



# Optional APIs Porting Guide Volume I: Sun APIs

---

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

Sun Microsystems, Inc.  
[www.sun.com](http://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, JDK, Javadoc, J2ME, J2SE, J2EE, Java Card, 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, JDK, Javadoc, J2ME, J2SE, J2EE, Java Card, 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

---

<b>Preface</b>	<b>vii</b>
<b>1. Overview</b>	<b>1</b>
<b>2. Porting the WMA Optional Packages</b>	<b>3</b>
Design	4
Design Overview	4
Internal and External Push	5
Inbox and Message Pool	5
Asynchronous Message Transfers	6
Porting Notes	7
Message Porting APIs	7
Message Pool APIs	9
Interaction With the Push Subsystem	10
WMA Push Connections	10
Interactions With the Permission Management Subsystem	11
<b>3. Porting the Mobile Media API</b>	<b>13</b>
Capabilities	13
Porting Information	15

<b>4. Porting the SATSA Optional Package</b>	<b>17</b>
SATSA Code Structure	17
Security Elements	20
SATSA-APDU Package	21
CardDevice Implementation	21
Porting Layer I	22
▼ Implementing Porting Layer I	23
Porting Layer II	23
▼ Implementing Porting Layer II	24
Porting with the JavaCall API	24
Initialization and Finalization	25
Data Exchange	25
Locking	25
Retrieving Information	26
Error Handling	26
Simple Implementation of the APDU Package	27
Platform API	27
Build Options	28
SAT Connection	28
SATSA-CRYPTO Package	29
Implementation	29
Porting Notes	30
SATSA-PKI package	31
Implementation	31
Porting Notes	33
Static Access Control Mechanism	34
▼ Looking for an Access Control File	34
Definitions	36

Reference Implementation Version of ACF 37

Porting Notes 37

**5. Porting CHAPI 39**

CHAPI Code Structure 39

Porting Layer Functionality 40

Porting API Implementation Cases 42

Porting CHAPI 43

Porting Steps for the JavaCall API 44

**6. OpenGL ES 45**

**Glossary 47**

**Index 51**



# Preface

---

This guide describes how to port Sun Java Wireless Client Software 2.0 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](#) contains a brief description of the rest of the book.

[Chapter 2](#) describes porting the Wireless Messaging API defined by JSR 120 and JSR 205.

[Chapter 3](#) explains how to port the JSR 135 Mobile Media API.

[Chapter 4](#) covers porting the JSR 177 Security and Trust Services API.

[Chapter 5](#) discusses porting the JSR 211 Content Handler API.

[Chapter 6](#) describes porting the JSR 239 OpenGL ES API.

---

## Using Operating System Commands

This document does not contain information on basic UNIX<sup>®</sup> 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.
<b>AaBbCc123</b>	What you type, when contrasted with on-screen computer output	% <b>su</b> 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 belong wholly or partly to Sun Microsystems, Inc. *Volume II* of the *Optional APIs Porting Guide* covers optional APIs that do not belong to Sun. You receive *Volume II* only if you have the corresponding product bundle.

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 Sun optional APIs and their porting requirements:

Optional API	Porting Requirements
JSR 120 and JSR 205	See <a href="#">Chapter 2</a>
JSR 135	See <a href="#">Chapter 3</a>
JSR 172	The implementation consists entirely of Java platform code. No porting is necessary.
JSR 177	See <a href="#">Chapter 4</a>
JSR 211	See <a href="#">Chapter 5</a>
JSR 239	See <a href="#">Chapter 6</a>

## Porting the WMA Optional Packages

---

This chapter discusses the design and porting of the Java Wireless Client software's wireless messaging subsystem. The wireless messaging subsystem implements the messaging APIs defined in the Wireless Messaging APIs (WMA) JSR 120 and JSR 205. See <http://www.jcp.org/en/jsr/detail?id=120> and <http://www.jcp.org/en/jsr/detail?id=205> for more information.

This chapter describes porting both the JSR 120 functionality and the additional functionality provided by JSR 205. Depending on the bundles of Java Wireless Client software you have available, you might or might not have the JSR 205 functionality available.

WMA specifies an API that provides a general messaging framework and a general mechanism for establishing a messaging application. The Wireless Messaging API specification also provides more explicit information about using the API to support the Global System for Mobile Communications (GSM) Short Message Service (SMS), Code Division Multiple Access (CDMA) SMS, GSM Cell Broadcast Service (CBS), and Multimedia Message Service (MMS) protocols.

WMA is based on the CLDC Generic Connection Framework (GCF) and is built on top of Java Wireless Client software's networking subsystem. The subsystem's design provides the notion of opening a connection based on a string address, and opening a connection in either client or server mode. The wireless messaging subsystem interacts with the networking subsystem, permission management subsystem, and the push subsystem. To support the messaging API, these subsystems must be extended in the ways described later in this chapter.

---

# Design

The wireless messaging service has the following features:

- It emulates the SMS, CBS, and MMS stacks so that the WMA functionality can be demonstrated and tested.
- It provides a porting layer so that the subsystem can easily be moved to a system that provides native support for SMS, CBS, and MMS.

## Design Overview

The WMA subsystem has the following three components or layers.

- **An interface layer that contains the wireless messaging API** - MIDlet developers use the interface layer that contains the messaging API when they write wireless messaging applications. The API is generic and independent of any messaging protocol. The messaging API is defined in the `javax.wireless.messaging` package.
- **An implementation layer (also the GCF porting layer) that contains the implementation classes** - The implementation layer contains classes that enable MIDlets to access wireless messaging on a mobile device. These classes implement the entire interface layer, except for the `MessageListener` interface that is implemented by a MIDlet using the messaging API.

The implementation layer also drills into the low-level native layer to access the platform's native API.

The following examples, among others, of the implementation layer are included with the Java Wireless Client software:

- `jsr120/src/protocol/sms/classes/com/sun/midp/io/j2me/sms/Protocol.java`
- `jsr120/src/protocol/sms/classes/com/sun/midp/io/j2me/sms/BinaryObject.java`
- **A low-level native transport layer that contains the low-level transport mechanisms** - The native low-level transport layer defines a porting API. An implementation of this API, using the platform's native capabilities, implements the protocols that carry messages to and from the device. It also listens for the arrival of SMS, CBS, and MMS messages and caches them in a message pool for subsequent retrieval. If a MIDlet using these protocols is running, it notifies the application's message listener.

The Java Wireless Client software implementation of this porting API uses datagrams to send and receive WMA messages. Although this is acceptable given Java Wireless Client software's target platform (which has no native WMA

support), it is not the right solution for all platforms. Most platforms that support WMA have a full-featured set of native SMS, CBS, and MMS APIs and use this native support to implement the porting API.

The Message Transport porting APIs are defined in the following header files:

- `jsr120/src/protocol/sms/native/share/inc/jsr120_sms_protocol.h`
- `jsr120/src/protocol/cbs/native/share/inc/jsr120_cbs_protocol.h`
- `jsr205/src/protocol/mms/native/share/inc/jsr205_mms_protocol.h`

The Message Pool APIs are defined in the following header files:

- `jsr120/src/core/common/native/share/inc/jsr120_sms_pool.h`
- `jsr120/src/core/common/native/share/inc/jsr120_cbs_pool.h`
- `jsr205/src/core/common/native/share/inc/jsr205_mms_pool.h`

An implementation of the pool APIs, which cache messages in memory, is provided and can be used on any platform. You can find this implementation in `jsr120/src/core/common/native/ram_pool`.

If your implementation requires persistent storage, then you must re-implement these pool APIs.

Before porting issues are discussed, clarification of the following functional areas is warranted:

- Internal and external push
- The relationship between the native inbox and the message pool
- Asynchronous message transfers

These relationships are discussed in the following sections.

## Internal and External Push

Push notifications can be a part of the core VM task or a part of an external application. Typically, when an external application monitors network connections, a native interface is used to invoke the Java platform AMS implementation. When the Java platform AMS is used, monitoring for network connections is internal to the same task, allowing for easier sharing of cached inbound messages and for rapid push dispatching.

## Inbox and Message Pool

The platform software presumes that all external networking operations have taken place, including all handshakes with the SMS and MMS service center. The platform software is responsible for segmentation and reassembly (SAR) of messages. When the network is not reachable, full messages are saved persistently in an inbox or queued for delivery in an outbox.

Normally, a pool of messages is cached in memory from the system's persistent storage. When the VM is terminated abnormally, messages in the cache might be discarded and fresh messages retrieved from persistent storage the next time the VM is started.

Under normal usage, messages are delivered from the RAM cache (message pool) to the waiting application. When the message has been dispatched, the buffered message is removed from the pool and a corresponding deletion from persistent storage removes the message from the inbox. A corresponding transaction removes the message from the service center queues.

In the emulated implementation using the datagram transport, no persistent storage is used for message storage. The memory-based message pool serves as both the persistent storage and the cached message pool. Operations are synchronous and only whole messages are exchanged. Message delivery of sent messages is not delayed and no extra handshakes exist for service center protocols. This code is meant only as an emulation for testing purposes. It is below the expected porting layer, and is not intended as code to be ported to WMA networks.

## Asynchronous Message Transfers

Because sending and receiving WMA messages can be time consuming, these operations are asynchronous.

When the `MessageConnection` object's `receive` method is invoked, it checks the message pool for cached messages. If a message is already in the pool, it is retrieved and the message is returned to the application MIDlet. If no messages are in the pool, then the calling Java platform thread blocks, waiting for a message to appear in the pool.

At the same time, the low-level native transport code listens for WMA messages. When a message arrives, it is read, assuming it is a packed formatted byte array.

The format of this byte array is specified in `jsr120_sms_protocol.h`, `jsr120_cbs_protocol.h`, and `jsr205_mms_protocol.h` in the documentation for the `jsr120_send_sms`, `jsr120_notify_incoming_cbs`, and `jsr205_send_mms` methods, respectively. After it is read, depending on the type of message, either the `SmsMessage`, `CbsMessage`, or `MmsMessage`, data structure is populated and added to the message pool.

See `jsr120/src/core/common/native/share/inc/jsr120_sms_structs.h`, `jsr120_cbs_structs.h`, and `jsr205/src/core/common/native/share/inc/jsr205_mms_structs.h` for definitions of `SmsMessage`, `CbsMessage`, and `MmsMessage`. Once the message is added to the pool, an arrival notifier method is called, which unblocks the previously blocked Java platform thread. Once unblocked, the message in the pool is retrieved, returned to the application MIDlet, and deleted from the pool.

When sending a message, the Java platform layer drills into the native transport layer and calls the appropriate send method (`jsr120_send_sms`, `jsr120_send_cbs`, or `jsr205_send_mms`) and blocks. The low-level transport code delivers the message to the network and then invokes a callback function that unblocks the Java platform thread. Note that the Java platform thread is unblocked when the message is delivered to the network, not when it is received by the callee.

Calls to platform-dependent low-level transport code are made from methods such as `Java_com_sun_midp_io_j2me_sms_Protocol_send0()` and `Java_com_sun_midp_io_j2me_sms_Protocol_receive0()` that are contained in `jsr120/src/protocol/sms/native/share/smsProtocol.c`.

Similar methods exist for CBS and MMS.

The platform-dependent methods are expected to return one of the following status types:

- `WMA_OK`
- `WMA_ERR`
- `WMA_NET_WOULDLOCK`

Networking methods that deliver or receive messages synchronously return either `WMA_OK` or `WMA_ERR` as is currently implemented on the Linux, QTE, x86 platform. On some platforms that use “master mode,” the networking methods return quickly and might return `WMA_NET_WOULDLOCK`. In this case, the calling Java platform thread must be blocked and awakened once the operation is completed.

---

## Porting Notes

When porting to a new platform, the Message Transport APIs must be ported using the platform’s native SMS, CBS, and MMS support. For the Message Pool APIs, Java Wireless Client software’s RAM implementation works on all platforms. If an implementation based on persistent storage or native message storage support is desired, then these APIs must also be ported.

For a detailed description of porting API data types and functions see the Native API Reference. From the top page, choose **Modules**, then **JSR205 Wireless Message API (WMA) 2.0**. The API documentation for both JSR 120 (SMS and CBS) and JSR 205 (MMS) is located there.

## Message Porting APIs

The Message Porting APIs include methods for the following functions:

- Receive and select messages based on an ID.

In the case of SMS, a port number can be registered, after which the platform continues to listen for incoming SMS messages that match the registered port number. For CBS a message ID can be registered, after which only those messages that match the message ID are received and stored in the pool. Similarly, for MMS, an application ID can be registered.

The following methods can be used:

- `jsr120_add_sms_listening_port()`
- `jsr120_add_cbs_listening_msgID()`
- `jsr205_add_mms_listening_appID()`

Java Wireless Client software provides a generic implementation for registering IDs using simple C data structures and lists. For an example, see the following file:

```
jsr120/src/core/common/native/share/jsr120_sms_listeners.c
```

This implementation works on all platforms. If platform support is available for distinguishing messages based on port numbers or message and application IDs, it can be used to implement these methods.

- Send an SMS or MMS message.

Two methods are available, one for sending the message and the other a callback function that is called once the message has been delivered to the network. To avoid blocking the VM for a long time, messages are sent asynchronously. The send methods transfer the header information and message buffer to the native platform, and return quickly. The native platform is responsible for handling the entire communication session and uses an asynchronous message or the previously mentioned callback function to notify the implementation layer of the result. The following methods can be used:

- `jsr120_send_sms()`
- `jsr120_notify_sms_send_completed()`
- `jsr205_send_mms()`
- `jsr205_mms_notify_send_completed()`

Note that if the message is too long, the native platform is responsible for segmenting and sending the message.

- Receive an SMS, CBS, or MMS message.

This involves one method for SMS and CBS and three methods for MMS. All three protocols have a callback function that is called, once a message is ready for retrieval. After a WMA application registers a specified message identifier, be it a port number, message or application ID, it will continue to listen for incoming SMS, CBS, and MMS messages whose identifier matches the registered identifier. Once a message has been added to the pool, the platform software should call this callback function with the stored message. The following methods can be used:

- `jsr120_notify_incoming_sms()`

- `jsr120_notify_incoming_cbs()`
- `jsr205_notify_incoming_mms()`

In addition, the MMS protocol requires you to port the following two additional functions:

- `MMSNotification()`
- `jsr205_fetch_mms()`

When an incoming MMS message arrives in the MMS proxy, the proxy calls the `MMSNotification()` function to send a notification to the MMS client (target device). The notification includes only the MMS header. The MMS client decides if it must fetch the message body of this MMS from the network. If the target device finds that the incoming MMS notification is for MIDP WMA, it invokes the callback function `jsr205_fetch_mms()`, which provides a confirmation to the native platform to go ahead and fetch the message body.

- Computing number of message segments.

The `jsr120_number_of_sms_segments`, `jsr120_number_of_cbs_segments`, and `jsr205_number_of_mms_segments` methods calculate the number of protocol segments needed for sending a message. They do not send the message.

## Message Pool APIs

The Message Pool APIs include methods for the following functions:

- Add a message to the pool.

A message in the pool is always associated with an identifier, be it a port number, message, or application ID. Invoke the following methods once a message is successfully added to the pool:

- `jsr120_sms_message_arrival_notifier()`
- `jsr120_cbs_message_arrival_notifier()`
- `jsr205_mms_message_arrival_notifier()`

These notifiers unblock any blocked Java platform threads.

- Retrieve a message with an identifier that matches a specified identifier.

Once the message is retrieved, delete it from the pool.

- Delete a specified message.
- Peek at the pool.

Return a message with an identifier that matches a specified identifier, but do not delete it from the pool.

The included RAM implementation does not provide persistent storage of messages. If you need a persistent implementation, use the APIs in `pcsl_file.h` and `pcsl_directory.h`.

## Interaction With the Push Subsystem

The WMA Push Registry subsystem manages the registration and unregistration of connections used with pushed transactions. The same code that registers a port number, message, or application ID for an application MIDlet also registers and unregisters message identifiers for the Push subsystem. See the API documentation for the following files:

- `jsr120/src/core/common/native/share/jsr120_sms_listeners.c`
- `jsr120/src/core/common/native/share/jsr120_cbs_listeners.c`
- `jsr205/src/core/common/native/share/jsr205_mms_listeners.c`

The registration returns a unique identifier. At startup, the registered connections are opened and the system listens for inbound requests.

When a connection request is detected, the inbound WMA message is read. The Push subsystem is searched for a filter and if one is found, the sender's phone number for SMS or the value of the `replyToAppID` variable is checked against the filter. If the filter check passes, the message is added to the pool.

When the MIDlet is launched, it performs the normal input and output operations to receive inbound messages. The `listConnections` method in the `PushRegistryImpl` class returns the registered connection string. This URI can be used with the `Connector.open()` method to open the connection. The first message retrieved comes from the cached message that caused the push launch to be triggered.

## WMA Push Connections

Native code extends the existing push implementation for WMA. This code resides in the file `jsr120/src/core/common/native/share/wmaPushRegistry.c`.

The push code in MIDP branches appropriately into the WMA-specific handling code. An example of this push code can be found in the file `midp/src/push/push_server/reference/native/push_server.c`.

The WMA push shared code shipped with Java Wireless Client software contains the following features.

- The push entry data structure is extended.

The push entry data structure in `push_server.c` includes WMA-specific information, such as `appID`.

- `push_server.c` is modified to accommodate WMA.
  - For initialization and finalization, the method `pushProcessPort` in `push_server.c` is modified to recognize the WMA connection.
  - To register WMA, the following methods are modified:
    - `pushDeleteEntry` – Closes the WMA push entry
    - `pushfindfd` – Returns the WMA entry
  - To hand off connections, the following method is modified:
    - `pushcheckout` – Dispatches open WMA connections

### *Porting the WMA Push Subsystem*

To port the WMA subsystem, the only required action is to modify the code that monitors inbound connections and reads incoming messages so that it checks the push filter and updates the push entry data structure if necessary. For an example, see `jsr120/src/core/common/native/linux_qte/wmaSocket.cpp`.

Calls are made to `pushsetcachedflag()` and `pushgetfilter()` after a message is received.

## Interactions With the Permission Management Subsystem

To support the wireless messaging subsystem, MIDlets must have permission to perform the following tasks:

- Open a connection
- Send messages
- Receive messages using the messaging API

[TABLE 2-1](#) indicates the permissions that the Wireless Messaging API Specification defines and that the permission management subsystem must manage.

**TABLE 2-1** Wireless Messaging API Permissions for Opening Connections

Permission Name	Functionality and Protocol
<code>javax.microedition.io.Connector.sms</code>	Opens a connection for GSM SMS and CDMA SMS.
<code>javax.wireless.messaging.sms.send</code>	Sends a GSM SMS or CDMA SMS message.
<code>javax.wireless.messaging.sms.receive</code>	Receives a GSM SMS or CDMA SMS message.

**TABLE 2-1** Wireless Messaging API Permissions for Opening Connections

<b>Permission Name</b>	<b>Functionality and Protocol</b>
<code>javax.microedition.io.Connector.cbs</code>	Opens a connection for GSM CBS.
<code>javax.wireless.messaging.cbs.receive</code>	Receives a GSM CBS message.
<code>javax.microedition.io.Connector.mms</code>	Opens a connection for MMS.
<code>javax.wireless.messaging.sms.send</code>	Sends a MMS message.
<code>javax.wireless.messaging.mms.receive</code>	Receives MMS message.

## Porting the Mobile Media API

---

The JSR 135 implementation in Java Wireless Client software is built using two components that are not part of the source code:

- QSound library `mQ_JSR234.lib`
- AMR decoder

To port the JSR 135 implementation to your own platform, you must supply these components or port to suitable replacements.

In addition, the JSR 135 implementation builds only on Windows, only using the JavaCall API.

---

## Capabilities

The JSR 135 implementation in Java Wireless Client software has the following capabilities:

The following formats, protocols and the corresponding JSR-135 controls are supported:

**TABLE 3-1** MMAPI Supported Features and Attributes

Feature	Protocols	Controls	Supported Formats
Sampled audio (WAV)	HTTP	• VolumeControl	• PCM 16-bit stereo • AMR narrow-band
	File	• StopTimeControl	
	InputStream		
Audio capture	capture://audio	• RecordControl	
Interactive MIDI	device://midi	• VolumeControl	
		• MIDIControl	
GIF (includes animated GIF)	HTTP	• FramePositioningControl	
	File	• GUIControl	
	InputStream	• RateControl	
		• StopTimeControl • VideoControl	
MIDI	HTTP	• MIDIControl	• MIDI type 0 • MIDI type 1 • SP-MIDI
	File	• PitchControl	
	InputStream	• RateControl	
		• StopTimeControl	
		• TempoControl	
		• VolumeControl	
Tone sequence	HTTP	• StopTimeControl	
	File	• ToneControl	
	InputStream	• VolumeControl	
Interactive tone sequence	device://tone	• StopTimeControl	
		• ToneControl	
		• VolumeControl	

Note that the JSR 135 capabilities in this release are narrower than in the 1.1.3 release.

---

# Porting Information

To build a JSR 135 implementation for your device, you must implement the JavaCall API functions that are related to multimedia. These are defined in the header `javacall_multimedia.h`, which is in `javacall/interface/jsr135_mmapi`.

See the JavaCall API documentation for details.

Currently the GIF player is implemented entirely in Java platform code and, consequently, does not need porting.

The JSR 135 implementation included with the Java Wireless Client software is an example. Any part of it can be used for your own implementation.

You can examine the Windows JavaCall API implementation in `javacall/implementation/win32/jsr135_mmapi`. This implementation is based on the QSound engine, the AMR decoder, and Windows DirectSound.



# Porting the SATSA Optional Package

---

This appendix discusses the design and porting of the Security and Trust Services API (SATSA) optional package (JSR 177) for the Java ME platform. SATSA is a collection of APIs that provide security and trust services through integration of a security element (SE). For more information about JSR 177, see <http://www.jcp.org/en/jsr/detail?id=177>.

---

## SATSA Code Structure

The SATSA file system is organized as shown in [FIGURE 4-1](#) and [FIGURE 4-2](#). Folders shown in bold in those figures represent the four main parts of the SATSA package and are described in [TABLE 4-1](#):

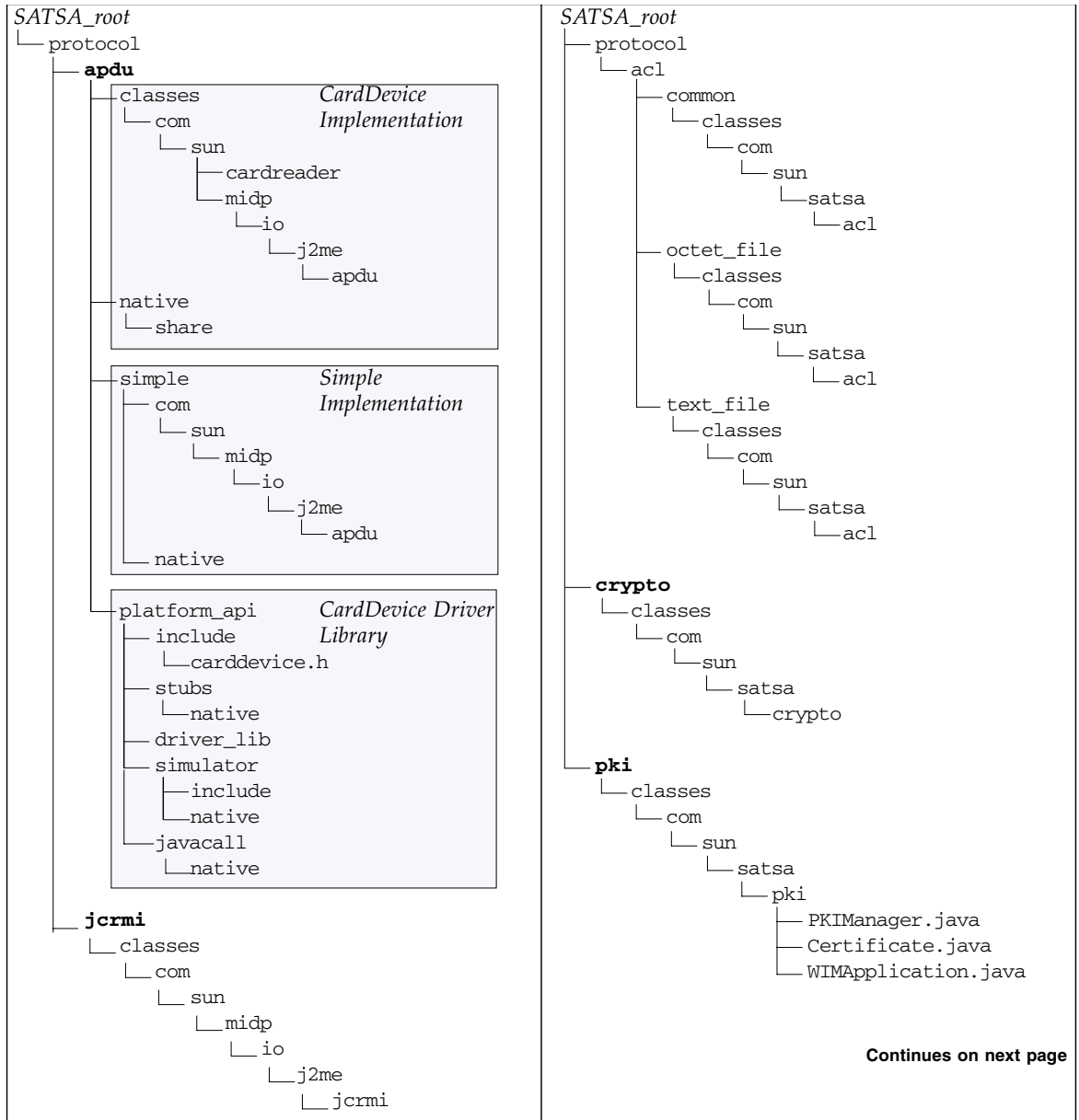
**TABLE 4-1** Four Main Parts of the SATSA Package

<b>Part</b>	<b>Description</b>
APDU	Supports data exchange with smart cards. Used by other parts of the system to access security elements. Porting is required for any type of security element.

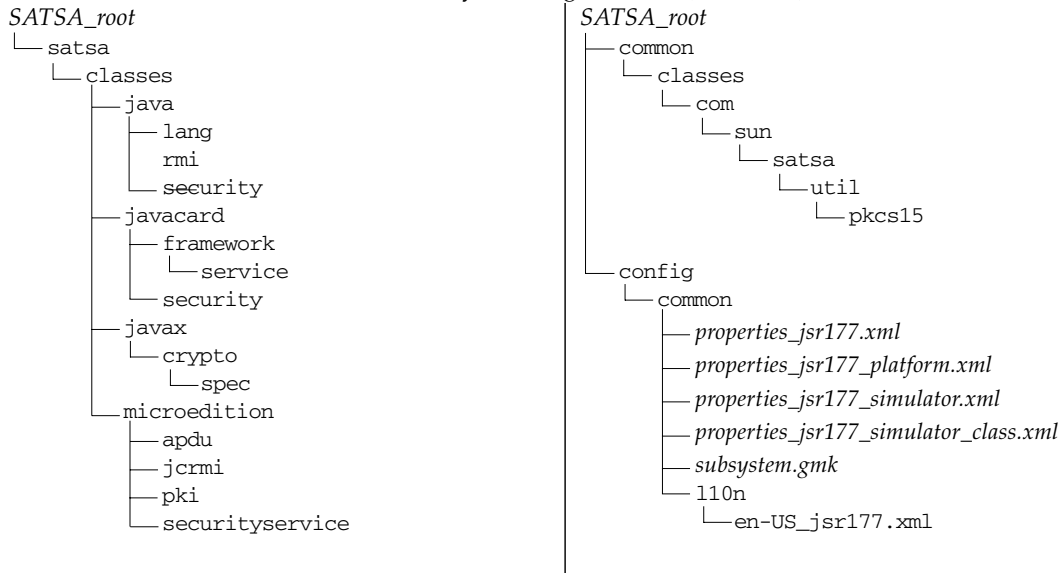
**TABLE 4-1** Four Main Parts of the SATSA Package

<b>Part</b>	<b>Description</b>
Java Card RMI	Provides RMI support for on-card applets. Can be ported without modification.
PKI	Supports the public key infrastructure and wireless identity module. Can be ported without modification unless the security element is not compatible with WIM, or WIM does not use a smart card.
CRYPTO	Provides interfaces for cryptographic algorithms for encoding and decoding data and generating and verifying signatures. Does not require contact with the platform. Extra porting effort might be necessary if additional algorithms are required or if some algorithms are implemented in the platform.

**FIGURE 4-1** SATSA File System Organization



**FIGURE 4-2** SATSA File System Organization (*Continued*)



## Security Elements

The SATSA specification introduces the SE as a “device” that holds the user’s credentials and helps him to cipher and sign messages. This implementation assumes that the SE is compliant with the ISO 7816 standard which defines the SE to be a smart card. If your platform uses an SE that does not comply with ISO 7816, you must rewrite some of the classes in the APDU and PKI packages and the ACL file implementation.

You can access the SE by two means:

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

This implementation (and the SATSA specification) assumes that only application oriented access is used. This means that to access the SE, the appropriate on-card application must be selected.

According to the PKCS-15 specification and appendix A of the SATSA specification, the implementation can read an access control file from an SE which can publish it. This is the only case where the implementation uses the file access option.

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

---

## SATSA-APDU Package

Every SE is assumed to be inserted in a slot. Slots belong to devices (card readers). The platform can support many devices and a device can contain a number of slots. The Java platform layer sees only one device: the platform that operates with (logical) slots. Each slot has a number (index). The current slot must be selected before any data-oriented operation is performed.

The porting layer has to provide transactional access to a slot. None of the operations (except `init` and `finalize`) can be performed if the slot is not locked. When the slot is locked, it cannot be locked again. All error information is cleared after an unlock operation. Withdrawal does not perform an unlock operation. Switching the slot number between transactional operations is not allowed.

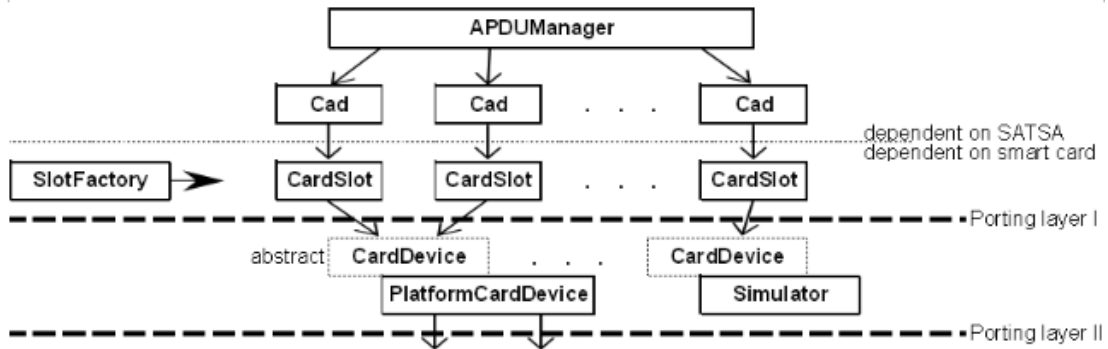
The SATSA-APDU package has two implementations:

- **CardDevice** – Can support any number of devices. It provides different interfaces for card device drivers developed in both the C and Java programming languages. You can use both types of drivers simultaneously. This implementation is comprehensive and contains a number of classes and methods.
- **Simple** – Can support only one device and only provides a C interface. It is simple because it contains only a few classes.

## CardDevice Implementation

FIGURE 4-3 shows the SATSA-APDU package from the porting point of view.

FIGURE 4-3 SATSA-APDU Porting Interfaces



The `CardDevice` implementation has two potential porting layers. The `APDUManager` class holds the table of `Cad` objects – one for each card slot. The `Cad` class implements all functionality needed by the SATSA specification, including re-sending commands and getting the response in accordance with status words. The initialization procedure uses the `SlotFactory` class for creating `CardSlot` objects. These are responsible for low-level functionality, for example, locking, sending commands, and resetting.

Usually only one of the two layers is required. In most cases this is Porting Layer II, a native porting layer that must be supported by the target platform.

Porting Layer I is intended for Java platform implementations. For example, the `Simulator` class uses the CREF emulator as Security Element (the CREF emulator is a part of Java Card Development Kit 2.2)

## Porting Layer I

This porting layer is implemented in the Java programming language. The abstract `CardDevice` class represents any device that can operate a smart card (card reader). If you use a real device, derive a class from `CardDevice`.

Any device has to know how many card slots it serves. During the initialization process, the device class must report this number to the `SlotFactory`.

The `SlotFactory` class uses the values assigned to the properties `com.sun.cardreader.deviceClassN` and `com.sun.cardreader.deviceNumber` to determine which classes must be used as `CardDevice` classes.

The card device class must support slot locking. It must prevent access from native applications and other tasks between the lock and unlock sequence. The `APDUManager` class is responsible for blocking access from Java programming language threads that are working in this task.

## ▼ Implementing Porting Layer I

The following steps describe the process required to implement Porting Layer I.

**1. Derive new classes from the `CardDevice` class.**

Each card device might require its own derived class.

**2. Implement all abstract methods from the `CardDevice` class.**

Use the `Simulator` class as an example.

**3. Define the `com.sun.cardreader.deviceClassN` and `com.sun.cardreader.deviceNumber` properties.**

The file `properties_jsr177_simulator_class.xml` (one of the italic entries in [FIGURE 4-1](#)) contains the properties that describe card devices. These properties are listed in [TABLE 4-2](#).

**TABLE 4-2** Properties That Describe the Platform’s Card Devices

Name	Type	Description
<code>com.sun.cardreader.deviceNumber</code>	Number	The total number of devices configured in the system.
<code>com.sun.cardreader.deviceClassN</code>	String	The class name for device number <i>N</i> . The total number of these properties equals the number defined by the <code>.deviceNumber</code> property.

You can create a new properties file with your class using `properties_jsr177_simulator_class.xml` as an example. If you do, remember to update the makefile (`subsystem.gmk`) accordingly.

## Porting Layer II

This porting layer supports the platform’s native API and the `PlatformCardDevice` class is used in place of Porting Layer I. The KNI layer converts native method invocations to native API calls. For a detailed description of porting API data types and functions see the Native API Reference ([html/group\\_satsa.html](http://html/group_satsa.html)).

The correct sequence (lock → select slot → operations → unlock) is supported by the Java platform and the KNI layer and is not managed by the porting layer. Access to the locked slot by native applications must be blocked by the porting layer.

## ▼ Implementing Porting Layer II

1. **Define the `com.sun.cardreader.deviceClass1` and `com.sun.cardreader.deviceNumber` properties in `properties_jsr177_platform.xml`.**

On the target device platform, define the property `com.sun.cardreader.deviceClass1` to be `com.sun.cardreader.PlatformCardDevice` and the property `com.sun.cardreader.deviceNumber` to be 1.

2. **Build a card device driver.**

Place the porting layer code (the device driver) in a library. Use an ordinary C-compatible static-linking library with a name like `libXXXX.a`, where `XXXX` is the name of the device (for example, `gemplus`)

3. **Place the card device driver into the workspace.**

Create a directory with an appropriate name. The path to the driver's library must be:

```
SATSA-root/src/protocol/apdu/platform_api/driver_lib/  
SATSA-root is the root of the SATSA workspace.
```

4. **Define card device driver.**

Define the environment variable `JSR_177_APDU_PLATFORM_API` to be the name of device (for example, `gemplus`) before building the SATSA workspace.

## Porting with the JavaCall API

If you are using the JavaCall API, set the build options as follows:

```
JSR_177_APDU_MANAGER=carddevice  
JSR_177_APDU_CARDDEVICE=platformcarddevice  
JSR_177_APDU_PLATFORM_API=javacall
```

or

```
JSR_177_APDU_MANAGER=simple  
JSR_177_APDU_CARDDEVICE (doesn't matter)  
JSR_177_APDU_PLATFORM_API=javacall
```

The JavaCall API for `CardDevice` consists of five functional groups:

1. Initialization and finalization

2. Data exchange
3. Locking
4. Retrieving information
5. Error handling

All five functional groups are mandatory.

## Initialization and Finalization

- `javacall_carddevice_init()` must initialize the card device
- `javacall_carddevice_finalize()` must finalize the card device. If no special finalization actions are required, it may do nothing.
- `javacall_carddevice_set_property()` is used for setting properties. If a JavaCall API implementation doesn't support given property it must return `JAVACALL_NOT_IMPLEMENTED`.
- `javacall_carddevice_select_slot()` must select "current" slot for use. If a card device has only one slot this method may do nothing. This method will be called only when the card device is locked (see the locking group).

## Data Exchange

There are two operations with a smart card that require data exchange: "reset" and "transfer data". The former means "do power up and receive the answer-to-reset (ATR)". The latter is used for sending and receiving data (APDU) to/from a card.

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

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

## Locking

Before any data exchange operation is performed a card device must be locked.

The SATSA implementation uses following scenario:

```
javacall_carddevice_lock
```

```

      |
      v
javacall_carddevice_select_slot
      |
      v
any card operation
      |
      v
javacall_carddevice_unlock

```

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

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

- `javacall_carddevice_lock()` must lock the card device
- `javacall_carddevice_unlock()` must unlock the card device

## Retrieving Information

- `javacall_carddevice_get_slot_count()` must return number of device slots. This method will be called once at initialization time.
- `javacall_carddevice_is_sat()` must return a boolean value: is this slot a SAT slot? This method will be called after any card insertion.
- `javacall_carddevice_card_movement_events()` must report which insertion/withdrawals were performed with a card. This method will be called before and after any card operation.

## Error Handling

The SATSA implementation will send a text message into the error handling system if it supposes that a trouble occurred. In that case an error state must be set. After that the SATSA implementation must be able retrieve all (if allowed by memory limitations) messages concatenated to one string, separated by the new line character. The error state must be clear after that.

- `javacall_carddevice_clear_error()` must clear the error state.
- `javacall_carddevice_set_error()` must receive a message and set the error state.
- `javacall_carddevice_get_error()` must return the concatenated messages and clear the error state.

# Simple Implementation of the APDU Package

Java Wireless Client software includes a sample implementation of the APDU package called Simple. The Simple implementation uses Porting Layer II and contains the following four classes.

- Protocol
- APDUManager
- Handle
- Slot

All the classes are located in the following directory:

```
SATSA_root/src/protocol/apdu/simple/classes/com/sun/  
midp/io/j2me/apdu
```

The Simple implementation uses the same Protocol class as does the CardDevice implementation. APDUManager is a static class with native methods. The Handle and Slot classes only contain data that corresponds to a connection handle and to a device slot.

All the native methods of the APDUManager class are located in

```
SATSA_root/src/protocol/apdu/simple/native.
```

In comparison to the CardDevice implementation, the Simple implementation has the following features:

- It is faster because it does not call virtual methods.
- It is smaller by about eight kilobytes.
- It supports the asynchronous mode and includes an example that uses native pthreads.
- It supports only one device.
- Its device driver cannot be written in the Java programming language.
- It has a limited buffer for incoming data. The buffer is defined in the Slot class, and you can enlarge it.
- It requires different implementations for different platforms.

## Platform API

The Platform API directory (*SATSA\_root/src/protocol/apdu/platform\_api*) contains the following different implementations of Porting Layer II.

- **stubs** – Stub files that allow the system to build correctly but do not do anything.
- **simulator** – A complete port of the Simulator class. This implementation uses the CREF utility as a Security Element.

- **javacall** – A JavaCall API port. This contains wrappers for JavaCall API functions. See details in the the JavaCall API documentation.
- **driver\_lib** – A location in which to put the library that contains the card device driver.

See “Porting Layer II” on page 23 for details about Porting Layer II.

## Build Options

Specify your implementation by setting build options. These options are set using environment variables prior to building the product. These options are used in `subsystem.gmk`.

The build options are listed and described in TABLE 4-3.

**TABLE 4-3** APDU Build Options

Option	Description
JSR_177_APDU_MANAGER	Selects the CardDevice or Simple implementation of the APDU package.
JSR_177_APDU_CARDDEVICE	Specifies the implementation of Porting Layer I. Possible values are <code>platformcarddevice</code> or <code>simulator</code> . This option is required only if the value of JSR_177_APDU_MANAGER is <code>carddevice</code> .
JSR_177_APDU_PLATFORM_API	Specifies the implementation of Porting Layer II. You can specify <code>stubs</code> , <code>simulator</code> , <code>javacall</code> , or any other library name (for details see “Porting Layer II” on page 23).

## SAT Connection

SAT is the (U)SIM Application Toolkit, an environment for applications that comply with the SAT specification. This environment is different from the Java Card technology runtime environment. When a Java ME technology application wants to communicate with the SAT, it must not select an on-card application. The application invokes events (by sending the `ENVELOPE` command) that are handled by the on-card SAT environment and then may be sent to registered on-card applications.

The implementation has not been tested against actual SAT applications. For testing purposes an emulator can be found in the SATSA workspace at the following locations:

- `SATSA_root/src/tool/javacard/com/sun/satsa/satapplet`

- `SATSA_root/src/tool/javacard/com/sun/satsa/gsmapplet`
- `SATSA_root/src/tool/javacard/sim`

The SIM emulator is an ordinary Java Card technology applet. Before using it select an on-card application. The property `com.sun.midp.io.j2me.apdu.satselectapdu` contains the APDU command to select the proper application. If the property is not defined, then no selection is made. This property is defined in the file `properties_jsr177.xml`.

---

## SATSA-CRYPTO Package

The SATSA-CRYPTO optional package defines a subset of the Java SE platform cryptography API. It provides basic cryptographic operations to support the following functionality:

- Message digest creation
- Signature verification
- Encryption using an asymmetric or symmetric cipher
- Decryption using a symmetric cipher

According to the specification, the package does not include an API to create a private key object. Because you cannot create an asymmetric cipher using a private key, an asymmetric cipher can only be used for signature verification and encryption.

## Implementation

SATSA-CRYPTO is an optional package. It is implemented independently from other SATSA packages and does not use other SATSA packages APIs.

Implementation of CRYPTO algorithms is based on the functionality provided by the Java Wireless Client software `security` subsystem's `crypto` library.

TABLE 4-4 summarizes the algorithms recommended by the JSR 177 specification and the algorithms that are implemented in this release.

TABLE 4-4 Algorithm Implementations

JSR 177 Specification	Java Wireless Client Software Implementation
<b>Message Digest</b>	
SHA-1	SHA-1 MD2 MD5
<b>Digital Signature (Verification Only)</b>	
SHA1withRSA	SHA1withRSA MD5withRSA
<b>Asymmetric Cipher (Encryption Only)</b>	
RSA	RSA, which by default is RSA/NONE/PKCS1Padding
<b>Symmetric Stream Cipher</b>	
	ARCFOUR (interoperable with RC4), which by default is ARCFOUR/NONE/NoPadding
<b>Symmetric Block Cipher</b>	
<b>Algorithm</b>	
DES	DES
DESede	DESede
AES	AES
<b>Mode</b>	
ECB	ECB (default)
CBC	CBC
<b>Padding</b>	
NoPadding	NoPadding
PKCS5Padding	PKCS5Padding (default)

## Porting Notes

The SATSA-CRYPTO package is implemented entirely in the Java programming language. It has no native code. No external or platform-specific resources are utilized. The only external dependency is on the SSL package that is created using

the Java programming language and platform-independent native code. For that reason, the SATSA-CRYPTO package implementation does not have a porting layer or API.

If the cryptographic algorithms are implemented in hardware on your platform, you must create a porting layer and a native API.

---

## SATSA-PKI package

This package provides functionality for user credential management that includes creating certificate signing requests, adding user credentials, and removing credentials. The package also provides an API for digital signature generation based on stored credentials.

### Implementation

This implementation of the PKI package is based on Appendix D, “WIM Recommended Practice,” of the SATSA specification. WIM (Wireless Identity Module) is a PKI-based identity module. It is used for storing keys and certificates and for performing cryptographic operations with these keys. The WIM can be implemented in smart cards, such as (U)SIM cards, or other tamper-resistant hardware. The current version of credential management supports X.509 version 3 certificates. This implementation does not support URIs. The implementation based on the WIM specification is: WAP-260-WIM-20010712-a, Version 12-July-2001.

This implementation assumes the use of an on-card applet that provides WIM functionality. That applet (WIM emulator) is contained in the SATSA workspace. The path to the emulator is `SATSA-root/src/tool/javacard/com/sun/satsa/pkiapplet`.

The implementation introduces a WIM-compatible command named `Generate Public Key Pair`. This command is not specified in the WIM specification because the specification anticipates that key pairs are generated as a part of the personalization process. Detailed information about this command can be found in [TABLE 4-5](#) and [TABLE 4-6](#). The property `com.sun.satsa.keygen` can disable this key generation feature. The value `true` enables key generation, and `false` disables it. This property is defined in the file `properties_jsr177.xml`.

**TABLE 4-5** Generate Public Key APDU Command - Check Key Support

Function	Field	Value (hex)
Command	CLA	80
	INS	BC
	P1	01
	P2	00
	Lc	03
	Data	<ul style="list-style-type: none"> <li>• One-byte flag: 00 if authentication key, 01 otherwise</li> <li>• Length of the key as a two-byte short</li> </ul>
	Le	F0
Response if requested key can be generated	Data	12 34 43 21
	SW	90 00
Response if requested key can not be generated	Data	Empty
	SW	90 01
Response if an error occurs	Any other result	

**TABLE 4-6** Generate Public Key APDU Command - Generate Key Pair

Function	Field	Value (hex)
Command	CLA	80
	INS	BC
	P1	00
	P2	00
	Lc	2B
	Data	<ul style="list-style-type: none"> <li>• One-byte flag: 00 if authentication key, 01 otherwise</li> <li>• Length of the key as a two-byte short</li> <li>• Eight bytes for the new PIN value (padded with hex FF bytes)</li> <li>• 20 (hex) bytes for the PIN label (padded with spaces)</li> </ul>
	Le	

**TABLE 4-6** Generate Public Key APDU Command - Generate Key Pair (*Continued*)

Function	Field	Value (hex)
	Le	F0
Response if requested key has been generated	Data	One-byte reference of the new key
	SW	90 00
Response if an error occurred	Any other result	

The implementation saves the public key ID (10-byte hash) in the local file storage as `_csr.id` when the key has already been used for certificate enrollment request (CER) generation. The key IDs must be stored until a new certificate is received. Saving is required to conform to the SATSA specification requirement that an implementation must prioritize keys during CER generation according to the following parameters:

- Unused keys
- Keys used in CERs
- Keys used in certificates

The property `com.sun.satsa.store_csr_list` is used to manage the save process. The value `true` enables saving, `false` (or no value) disables it. This property is placed in the file `properties_jsr177.xml`. When key IDs are not saved in storage, they are placed in a memory array.

The PKI part of the implementation is placed in the directory `SATSA_root/src/classes/com/sun/satsa/pki`.

## Porting Notes

The `PKIManager` class provides a basic interface to the off-card WIM application. The `PKIManager` class uses the `WIMApplication` class, which provides an interface to the on-card WIM application. If the target device already has mechanisms in place to talk to an on-card WIM application, this class must be replaced with those mechanisms and the `PKIManager` class must be updated accordingly. Only two classes (`PKIManager` and `Certificate`) do not depend on an on-card WIM application. The others are helpers for the `WIMApplication` class.

If the target device does not have a mechanism in place to talk to the WIM, the implementation can be ported without modification. Note that this implementation has not been tested with an actual WIM, therefore it requires some modification to talk to an actual WIM.

The `Dialog` class is used to create dialog boxes that do the following:

- Display messages
- Prompt the user for confirmation
- Prompt the user to sign
- Prompt the user to choose a certificate
- Prompt the user to enter new PIN parameters

---

**Note** – The implementation of this class was moved from the reference implementation without modification.

---

According to the SATSA specification, this dialog box must be secure and must include security features that cannot be duplicated by any regular Java ME platform application. Therefore, this security requirement cannot be satisfied by the `Dialog` class in the implementation. On the target platform, this class can be rewritten so that it provides the same basic functionality of the `Dialog` class. To ensure security, the class must also have one or more unique features that are distinguishable from a user interface generated by external sources such as Java ME platform applications.

---

## Static Access Control Mechanism

The SE is able to publish its access control requirements in an access control file (ACF). The ACF is created by the issuer of the application or the SE. The ACF contains an access control list (ACL).

The implementation of the static access control mechanism complies with Appendix A, “Recommended Security Element Access Control,” of the SATSA specification.

The path to an implementation of the ACL is `SATSA_root/src/protocol/acl/`. Two different versions of the implementation are possible. One is based on text files (from the RI) and the other is mandated by the SATSA specification and is based on octet files.

### ▼ Looking for an Access Control File

The implementation can retrieve an ACF from security element that satisfies the ISO7816-4 standard. The SATSA and the PKCS-15 specifications partially explain how to find an ACF stored on a card. The implementation performs the following sequence of steps to do that.

---

**Note** – Acronyms and terms in italics are defined following the sequence of steps.

---

**1. Look for the DIR file.**

The implementation *selects* MF and then tries to *select* EF with file ID=2F00.

**2. If the DIR file is not found (Step 1 fails) try to *select* the PKCS-15 application using its AID.**

**3. If neither the DIR file or the PKCS-15 application are found (Step 1 and Step 2 fail) decide that *all permissions must be granted*.**

The sequence is complete.

**4. If the application was successfully selected in Step 2, go to Step 9.**

**5. *Read* the DIR file and try to find the PKCS-15 application's AID in it.**

**6. If the required AID is not found, then decide that *all permissions must be granted*.**

The sequence is complete.

**7. *Select* the DF that corresponds with first DIR entry where the PKCS-15 application's AID is found.**

**8. If Step 7 fails, then decide that *all permissions must be revoked*.**

The sequence is complete.

**9. Look for the ODF.**

The implementation tries to *select* EF with the file ID retrieved from the DIR file or, if it is not specified, with the default file ID=5031.

**10. If Step 9 fails, decide that *all permissions must be revoked*.**

The sequence is complete.

**11. *Read* the ODF.**

**12. Repeat Step 13 through Step 16 for each DODF entry found in the ODF, then go to Step 17.**

**13. *Select* the DODF and then *read* it.**

**14. If Step 13 fails, go to the next iteration of the loop defined in Step 12.**

**15. Search through the DODF for an entry with the OID as defined in subsection A.4.2.1 of the SATSA specification.**

16. **If the required entry is found, parse the access control index file and load permissions as specified in the Appendix A of the SATSA specification.**

The sequence is complete.

17. **Decide that *all permissions must be revoked*.**

The sequence is complete.

All errors that occur during a given sequence of operations (for example, I/O errors, invalid file structures, broken card problems, and missing card features) lead to the revocation of all permissions. In general, the implementation allows access if it reads a correct PKCS-15 structure (with or without an ACF) and denies access when it cannot verify the existence of an ACF.

## Definitions

TABLE 4-7 defines the terms shown in italics in the previous sequence of steps.

**TABLE 4-7** Terms Used in Previous Steps

Term	Meaning
<i>select</i>	Run the <code>SELECT FILE</code> command as defined by the ISO7816-4 standard. If this verb is applied to a file (MF, ODF, DODF, or DIR), select using the file ID. If this verb is applied to an application, select using the name (by AID, as stated in the ISO7816 standard).
<i>read</i>	Run the <code>READ BINARY</code> command as defined by the ISO7816-4 standard.
<i>all permissions must be granted</i>	<ul style="list-style-type: none"> <li>• Do not apply the static mechanism for evaluating permissions</li> <li>• Do not deny any ADPU command (except <code>SELECT FILE</code> or <code>MANAGE CHANNEL</code>)</li> <li>• Do not deny invocation of all Java Card RMI methods</li> <li>• Do not use the APDU command and Java Card RMI methods with PIN related methods</li> </ul> <p>Other access control mechanisms (for example, the domain mechanism) can deny some operations.</p>
<i>all permissions must be revoked</i>	<ul style="list-style-type: none"> <li>• Deny all ADPU commands</li> <li>• Deny invocation of all Java Card RMI methods</li> <li>• Do not use the APDU command and Java Card RMI methods by PIN related methods.</li> </ul> <p>Other access control mechanisms (for example domain mechanism) cannot allow the use of the APDU command or Java Card RMI methods.</p>

TABLE 4-8 defines the acronyms used in the previous steps and shows the specification in which it is specified.

TABLE 4-8 Acronyms Used In Previous Steps

Acronym	Definition	Specification
DIR	Directory file	ISO7816
MF	Master file	ISO7816
DF	Dedicated file	ISO7816
EF	Elementary file	ISO7816
OID	Object identifier	PKCS-15
AID	Application identifier	PKCS-15
ODF	Object directory file	PKCS-15
DODF	Data object directory file	PKCS-15

## Reference Implementation Version of ACF

In the SATSA RI, implementation of the access control mechanism is based on an access control file (ACF), which is a plain-text file located in permanent storage. The ACF is named `acl_N`, where *N* is a slot number. The structure of that ACF is fully described in the RI's porting appendix.

Although Appendix A of the SATSA specification is now mandatory, the RI's version is still supported. To switch between versions, set the variable `USE_JSR_177_ACL_TEXT` in `subsystem.gmk`.

The error handling behavior of the RI version differs from Appendix A of the SATSA specification in the following way: if any error occurs when loading the ACF, *all permissions must be granted*.

## Porting Notes

If the security element is compatible with ISO7816, then the access control mechanism can be ported without modification.

If the SE does not support ISO7816-4 commands, then a new file system class must be derived from the `FileSystemAbstract` class. This new class replaces the `AclFileSystem` class with new methods for accessing files. The `ACS1ot` class must be updated accordingly.

The `PINEntryDialog` class is used to create a dialog that is shown to users when they are required to enter PIN values. The implementation of this class was moved from the RI without modification. According to the SATSA specification, this dialog must be secure and must include security features that cannot be duplicated by any regular Java ME platform application. Therefore, this security requirement cannot be satisfied by the `PINEntryDialog` class in the implementation. On the target platform, this class can be rewritten so that it provides the same basic functionality of the `PINEntryDialog` class. The class must provide one or more unique features that are distinguishable from a user interface generated by external sources such as Java ME platform applications.

## Porting CHAPI

---

This chapter describes how to port the Content Handler API (CHAPI) optional package (JSR 211) for the Java ME platform. The CHAPI optional package permits the enhanced integration of Java ME platform applications into a device's application environment and manages the handling of Uniform Resource Identifiers (URIs) based on a MIMEtype or scheme. The CHAPI optional package provides the capabilities for browsers and native applications as well as Java ME platform applications to invoke other Java ME platform applications that dynamically extend the media types and capabilities supported by the device's application environment. For more information about JSR 211, see

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

---

## CHAPI Code Structure

This document focuses on the porting layer which contains all of the platform dependent functionality. In order to port CHAPI to a new platform, you must implement all the functions in the porting API.

The CHAPI file system is structured as follows.

```
CHAPI_root/src
  classes
  config          - Configuration files used in the CHAPI build process
  core
  jams
    classes
  nams
    classes
  native          - All the native sources and include files
    platform_a   - Porting API implementation for platform "a"
    platform_b   - Porting API implementation for platform "b"
  javacall       - JavaCall API implementation
```

```

share    - Platform independent sources
inc      - Include files that define the CHAPI porting API
stubs    - Stubs for the porting API functions
i3test

```

The files listed in TABLE D-1 are particularly important during porting.

**TABLE 5-1** Important Files Used in Porting

File	Description
src/config/ subsystem.gmk	Rules and variables used to build the CHAPI subsystem. Change this file only if you add, move, rename, or delete files.
src/core/native/share/inc/ jsr211_registry.h	Header file that contains the CHAPI porting API functions and structures. All functions required for porting are declared in this file.
src/core/native/stubs/ jsr211_registry_impl.c	Stubs for the CHAPI porting API function implementations.
src/core/native/reference/ jsr211_registry_impl.c	Examples of the CHAPI porting API functions.
src/core/native/share/ jsr211_nams_installer.c	Implementation of the following functions: <ul style="list-style-type: none"> <li>• jsr211_verify_handlers()</li> <li>• jsr211_store_handlers()</li> <li>• jsr211_remove_handlers()</li> </ul> These functions are invoked when MIDlet suites are installed or removed from the native AMS runtime.
src/core/native/share/ jsr211_deploy.c	Internal content handler data structures and registration implementations are defined in these files. These structures are defined separately for the native AMS (NAMS) and Java platform AMS runtimes. jsr211_check_internal_handlers() is implemented in these files. This function is invoked during CHAPI initialization.

## Porting Layer Functionality

Most CHAPI functionality is implemented at the Java platform level and need not be ported. The content handler registry is the only platform-dependent functionality and must be ported. Native support for the content handler functionality is defined in `javax.microedition.content.Registry` (see the CHAPI specification). The content handler registry is implemented in native code for the following reasons.

- To store content handler description more efficiently. The native system is more efficient for storing the complicated content handler description data model.
- To enable CHAPI to interact with the native application registry. This interaction enables CHAPI to use platform content handlers to process Java ME application content handling requests.
- To enable CHAPI to process the platform's application content handling requests for types supported by the registered content handlers.
- To allow the native AMS to register new content handlers during the installation of new MIDlet suites. Conversely, it allows the native AMS to remove the content handlers that belong to a MIDlet suite that is being uninstalled.

A native registry is responsible for the following functionality:

- Register content handlers and provide the following mandatory attributes:
  - ID
  - Suite ID
  - Class name

The following optional attributes can also be provided:

- Authenticated authority
- MIME types
- Suffixes
- Actions
- Action names for given locales
- Access restrictions

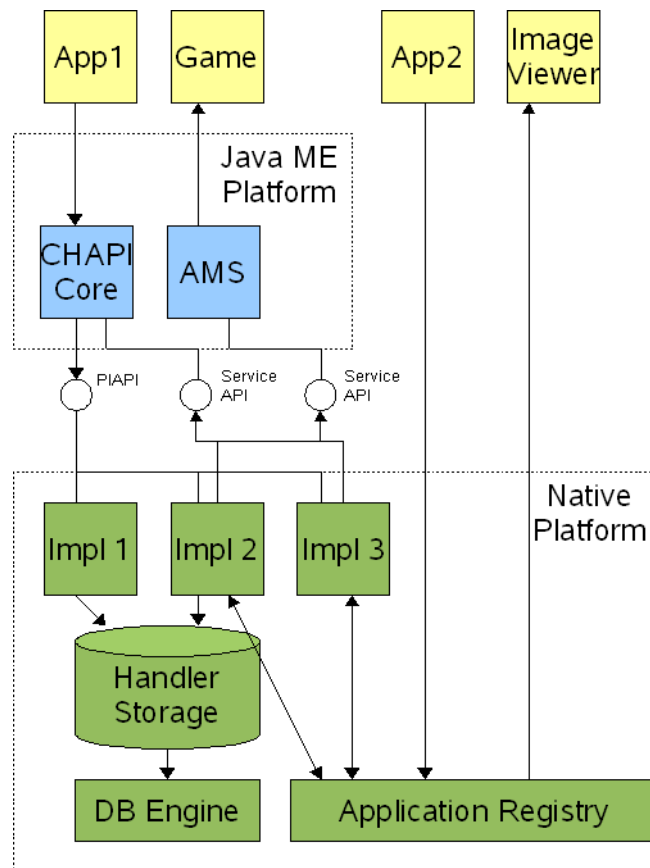
The content handler registration method is also provided. The following registration methods are supported:

- **Static** - The content handler is registered during the installation of a new MIDlet suite. Statically registered handlers can only be removed from the registry when their MIDlet suite is removed.
- **Dynamic** - User applications register the content handler by calling the `register()` method. User applications can remove dynamically registered handlers by calling the `unregister()` method.
- Unregister content handlers based on their IDs.
- Search content handlers by the following:
  - Attribute value
  - Search condition (see the description of enum values in `jsr211_registry.h`)
  - Case sensitivity
- List registered content handlers' values for a specified attribute.
- Load content handler attribute values for a specified content handler ID.

# Porting API Implementation Cases

The features described in the previous section illustrate the need for the close integration of the CHAPI content handler registry and native platform application registry. How the integration is implemented depends on native platform registry features and the API. Some variation can exist in how functionality is distributed between the CHAPI registry and platform registry. Three different implementations are illustrated in the following figure.

**FIGURE 5-1** CHAPI Registry Implementation Scenarios



The figure shows two Java platform applications (App1 and Game) and two native applications (App2 and Image Viewer). App1 uses CHAPI to open images in PNG format, whereas Game is registered as a handler for game level files.

App2 uses the platform registry to install the Java platform game level distributed in the game level file. The native image viewer is registered in the platform registry as the content handler for type PNG.

There are three implementations of the Porting/Internal API (PIAPI) depicted in the figure (Impl1, Impl2, and Impl3). Each interacts with the native platform registry differently, as follows:

- **Impl1** - This implementation represents the scenario in which the target platform does not have an application registry, or its API is not suitable for use with CHAPI. All the data related to registered content handlers descriptions is saved in internal CHAPI storage. This implementation does not allow Java Wireless Client software to use native platform applications because content handlers and native platforms cannot use Java Wireless Client software applications as content handlers. Therefore, in the example in the figure, App1 is unable to open a PNG image using CHAPI and App2 is unable to install the Java platform game level using a game level file.
- **Impl3** - This implementation represents the scenario in which the target platform has a full-featured application registry that can be used to store all the content handler description information. In this case, the implementation of PIAPI functions consists of calls translated to the native registry API, and there is no need to implement internal CHAPI storage. This implementation uses a full-featured platform registry for mapping CHAPI API registry storage requirements. For that reason, this implementation might be the most efficient and provide performance close to a platform-based registry.
- **Impl2** - This implementation represents the most realistic scenario. In this situation a native platform registry API allows CHAPI to access registered native platform content handlers and to be registered as a content handler for types that are supported by the Java platform handlers. There is no possibility to store additional Java platform content handler description data (such as supported actions, locales, action and map). Therefore, this implementation requires the availability of CHAPI internal storage in which to save additional content handler data. This implementation provides support for a full duplex Java platform mechanism for content handling processing.

The Impl2 and Impl3 implementations allow App1 to use CHAPI to execute the native image viewer to open PNG images. App2 is able to install the Java platform game level using its game level file as the content to be handled.

---

## Porting CHAPI

The following steps provide a high-level overview of the tasks required to port the CHAPI optional package to a new device (referred to here as *new-platform*).

1. Create the *new-platform* directory in *CHAPI-root/src/core/native*.
2. Copy the files in *CHAPI-root/src/core/native/stubs* to the *new-platform* directory.
3. Implement the functions in *CHAPI-root/src/core/native/new-platform/jsr211\_registry\_impl.c* as appropriate for your platform.
4. Modify the file list or add new source files, if any, to `INTERNAL_JSR_211_PLATFORM_FILES` property in *CHAPI-root/src/config/subsystem.gmk*.
5. If an internal content handler is implemented (for example, `GraphicalInstaller` in the Java platform AMS runtime of MIDP), insert it as a filled `internal_content_handler` structure into the array `internal_handlers` in *jsr211\_deploy.c*. This data is used by the method `jsr211_check_internal_handlers()` during internal handler registration.
6. If the platform has its own content handler application, it can also be registered as internal content handler. Implement the method `jsr211_execute_handler()` to ensure that the platform handlers are launched properly.

---

## Porting Steps for the JavaCall API

The JavaCall API implementation exists in the CHAPI directory as another platform in *src/core/native*. The actual platform functionality has moved down to the JavaCall layer and currently appears as files in the top-level *javacall* directory.

The JavaCall API CHAPI interface is in *javacall/interface/jsr211\_chapi/javacall\_chapi.h*.

The only implementation provided is for win32. It is located in *javacall/implementation/win32/jsr211\_chapi/chapi.c*.

To create a JavaCall API implementation for a new platform, implement the functions defined in *javacall\_chapi.h*. Store your implementation in *chapi.c* in a directory corresponding to your platform.

Consult the JavaCall API documentation for full details on each function.

## OpenGL ES

---

JSR 239 defines a Java platform API binding for the industry standard OpenGL® ES API. Your job as porting engineer is to bind the JSR 239 API to the underlying OpenGL ES API for your device.

This porting process is the same regardless of whether you are using the JavaCall API or not. The implementation of JSR 239 in Java Wireless Client software bypasses the JavaCall layer and binds directly to your device's OpenGL ES API.

In most cases, you will be porting to a third-party OpenGL ES engine. Then the only task is to tune the build system to collect together Java Wireless Client software and the third-party engine.

Java Wireless Client software has ready-to-use make files for Hybrid's Gerbera engine (Win32 and Linux x86 platform are supported), nVidia's MX and PowerVR (TI OMAP 2420/2430 boards also known as H4/H5 boards). These files are located in `jsr239-com/src/config/sdk`.

To integrate a third-party OpenGL ES engine, make the following changes in the makefile, `jsr239/src/cldc-oi/config/subsystem.gmk`:

1. Add or modify paths to library header files. Here is an example:

```
SUBSYSTEM_JSR_239_EXTRA_INCLUDES += \  
    -I$(JSR_239_SDK_DIR)/include
```
  2. Change the path to the directory where library files are located, like this:

```
JSR_239_LIB_DIR = $(JSR_239_SDK_DIR)/lib/linux
```
  3. Specify the particular names of library files. You may use fully qualified names or specify directory and library names separately.

```
LIBS += -L$(JSR_239_LIB_DIR) -lGLES_CM
```
- or
- ```
LIBS += $(JSR_239_LIB_DIR)/GLES_CM.lib
```

4. Make sure that there is no conflict with standard libraries. For example, to assemble the code for Win32 using Visual C++ tool you need to exclude `libc` to avoid conflicts with `msvcrt`:

```
LD_FLAGS += -ndefaultlib:libcmt.lib
```

# 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.

|                                               |                                                                                                                                                                                                                                                                                                                                                                                     |
|-----------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>GCF</b>                                    | Generic Connection Framework. A part of CLDC, it improves network connectivity for wireless devices.                                                                                                                                                                                                                                                                                |
| <b>Home screen</b>                            | The main screen of the application manager. This is the screen the user sees after they exit an application.                                                                                                                                                                                                                                                                        |
| <b>HTTP</b>                                   | 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.                                                                                                                                                                                                             |
| <b>HTTPS</b>                                  | Secure HyperText Transfer Protocol. A protocol for transferring encrypted hypertext data using Secure Socket Layer (SSL) technology.                                                                                                                                                                                                                                                |
| <b>JAD file</b>                               | 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.                                                                                                                     |
| <b>JAR file</b>                               | 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.                                                               |
| <b>Java Community Process™ (JCP™) program</b> | 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.                                                                                                                    |
| <b>Java ME platform</b>                       | 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. |
| <b>Java Specification Request (JSR)</b>       | A proposal for developing new Java platform technology, which is reviewed, developed, and finalized into a formal specification by the JCP program.                                                                                                                                                                                                                                 |
| <b>Java Virtual Machine</b>                   | A software “execution engine” that safely and compatibly executes the byte codes in Java class files on a microprocessor.                                                                                                                                                                                                                                                           |
| <b>KVM</b>                                    | 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.                                                                                                                                                                                                                                        |
| <b>LCD</b>                                    | Liquid Crystal Display. A common kind of screen display often used in small devices.                                                                                                                                                                                                                                                                                                |

|                         |                                                                                                                                                                                                                                                                                                                          |
|-------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>LCDUI</b>            | 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.                                                                                                                         |
| <b>MIDlet</b>           | An application written for MIDP.                                                                                                                                                                                                                                                                                         |
| <b>MIDlet suite</b>     | 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.              |
| <b>MIDP</b>             | 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.                                                                          |
| <b>Obfuscation</b>      | 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.                                                                                                                                    |
| <b>Optional Package</b> | A set of Java ME platform APIs that provides additional functionality by extending the runtime capabilities of an existing configuration and profile.                                                                                                                                                                    |
| <b>PNG</b>              | Portable Network Graphics. An image format commonly used with MIDP that can be compressed, transmitted, and stored without losing image quality.                                                                                                                                                                         |
| <b>Preemption</b>       | Taking a resource, such as the foreground, from another application.                                                                                                                                                                                                                                                     |
| <b>Preverification</b>  | 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.                         |
| <b>Profile</b>          | 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.                                                                                                                    |
| <b>Provisioning</b>     | A mechanism for providing services, data, or both to a mobile device over a network.                                                                                                                                                                                                                                     |
| <b>Push Registry</b>    | 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. |
| <b>RMI</b>              | 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

---

## C

### CHAPI

- code structure, 39
- implementation, 42
- porting, 43
- porting layer, 40
- porting with JavaCall API, 44

## G

- Generic Connection Framework, 3

## J

- JavaCall API, 1

## M

### MMAPI

- capabilities, 13
- implementation, 15

## O

- OpenGL ES, 45
- overview, 1

## P

- porting SATSA optional packages, 17

## S

### SATSA

- access control, 34
- code structure, 17

### SATSA-APDU, 21

### SATSA-CRYPTO, 29

### SATSA-PKI, 31

### Security and Trust Services API

- porting of, 17
- source code, 1

## W

### Wireless Messaging API, 3

### Wireless Messaging subsystem

- permissions management, 11

### WMA, 3

- asynchronous transfers, 6
- design, 4
- message pool, 5, 9
- message size, 9
- permissions, 11
- porting, 7, 15
- push, 5, 10
- receiving messages, 8
- relationship between JSR 120 and JSR 205, 3
- sending messages, 8
- subsystem components, 4

