation for `StringComparator`, which is independent from other classes.

The JavaCall layer `ResourceManager` API is responsible only for accessing some predefined platform (`DEVICE`) resources. All application resources are handled on the Java platform level.

Platform resources can be strings, images, or sounds existing on the platform.

The API includes functions for querying the resource type and length and for reading resource data by small chunks (in case the resource is large, such as a large sound file).

All string resources must be passed in a UTF-8 encoded byte array, so the implementation must support encoding of native strings to UTF-8.

All platform resource IDs can be taken from corresponding platform APIs or assigned explicitly and published for future developers.

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

A

atan2
implementation, 33

B

Bluetooth
BluetoothStack, 15
BlueZ library, 23
BTSP, 23
L2CAP, 23
overview, 13
porting, 17
porting BluetoothStack, 16
porting with JavaCall API, 28

F

FileConnection
design, 3
devices, 4
JavaCall API, 7
porting, 5
storage directories, 10
system properties, 5
FileConnection API, 3

J

JSR 75
building, 6
porting with JavaCall API, 11
testing, 7

L

LandmarkStore
implementation, 32
Location API
callbacks, 40
code structure, 33
implementation, 31
JavaCall layer, 42
native porting layer, 35
overview, 31
porting, 41
porting layer, 34
porting with JavaCall API, 47
LocationProvider
implementation, 33

M

Mobile Internationalization API, 51
porting, 52
porting with JavaCall API, 54

O

OBEX
porting, 27
orientation, 37

P

PIM
JavaCall API, 9
PIM API, 3
porting
JavaCall API, 1

overview, 1
process, 1

S

source code
 types, 1
SVG, 49
 Pisces renderer, 49