Development Kit User's Guide

For the Binary Release with Cryptography Extensions

Java Card™ Platform, Version 2.2.1

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

October, 2003

Contents

Tables

Figures

Preface

Java Card™ technology combines a subset of the Java™ programming language with a runtime environment optimized for smart cards and similar kinds of small-memory embedded devices. The goal of Java Card technology is to bring many of the benefits of Java software programming to the resource-constrained world of smart cards.

The Java Card API is compatible with international standards, such as ISO7816, and industry-specific standards, such as Europay/Master Card/Visa (EMV).

The *Development Kit User's Guide for the Java Card™ Platform, Version 2.2.1* contains information on how to install and use the Development Kit tools.

Who Should Use This Book

The *Development Kit User's Guide* is targeted at developers who are creating applets using the *Application Programming Interface for the Java Card™ Platform, Version 2.2.1* , and also at developers who are considering creating a vendor-specific framework based on the Java Card technology specifications.

Before You Read This Book

Before reading this guide, you should be familiar with the Java programming language, object-oriented design, the Java Card technology specifications, and smart card technology. A good resource for becoming familiar with Java and Java Card technology is the Sun Microsystems, Inc. web site, located at: `http://java.sun.com`.

How This Book Is Organized

Chapter 1 "Introduction to the Development Kit for the Java Card Platform," provides an overview of the Development Kit and its tools.

Chapter 2 "Installation," describes the procedures for installing the tools included in this release.

Chapter 3 "Development Kit Samples and Demonstrations," describes sample applets which illustrate the use of the Java Card API. It also describes demonstration programs which illustrate very important scenarios of applet masking and post-manufacture installation.

Chapter 4 "Running Applets in an Emulated Card Environment," provides an overview of the Java Card technology-based Workstation Development Environment ("Java Card WDE") and details of how to run it.

Chapter 5 "Converting Java Class Files," provides an overview of the Converter and details on how to run it.

Chapter 6 "Viewing an Export File," describes how to use the `exp2text` tool to view any export file in ASCII format.

Chapter 7 "Verifying CAP and Export Files," provides an overview of the off-card verifier tool and details of running it.

Chapter 8 "Generating a CAP File from a Java Card Assembly File," describes how to use the capgen utility.

Chapter 9 "Producing a Text Representation of a CAP File," describes how to use the capdump utility.

Chapter 11 "Using the Java Card Reference Implementation," describes how to use the C-language runtime environment simulator for the Java Card platform ("Java Card Runtime Environment" or "Java Card RE").

Chapter 12 "Using the Installer," describes how to download and delete packages, and create and delete applet instances using the installer.

Chapter 13 "Sending and Receiving APDU Commands," describes how to use `apdutool` to transfer APDUs to and from the C-language Java Card Runtime Environment or Java Card Workstation Development Environment (WDE).

Chapter 14 "Using Cryptography Extensions," describes the cryptography APIs optionally provided with this release.

Chapter 15 "Java Card RMI Client-Side Reference Implementation," describes the reference implementation of Java Card technology-based client-side Remote Method Invocation API ("client-side Java Card RMI API").

Appendix A "Java Card Assembly Syntax Example," describes the Java Card platform assembly output of the Converter using a commented example file.

Appendix B "CAP File Manifest File Syntax," describes the syntax of the manifest file which the Converter includes in the CAP file.

Appendix C "Using the Large Address Space," describes how your applications can get the most out of a large address space implementation.

Appendix D "Reference Implementation of Java Card RMI Client-Side API," provides the documentation for the client-side Java Card RMI API.

Related Books

References to various documents or products are made in this manual. You should have the following documents available:

- *Application Programming Notes for the Java Card™ Platform, Version 2.2.1* (Sun Microsystems, Inc., 2003)
- *Application Programming Interface for the Java Card™ Platform, Version 2.2.1* (Sun Microsystems, Inc., 2003)
- *Virtual Machine Specification for the Java Card™ Platform, Version 2.2.1* (Sun Microsystems, Inc., 2003)
- *Runtime Environment Specification for the Java Card™ Platform, Version 2.2.1* (Sun Microsystems, Inc., 2003)

- *Off-Card Verifier for the Java Card™ Platform, Version 2.2.1, White Paper* (Sun Microsystems, Inc., 2003)
- *Java Card™ RMI Client Application Programming Interface, Version 2.2.1* (Sun Microsystems, Inc., 2003)
- *The Java™ Programming Language (Java Series), Second Edition* by Ken Arnold and James Gosling (Addison-Wesley, 1998).
- *The Java™ Virtual Machine Specification (Java Series), Second Edition* by Tim Lindholm and Frank Yellin (Addison-Wesley, 1999)
- *Java Card Technology for Smart Cards* by Zhiqun Chen (Addison-Wesley, 2000)
- *The Java Class Libraries: An Annotated Reference, Second Edition (Java Series)* by Patrick Chan, Rosanna Lee and Doug Kramer (Addison-Wesley, 1999)
- *ISO 7816 Specification* Parts 1-6

You can download version 2.2.1 of the Java Card specifications from the Sun Microsystems web site:

`http://java.sun.com/products/javacard`

Typographic Conventions

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

Accessing Sun Documentation Online

The Java Developer Connection[sm] web site enables you to access Java™ platform technical documentation on the Web:

http://developer.java.sun.com/developer/infodocs/

Sun Welcomes Your Comments

We are interested in improving our documentation and welcome your comments and suggestions. You can email your comments to us at:

docs@java.sun.com

CHAPTER 1

Introduction to the Development Kit for the Java Card Platform

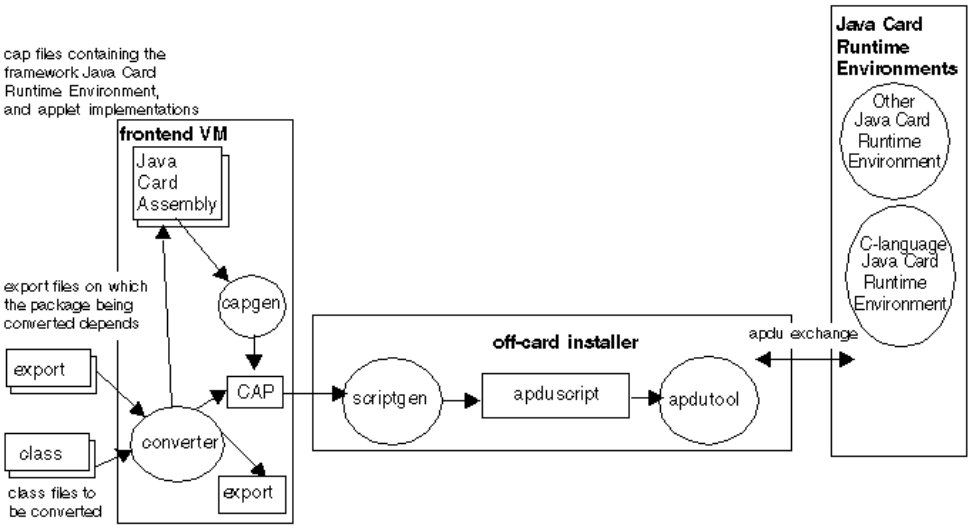
The Development Kit for the Java Card platform (“Java Card Development Kit”) is a suite of tools for designing Java Card technology-based implementations and for developing applets based on the *Application Programming Interface for the Java Card™ Platform, Version 2.2.1* .

CAP File Flow

As illustrated in FIGURE 1, the data flow starts with Java source being compiled and input to the Converter. The Converter tool can convert classes that comprise a Java package to a converted applet (CAP) or a Java Card technology-based Assembly (“Java Card Assembly”) file. A CAP file is a binary representation of converted Java package.

A Java Card Assembly file is a human-readable text representation of a converted package which you can use to aid testing and debugging. A Java Card Assembly file can also be used as input to the `capgen` tool to create a CAP file.

FIGURE 1 Java Card Platform Version 2.2.1 CAP Tool Architecture



Not shown in the figure is a utility called `capdump`, which produces a simple ASCII version of the CAP file to aid in debugging.

CAP files are processed by an off-card installer (`scriptgen`). This produces an APDU script file as input to the `apdutool`, which then sends APDUs to a Java Card RE implementation.

Any implementation of a Java Card RE contains a virtual machine (VM) for the Java Card platform ("Java Card virtual machine"), the Java Card Application Programming Interface (API) classes, and support services.

FIGURE 1 also illustrates other Java Card REs which may be available in other products. In this release, the only Java Card RE provided is written in the C programming language.

CHAPTER 2

Installation

This release is provided for the Solaris™ Operating System (OS) release 8, Red Hat Linux version 7.2, and for Microsoft Windows 2000 (with Service Pack 4) as compressed Zip archives.

Note – The Linux platform version of the toolkit is unsupported and has undergone only limited testing. The Linux platform version was tested only on the English language Red Hat Linux 7.2 and `gcc` version 2.96.

In the future, we will release new Linux platform versions of the toolkit, but we do not commit to doing so with the same frequency as, nor simultaneously with new Solaris or Microsoft Windows 2000 platform versions. We do not commit to making fully-supported Linux platform versions of the toolkit. In addition, we do not commit to addressing problems or bug reports submitted against the Linux platform version.

Note – Do not overlay a previous release with this release. Instead perform the installation into a new directory.

Prerequisites for Installing the Binary Release

1. **Install the Java 2 Standard Edition Software Developer's Kit (SDK) from:**

   http://java.sun.com/j2se/

   The supported SDK version is 1.4.1. If you are installing the SDK on the Solaris or Linux platform, make sure that all of the required patches are installed. To get more information, refer to the product documentation available at:

   http://www.sun.com/solaris/java

2. **Install `javax.comm`. If you are not using the Development Kit to communicate with a card reader installed on a serial port, you can skip this step.**

   The package `javax.comm` can be found in the Java Communications API 2.0. To obtain the package, please visit Sun's web site at:

   http://java.sun.com/products/javacomm

   Separate versions of the `javax.comm` API are available for the Solaris/SPARC, Microsoft Windows 2000, and Solaris/x86 platforms.

   Note – If you are using the Development Kit on the Linux platform, download the Solaris/x86 release of the `javax.comm` API and install only the jar files.

   Follow the instructions provided in the file `Readme.html` to install the package. Make sure that the `comm.jar` file is added to the `CLASSPATH`.

3. **Install the OpenCard Framework (OCF) Version 1.2.**

   Download the zipped version of the OpenCard Framework Base package from the OCF web site:

   http://www.opencard.org/index-downloads.html

   Unzip the package into a separate directory.

Installing the Development Kit Binaries

There are three steps to install the Development Kit binaries. Separate sections cover installation for the Solaris, Linux, and Microsoft Windows 2000 platforms.

- **Installing Development Kit binary release files**: See the "Installing on the Solaris or Linux Platform" or the "Installing on the Microsoft Windows 2000 Platform" on page 6.
- **Setting Environment Variables**: See "Setting Environment Variables for the Solaris or Linux Platform" on page 5 or "Setting Environment Variables for Microsoft Windows 2000 Platform" on page 7.
- **Copying Open Card Framework files** (required by Java Card RMI client-side Reference Implementation): See "Copying OpenCard Framework Files" on page 7.

▼ Installing on the Solaris or Linux Platform

The Java Card Development Kit provides separate download files for the binary release for the Solaris and Linux platforms. For the exact names of the download files, see the Java Card Development Kit Release Notes for the Binary Release.

1. **Save the file in a convenient installation location of your choice: for example, in the directory** `/javacard`**.**
2. **Navigate to the** `/javacard` **directory:**

   ```
   % cd /javacard
   ```

3. **Unzip the file provided with the release with the** `unzip` **utility.**

   ```
   % unzip <Development_Kit_binary_distribution>.zip
   ```

   where *`Development_Kit_binary_distribution`* refers to the name of the bundle containing the binary release installation files for the Solaris or Linux platform.

   The installation creates a directory `java_card_kit-2_2_1` under `/javacard`. The `/javacard/java_card_kit-2_2_1` directory is now the root of the development kit installation.

4. **Follow the directions** "Setting Environment Variables for the Solaris or Linux Platform" **to set the environment variables required by the Development Kit.**

   For a description of the files that are installed under `java_card_kit-2_2_1`, see "Files Installed for the Binary Release" on page 8.

▼ Setting Environment Variables for the Solaris or Linux Platform

1. **Set the environment variable** `JC_HOME` **to the installation directory. For example (using** `csh`**), if you unzipped the release in the directory** `/javacard`**:**

   ```
   setenv JC_HOME /javacard/java_card_kit-2_2_1
   ```

Or, if you unzipped the installation into a different directory, define the environment variable `JC_HOME` accordingly.

2. **Set the environment variable `JAVA_HOME` to the directory where you installed your Java development tools. For example,**

   ```
   setenv JAVA_HOME /usr/j2sdk1.4.1
   ```

   The following optional path setting will enable you to run the Development Kit tools from any directory.

   ```
   setenv PATH .:$JC_HOME/bin:$PATH
   ```

   We suggest you automate these environment settings. Create a `csh` script file (named, for example, `javacard_env.cshrc`) which includes the `setenv` statements:

   ```
   setenv JC_HOME /javacard/java_card_kit-2_2_1
   setenv JAVA_HOME /usr/j2sdk1.4.1
   setenv PATH .:$JC_HOME/bin:$JAVA_HOME/bin:$PATH:
   ```

   Run the script file from the command prompt before running the Development Kit tools, samples, and demonstrations (refer to Chapter 3 “Development Kit Samples and Demonstrations”):

   ```
   % source javacard_env.cshrc
   ```

▼ Installing on the Microsoft Windows 2000 Platform

The Java Card Development Kit provides a separate download file for the binary release for the Microsoft Windows 2000 platform. For the exact name of the download file, see the Java Card Development Kit Release Notes for the Binary Release.

1. **Save the zip file in a convenient installation location of your choice. For example, the root of the `C:` drive.**

2. **Unzip the file provided with the release with the Winzip utility (available from `http://www.winzip.com`).**

   ```
   C:\> winzip32 <Development_Kit_binary_distribution>.zip
   ```

   where *`Development_Kit_binary_distribution`* refers to the name of the bundle containing the installation files for the Microsoft Windows 2000 platform.

   In the Winzip dialog, choose Select All and Extract from the Actions menu. Enter `C:\` into the Extract To field to unzip the contents of the zip file into that directory. (For more information, refer to the Winzip documentation.)

   The `java_card_kit-2_2_1` directory is the root of the Development Kit installation.

3. **Follow the directions below to set the Microsoft Windows 2000 platform environment variables required by the Development Kit.**

   For a description of the files that are installed under `java_card_kit-2_2_1`, see "Files Installed for the Binary Release" on page 8.

▼ Setting Environment Variables for Microsoft Windows 2000 Platform

1. **Set the environment variable `JC_HOME` to the installation directory. For example, if you unzipped the release in the root directory of the `C:` volume:**

   ```
   set JC_HOME=c:\java_card_kit-2_2_1
   ```

   Or, if you unzipped the installation into a different directory, define the environment variable `JC_HOME` accordingly.

2. **Set the environment variable `JAVA_HOME` to the directory where you installed your Java development tools. For example,**

   ```
   set JAVA_HOME=c:\j2sdk1.4.1
   ```

   The following optional path setting will enable you to run the Development Kit tools from any directory.

   ```
   set PATH=%JC_HOME%\bin;%JAVA_HOME%\bin;%PATH%
   ```

   We suggest you automate these environment settings. Create a batch file (named, for example, `javacard_env.bat`) which includes the `set` statements:

   ```
   @echo off
   set JC_HOME=C:\java_card_kit-2_2_1
   set JAVA_HOME=c:\j2sdk1.4.1
   set PATH=.;%JC_HOME%\bin;%JAVA_HOME%\bin;%PATH%
   ```

   Run the batch file from the command prompt before running the Development Kit tools, samples, and demonstrations (refer to Chapter 3 "Development Kit Samples and Demonstrations"):

▼ Copying OpenCard Framework Files

For Solaris, Linux, and Microsoft Windows 2000 Platforms:

To run the Java Card RMI demos, you must copy certain OpenCard Framework (OCF) jar files into the `lib` subdirectory of your `JC_HOME` directory.

1. **Navigate to the directory** <*ocf_download_home*>/OCF1.2/lib/ **(on the Solaris and Linux platform) or** <*ocf_download_home*>\OCF1.2\lib\ **(on the Microsoft Windows 2000 platform) of the OCF installation. In this path,** <*ocf_download_home*> **represents the directory where OpenCard Framework files were unzipped.**

2. **Copy the files** base-core.jar **and** base-opt.jar **from this directory into the** $JC_HOME/lib **(Solaris or Linux platform) or** %JC_HOME%\lib **(Microsoft Windows 2000 platform) directory.**

Files Installed for the Binary Release

TABLE 1 describes the files and directories that the binary installation procedure installs under java_card_kit-2_2_1.

Note – If you are using the Microsoft Windows 2000 platform, substitute the \ character for / in the paths.

TABLE 1 Directories and Files Installed for the Binary Release

Directory/File	Description
api_export_files	Contains the export files for version 2.2.1 of the Java Card API packages.
bin	Contains all shell scripts or batch files for running the tools (such as the apdutool, capdump, converter and so forth), and the cref binary executable.
doc/en	The api, devnotes, and guides subdirectories contain the English-language guides for this release: • api—contains the HTML files for the Java Card specification produced by the Javadoc™ tool. • devnotes—contains the *Application Programming Notes for the Java Card™ Platform, Version 2.2.1* in PDF format. The html subdirectory contains the same manual in HTML format. • guides—contains this document: the *Development Kit User's Guide for the Java Card™ Platform, Version 2.2.1*, in PDF format, and an html directory containing the document in HTML format. • j2me-docs.css—cascading style sheet used by the release note HTML files.

TABLE 1 Directories and Files Installed for the Binary Release

Directory/File	Description
`lib`	Contains all Java jar files required for the tools: • `apdutool.jar` and `apduio.jar`—used by `apdutool` • `api.jar` (with cryptography extensions)—needed to write Java Card technology-based applets ("Java Card applets") and libraries • `capdump.jar`—needed to produce an ASCII representation of a CAP file • `converter.jar`—needed to process Java class files and Java Card export files • `javacardframework.jar`—used by the Java RMIC compiler for generating stubs for Java Card RMI applications • `jcclientsamples.jar`—contains the client part of the Java Card RMI samples • `jcrmiclientframework.jar`—contains the classes of the Java Card RMI Client API • `jcwde.jar` (with cryptography extensions)—used by Java Card WDE • `installer.jar`—contains the installer applet • `offcardverifier.jar`—needed to evaluate CAP and export files in a desktop environment • `scriptgen.jar`—needed to convert a package in a CAP file into a script file containing a sequence of APDUs
`samples`	Contains sample applets and demonstration programs. For more information on the contents of this directory, see "Sample Programs and Demonstrations" on page 9.
`COPYRIGHT_dom`	(`COPYRIGHT_dom.txt` on Microsoft Windows 2000) Contains the copyright notice for the product.
`LICENSE.html`	Contains the text of the license agreement.
`RELEASENOTES.html`	Contains important information about this release.

Sample Programs and Demonstrations

All samples are contained in the `samples` directory under `JC_HOME`. TABLE 2 describes the contents of the directory.

TABLE 2 Directory Structure for Sample Programs and Demonstrations

Directory/File	Description
`classes`	Contains pre-built sample classes.
`build_samples` or `build_samples.bat`	A script or batch file to automate building samples.
`src`	Contains the sources for the sample applets that belong to the packages `com.sun.javacard.samples.*`.
`src/demo`	Contains all of the files needed to run the Java Card demonstration programs. For more information on the contents of the `demo` directory, see "Directories and Files in the demo Directory" on page 12.
`src/com/sun/javacard/samples`	Contains the source code for the sample applets.
`src_client`	Contains sample card acceptance device (CAD) client programs for the Photo Card, Java Card RMI, and secure Java Card RMI demos.

CHAPTER 3

Development Kit Samples and Demonstrations

This release includes several demonstration programs which illustrate the use of the Java Card API, and a scenario of post-manufacture installation.

The Demonstrations

Version 2.2.1 of the Development Kit includes the following demonstration programs:

- Demo 1 (`demo1`)—illustrates the use of packages masked into card ROM: `JavaPurse`, `JavaLoyalty`, `Wallet` and `SampleLibrary`.
- Demo 2 (`demo2`)—downloads these packages into the C-language Java Card RE using the installer applet: `JavaPurse`, `JavaLoyalty`, `Wallet`, `SampleLibrary`, `RMIDemo`, `SecureRMIDemo`, and `photocard`. `demo2` also exercises the `JavaPurse`, `JavaLoyalty`, and `Wallet` applets.
- Demo 2 Cryptography Demo (`demo2crypto`)—is similar to `demo2`, except it uses a version of `JavaPurse` that uses a DES MAC algorithm.
- Demo 3 (`demo3`)—illustrates the second time power-up of an already initialized mask. It uses the card state file created by `demo2`.
- Java Card RMI Demo (`RMIDemo`)—demonstrates the use of the Java Card platform Remote Method Invocation ("Java Card RMI") API. The basic example used is a program that manages a counter remotely, and is able to decrement, increment, and return the value of an account. On `cref`, `RMIDemo` uses the card state file created by `demo2`.
- Logical Channels Demo (`channelDemo`)—demonstrates the use of logical channels which allows selecting multiple applets at the same time.

- Object Deletion Demo 1 (`odDemo1`)—demonstrates applet and package deletion, as well as the object deletion mechanism which removes unreachable objects.
- Object Deletion Demo2 (`odDemo2`)—demonstrates package deletion and checks that persistent memory has been returned to the memory manager.
- Photo Card Demo (`photocard`)—demonstrates how you can store images in the large address space which is available in the 32-bit version of the reference implementation for the Java Card platform ("Java Card reference implementation"), version 2.2.1.
- Secure Java Card RMI Demo (`SecureRMIDemo`)—is similar to `RMIDemo`, but demonstrates additional security at the transport level. It also uses the card state file created on `cref` by `demo2`.

Directories and Files in the demo Directory

The directories and files for the Development Kit demonstrations are described in TABLE 3 and TABLE 4.

Note – Many of the directories listed in TABLE 3 and TABLE 4 contain a `_tmp` subdirectory. This subdirectory contains intermediate temporary files needed to construct the final `*.scr` source files.

TABLE 3 Directories and Files in the demo Directory

Directories/Files	Description
demo1	Contains the files required to run and verify demo1: • demo1.scr—demonstration apdutool script file • demo1.scr.expected.out—for comparison with apdutool output when the demo is run
demo2	Contains the files required to run and verify demo2 and demo2crypto: • demo2.scr, demo2crypto.scr—demonstration apdutool script files • demo2.scr.expected.out, demo2crypto.scr.expected.out—for comparison with apdutool output when the demo is run This directory also contains the subdirectories for the demos that depend on the output of demo2. For more information on the contents of these subdirectories and the demos they represent, see TABLE 4.
demo3	Contains the files required to run and verify demo3: • demo3.scr—demonstration apdutool script file • demo3.scr.expected.out—for comparison with apdutool output when the demo is run
jcwde	Contains the files required to run Java Card WDE: • jcwde.app—lists all of the applets (and their AIDs) to be loaded into the simulated mask for Java Card WDE. • jcwde_rmi.app and jcwde_securermi.app—lists the contents of Java Card WDE for running the RMIDemo and SecureRMIDemo respectively.

TABLE 3 Directories and Files in the demo Directory

Directories/Files	Description
`logical_channels`	Contains the files required to run and verify the logical channels demo: • `channel.scr`, `channelDemo.scr`, `ChnDemo.scr`—demonstration `apdutool` script files • `channelDemo.scr.expected.out`—for comparison with `apdutool` output when the demo is run
`misc`	`Footer.scr`, `Header.scr`—scripts to terminate and initialize the session, respectively.
`object_deletion`	Contains the files required to run and verify `odDemo1` and `odDemo2`: • `packageA.scr`, `packageB.scr`, `packageC.scr`—intermediate script files for building the final `odDemo1-*.scr` files. • `odDemo1-1.scr`, `odDemo1-2.scr`, `odDemo1-3.scr`—demonstration `apdutool` script files • `od1.scr`, `od2.scr`, `od2-2.scr`, `od3.scr`, `od3-2.scr`— script files used for building the `odDemo1-*.scr` files • `odDemo1-1.scr.expected.out`, `odDemo1-2.scr.expected.out`, `odDemo1-3.scr.expected.out`, `odDemo2.scr.expected.out`—for comparison with `apdutool` output when the demos are run

Several of the Development Kit demonstrations use the output generated by the Demo 2 demonstration. These demonstrations are stored in subdirectories of `demo2`. The `demo2` directory also contains the files that the demos need to run `JavaPurse`, `JavaLoyalty`, and `Wallet`. The demonstrations and subdirectories contained in `demo2` are described in TABLE 4.

TABLE 4 Subdirectories and Demonstrations in the demo2 Directory

Subdirectories	Description
`javapurse`	Contains the files required to run the demos that use `JavaPurse`: • `AppletTest.scr`, `AppletTestCrypto.scr`—downloads and executes the demonstration applets • `JavaLoyalty.scr`—installation script for the `JavaLoyalty` Java Card applet • `JavaPurse.scr`, `JavaPurseCrypto.scr`—installation scripts for the `JavaPurse` Java Card applet • `SampleLibrary.scr`—installation script for the `SampleLibrary` library package
`photocard`	Contains the files required to run and verify the photo card demo: • `photocard`, `photocard.bat`—script/batch file to run the photo card demo • `photocard.scr`—installation script for the photocard applet package • `photocard.scr.expected.out`—for comparison with `apdutool` output when the demo is run • `*.gif` files—sample photo files • `opencard.properties`—contains OCF settings for the photo card demo
`rmi`	Contains the files required to run and verify `RMIDemo` and `SecureRMIDemo`: • `rmidemo` or `rmidemo.bat`, `securermidemo` or `securermidemo.bat`—Shell scripts and batch files for running the Java Card RMI and secure Java Card RMI demos, respectively • `rmidemo.scr.expected.out`, `securermidemo.scr.expected.out`—for comparison with `apdutool` output when the demos are run • `RMIDemo.scr`, `SecureRMIDemo.scr`—installation scripts to install the `RMIDemo` and `SecureRMIDemo` applet packages, respectively • `opencard.properties`—contains OCF settings for the Java Card RMI demos
`wallet`	Contains the file required to run the demos that use the `Wallet` applet: `Wallet.scr`—installation script for the `Wallet` applet package

Preliminaries

All demo programs are pre-built. If you make any changes to the demos, the following sections describe how you can rebuild them.

Building Samples

A script file is provided to build the samples. To understand what is going on behind the scenes, it is very instructive to look at the script.

The script file is `$JC_HOME/samples/build_samples` (for a Solaris or Linux platform installation) or `%JC_HOME%\samples\build_samples.bat` (for a Microsoft Windows 2000 platform installation).

Running the Script

The command line syntax for the script is:

```
build_samples [options]
```

TABLE 5 describes the possible values for *options*.

TABLE 5 build_samples Command Line Options

Value of *options*	Description
`-clean`	Removes all files produced by the script.
`-help`	Prints out a help message and exits.
[no options]	Builds all sample applets, client samples, and demo scripts.

Setting Environment Variables

The `build_samples` script uses the environment variables `JC_HOME` and `JAVA_HOME`. To correctly set these environment variables, refer to "Setting Environment Variables for the Solaris or Linux Platform" on page 5 or "Setting Environment Variables for Microsoft Windows 2000 Platform" on page 7.

Building the Sample Applets

Run the script without parameters to build the samples:

```
build_samples
```

▼ Preparing to Compile the Sample Applets

Note – This section details the steps taken by the script, but you can run the commands yourself if you choose.

1. **A `classes` directory is created as a peer to `src` under the `samples` directory.**
2. **The Java Card API export files are copied to the `classes` directory.**

Compiling the Sample Applets

The next step is to compile the Java sources for the sample applets. For example, from the `samples` directory, issue the following command:

Solaris or Linux platform:

```
javac -g -classpath ./classes:../lib/api.jar:../lib/installer.jar
    src/com/sun/javacard/samples/HelloWorld/*.java
```

Microsoft Windows 2000 platform:

```
javac -g -classpath .\classes;..\lib\api.jar;..\lib\installer.jar
    src\com\sun\javacard\samples\HelloWorld\*.java
```

where:

- `api.jar` contains the Java Card API
- `installer.jar` contains the installer applet

- the `classes` directory is required for packages that import other sample packages

Converting the Class Files

The next step is to convert the Java class files.

Conversion parameters for each package are specified in a configuration file.

For example, a configuration file contains items such as:

```
-out EXP JCA CAP
-exportpath .
-applet  0xa0:0x0:0x0:0x0:0x62:0x3:0x1:0xc:0x1:0x1
com.sun.javacard.samples.HelloWorld.HelloWorld
com.sun.javacard.samples.HelloWorld
0xa0:0x0:0x0:0x0:0x62:0x3:0x1:0xc:0x1 1.0
```

In this example, the `converter` will output three kinds of files: export (`*.exp`), CAP (`*.cap`) and Java Card Assembly (`*.jca`) files.

Refer to the "Convert samples" section of the script file to see the detailed `converter` tool steps.

For more information about the `converter` tool, refer to Chapter 5 "Converting Java Class Files."

Running scriptgen to Generate Script Files

Generate script files for `apdutool` using the `scriptgen` tool. This step must be done for each package to be downloaded. For example:

```
scriptgen -o JavaLoyalty.scr ./classes/com/sun/javacard/samples/
    JavaLoyalty/javacard/JavaLoyalty.cap
```

The new scripts are included into the demonstration scripts. For example, `demo2.scr` file is composed of these scripts:

- `Header.scr`—a script that initializes the session
- `SampleLibrary.scr`, `JavaLoyalty.scr`, `JavaPurse.scr`, `Wallet.scr`, `RMIDemo.scr`, `SecureRMIDemo.scr`, `photocard.scr`—package installation scripts
- `AppletTest.scr`—a script that creates the `JavaLoyalty`, `JavaPurse`, `Wallet`, `JavaPurseCrypto`, `RMIDemo`, and `SecureRMIDemo` applets so that you can see each of them invoked when the simulation is run. `AppletTest.scr` also excercises the `JavaLoyalty.scr`, `JavaPurse.scr`, and `Wallet.scr` applets.

- `Footer.scr`—a script that terminates the session

Note – The script files for the demonstrations use the `output off;` apdutool directive to suppress the logging of CAP file download APDU commands to the output log file, and the `output on;` directive to enable the logging of other commands. To enable logging of package download commands, comment out the `output off;` directive in the script file `Header.scr` and run the `build_samples` script.

Running the Demonstrations

The following sections describe the Development Kit demonstrations and how to run them.

A demonstration can use a card EEPROM image created by another demonstration. The `cref` command line option `-o` <*filename*> lets you save the EEPROM image into a file after a simulated card session. The option `-i` <*filename*> restores the image from the file for a new card session. For more information, see Chapter 11 "Using the Java Card Reference Implementation".

Demo 1

The Demo 1 demonstration, `demo1`, exercises the `JavaPurse`, `JavaLoyalty`, and `Wallet` applets by simulating transactions where amounts are credited and debited from the card. The demonstration begins by powering up the Java Card technology-enabled smart card and creating the applets `JavaPurse`, `JavaLoyalty`, and `Wallet`.

The `JavaPurse` applet demonstrates a simple electronic cash application. The applet is selected and initialized with various parameters such as the Purse ID, the expiration date of the card, the Master and User PINs, maximum balance, and maximum transaction. Transaction operations perform the actual debits and credits to the electronic purse. If there is a configured loyalty applet assigned for the CAD performing the transaction `JavaPurse` communicates with it to grant loyalty points. In this case, `JavaLoyalty` is the provided loyalty applet.

A number of transaction sessions are simulated where amounts are credited and debited from the card. In an additional session, transactions with intentional errors are attempted to demonstrate the security features of the card.

The JavaLoyalty applet is designed to interact with the JavaPurse applet, and to demonstrate the use of shareable interfaces. The shareable JavaLoyaltyInterface is defined in a separate library package, com.sun.javacard.SampleLibrary.

JavaLoyalty is a minimalistic loyalty applet. It is registered with JavaPurse when a Parameter Update APDU command with an appropriate parameter tag is executed, and when the AID part of the parameter corresponds to the AID of the JavaLoyalty applet. The applet contains a grantPoints method. This method implements the main interaction with the client. When the first 2 bytes of the CAD ID in a request by a JavaPurse transaction correspond to the 2 bytes of CAD ID in the corresponding Parameter Update APDU command, the grantPoints method implementing the JavaLoyaltyInterface is requested.

JavaLoyalty maintains the balance of loyalty points. The applet contains methods to credit and debit the account of points and to get and set the balance.

The Wallet applet demonstrates a simplified cash card application. It keeps a balance, and exercises some of the Java Card API features such as the use of a PIN to control access to the applet.

▼ Running demo1

demo1 runs in the Java Card WDE.

1. **Navigate to the** jcwde **directory (this will be** $JC_HOME/samples/src/demo/jcwde **on the Solaris or Linux platform or** $JC_HOME\samples\src\demo\jcwde **on the Microsoft Windows 2000 platform). Enter the command:**

   ```
   jcwde jcwde.app
   ```

2. **In a separate command window, navigate to the** $JC_HOME/samples/src/demo/demo1 **directory (or** %JC_HOME%\samples\src\demo\demo1 **on the Microsoft Windows 2000 platform) and run** apdutool**, using the following command:**

   ```
   apdutool -nobanner -noatr demo1.scr > demo1.scr.jcwde.out
   ```

   If the run is successful, the apdutool log, demo1.scr.jcwde.out, is identical to the file demo1.scr.expected.out.

Demo 2

The Demo 2 demonstration, demo2, illustrates downloading Java Card packages onto the card. This demonstration contains the installer applet in the mask image. After the card is powered up, the Photocard, SampleLibrary, JavaPurse, JavaLoyalty, Wallet, RMIDemo, and SecureRMIDemo packages are downloaded. The commands from demo1 are repeated. Finally, the card is powered down.

▼ Running demo2

demo2 runs in the C-language Java Card RE because the Java Card WDE is not able to support the downloading of CAP files.

1. **Run** cref **using the following command:**

   ```
   cref -o demoee
   ```

2. **In a separate command window, navigate to the** $JC_HOME/samples/src/demo/demo2 **directory (or** %JC_HOME%\samples\src\demo\demo2 **on the Microsoft Windows 2000 platform) and run** apdutool**, using the following command:**

   ```
   apdutool -nobanner -noatr demo2.scr > demo2.scr.cref.out
   ```

   If the run is successful, the apdutool log, demo2.scr.cref.out, should be identical to the file demo2.scr.expected.out.

   After cref completes executing, an EEPROM image is stored in the file demoee. (For more information, refer to Chapter 11 "Using the Java Card Reference Implementation.")

Demo 3

The Demo 3 demonstration, demo3, illustrates the capabilities of a Java Card technology-enabled smart card to save its state across sessions. After running demo2, the state of the card can be saved. This card state must be used as the initial state for running demo3.

▼ Running demo3

demo3 should be run after demo2. demo3 runs in the C-language Java Card RE because the virtual machine state must be restored after the initial run.

1. **Run** cref **using the following command:**

   ```
   cref -i demoee
   ```

   cref will restore the EEPROM image from the file demoee. (For more information, refer to Chapter 11 "Using the Java Card Reference Implementation.")

2. **In a separate command window, navigate to the** $JC_HOME/samples/src/demo/demo3 **directory (or** %JC_HOME%\samples\src\demo\demo3 **on the Microsoft Windows 2000 platform) and run** apdutool**, using the following command:**

   ```
   apdutool -nobanner -noatr demo3.scr > demo3.scr.cref.out
   ```

   If the run is successful, the apdutool log, demo3.scr.cref.out, should be identical to the file demo3.scr.expected.out.

Java Card RMI Demo

Every Java Card RMI application consists of two parts: a card applet and a client program communicating with it. In this case, the `RMIDemo` applet is installed in EEPROM image when you run `demo2` on CREF. On Java Card WDE, the applets are included in the simulated mask.

The client framework uses the OCF framework, version 1.2 (files `base-core.jar`, and `base-opt.jar`), to communicate with C-language Java Card RE. The OCF settings are stored in file `opencard.properties`. The file `opencard.properties` is located in the `demo\demo2\rmi` directory.

Note that OCF searches for this file in the following locations:

- `[java.home]/lib/opencard.properties`
- `[user.home]/.opencard.properties`
- `[user.dir]/opencard.properties`
- `[user.dir]/.opencard.properties`

where `[java.home]`, `[user.home]`, and `[user.dir]` correspond to the Java system properties.

The `RMIDemo` uses the card applet `PurseApplet`, the `Purse` interface and its implementation `PurseImpl`. These classes reside in the package `com.sun.javacard.samples.RMIDemo`. The client-side program `PurseClient` resides in the package `com.sun.javacard.clientsamples.purseclient`.

The `Purse` interface describes the supported functionality: methods for getting the account balance, debiting and crediting the account, and getting and setting an account number. The interface also defines the constants used for error reporting. The `PurseImpl` class implements `Purse`.

The card applet `PurseApplet` creates and registers instances of the dispatcher and the Java Card RMI service.

The client-side program, `PurseClient`, represents a simple Java Card RMI client and uses the Open Card Framework (OCF) as the client side framework. The program initializes the OCF, creates the Java Card RMI `Connect` instance, and selects the Java Card applet (in this case, the `PurseApplet`). The program then gets the initial reference from `PurseApplet` (the reference to an instance of `PurseImpl`) and casts it to the `Purse` interface type. This allows `PurseImpl` to be treated as a local object. The program can then exercise the card by debiting and crediting different amounts, and by setting and getting the account number. The program demonstrates error handling by intentionally attempting to set an account number of incorrect size. This will cause a `UserException` to be thrown with the appropriate error code.

The client part of the RMIDemo can be run without parameters or with the -i parameter:

- If the demo is run without parameters, remote references are identified using the class name of the remote object.
- If the demo is run with the -i parameter, remote references are identified using the list of remote interfaces implemented by the remote object.

For more information on these formats, see Chapter 8 of the *Runtime Environment Specification for the Java Card™ Platform, Version 2.2.1* (Sun Microsystems, Inc., 2003).

▼ Running the Java Card RMI Demo

The RMIDemo can be run on either C-language Java Card RE or Java Card WDE. On C-language Java Card RE, RMIDemo can be run only after demo2 has successfully completed.

The RMIDemo applet can be run in Java Card WDE by listing it on the first line of the applet configuration file jcwde_rmi.app.

If the run is successful, the output in the file will be the same as contained in file rmidemo.scr.expected.out.

To Run RMIDemo on the C-Language Java Card RE:

1. **Run** cref **using the command:**

   ```
   cref -i demoee
   ```

2. **Run the Java Card RMI client program with either of these commands:**

   ```
   rmidemo > rmidemo.scr.cref.out
   rmidemo -i > rmidemo.scr.cref.out
   ```

To Run RMIDemo on Java Card WDE:

1. **Run Java Card WDE using the command:**

   ```
   jcwde jcwde_rmi.app
   ```

2. **In a separate command window, navigate to the** $JC_HOME/samples/src/demo/demo2/rmi **directory (on the Solaris or Linux platform) or** %JC_HOME%\samples\src\demo\demo2\rmi **directory (on the Microsoft Windows 2000 platform). Run the Java Card RMI client program with either of these commands:**

   ```
   rmidemo > rmidemo.scr.jcwde.out
   rmidemo -i > rmidemo.scr.jcwde.out
   ```

Secure Java Card RMI Demo

The secure Java Card RMI demo, `SecureRMIDemo`, can be thought of as a version of the `RMIDemo` with an added security service. `SecureRMIDemo` uses the card applet `SecurePurseApplet`, the `Purse` interface and its implementation `SecurePurseImpl`, and a definition of the security service `MySecurityService`. These classes reside in the package `com.sun.javacard.samples.SecureRMIDemo`. The demo also uses the client-side program `SecurePurseClient` and the specialized card accessor `SecureOCFCardAccessor`. These classes reside in the package `com.sun.javacard.clientsamples.securepurseclient`.

The `Purse` interface is similar to the interface used in the non-secure case, however, there is an extra constant: `REQUEST_DENIED`. This constant is used to report situations where the client tries to invoke a method that it is not allowed to access.

The `MySecurityService` class is a security service that is responsible for ensuring data integrity by verifying checksums on incoming commands and attaching checksums to outgoing commands. The program also requires the client to authenticate itself as the principal application provider or principal cardholder by sending a two-byte PIN.

The implementation of `Purse`, `SecurePurseImpl`, is similar to the non-secure case, however, at the beginning of each method call, there is a call to the security service which ensures that the business rules are satisfied and that the data is not corrupted.

The applet `SecurePurseApplet` is similar to the non-secure case, but it also creates and registers an instance of `MySecurityService`.

The client-side program, `SecurePurseClient`, is similar to the non-secure case, but instead of a generic card accessor, it uses its own implementation: `SecureOCFCardAccessor`, which performs additional pre- and post-processing of data and supports the additional command `authenticateUser`.

`SecurePurseClient` also requires verification of the user. After the applet is inserted, a PIN must be given to the card-side applet by calling `authenticateUser` on `SecureOCFCardAccessor`.

When `authenticateUser` is called, `SecureOCFCardAccessor` prepares and sends the following command:

TABLE 6 Authenticate User Command

CLA_AUTH	INS_AUTH	P1 field	P2 field	LC field	PIN (two bytes)	
0x80	0x39	0	0	2	xx	xx

On the card side, `MySecurityService` processes the command. If the PIN is correct, then the appropriate flags are set in the security service and a confirmation response is returned to the client. Once authentication is passed, the client program receives the balance, credits the account, and again receives the balance. The program demonstrates error handling when the client attempts to debit a number of units from the account. This will cause the program to throw a `UserException` with the code `REQUEST_DENIED`.

As with `RMIDemo` the client part of the `SecureRMIDemo` can be run without parameters or with the `-i` parameter:

- If the demo is run without parameters, remote references are identified using the class name of the remote object.
- If the demo is run with the `-i` parameter, remote references are identified using the list of remote interfaces implemented by the remote object.

For more information on these formats, see Chapter 8 of the *Runtime Environment (JCRE) Specification for the Java Card™ Platform, Version 2.2.1*.

▼ Running the Secure Java Card RMI Demo

`SecureRMIDemo` can be run on either the C-language Java Card RE or Java Card WDE. The `SecureRMI` demo applet is installed in the EEPROM image when you run `demo2`.

The `SecureRMIDemo` can be run in Java Card WDE by listing it on the first line of the applet configuration file `jcwde_securermi.app`.

If the run is successful, the output in the file will be the same as contained in file `securermidemo.scr.expected.out`.

To Run SecureRMIDemo on the C-Language Java Card RE:

1. **Run `cref` using the command:**

```
cref -i demoee
```

2. **Run the Secure Java Card RMI client program with either of these commands:**

```
securermidemo > securermidemo.scr.cref.out
securermidemo -i > securermidemo.scr.cref.out
```

To Run SecureRMIDemo on Java Card WDE:

1. **Navigate to the `jcwde` directory (this will be `$JC_HOME/samples/src/demo/jcwde` on the Solaris or Linux platform or `$JC_HOME\samples\src\demo\jcwde` on the Microsoft Windows 2000 platform). Run Java Card WDE using the command:**

```
jcwde jcwde_securermi.app
```

2. **In a separate command window, navigate to the `rmi` directory. This will be `$JC_HOME/samples/src/demo/demo2/rmi` directory (on the Solaris or Linux platform) or `%JC_HOME%\samples\src\demo\demo2\rmi` directory (on the Microsoft Windows 2000 platform). Run the Secure Java Card RMI client program with either of these commands:**

```
securermidemo > securermidemo.jcwde.out
securermidemo -i > securermidemo.jcwde.out
```

Object Deletion Demo 1

The Object Deletion Demo 1, `odDemo1`, demonstrates the object deletion mechanism, applet deletion, and package deletion. The `odDemo1` demonstration has three parts:

- `odDemo1-1.scr` demonstrates the object deletion mechanism and verifies that memory for objects referenced from transient memory of type `CLEAR_ON_DESELECT` has been reclaimed after an applet is deselected.

  `odDemo1-1.scr` does not depend on any other demo. The final state of `cref` memory must be saved to a file for `odDemo1-2.scr` to use.

- `odDemo1-2.scr` demonstrates the object deletion mechanism and verifies that memory for objects referenced from transient memory of type `CLEAR_ON_RESET` has been reclaimed after card reset.

  The `odDemo1-2.scr` demo must be run after `odDemo1-1.scr` because the initial state of `cref` must be the same as its final state after running `odDemo1-1.scr`. After running `odDemo1-2.scr`, the final state of `cref` must be saved to a file so it can be used by `odDemo1-3.scr`.

- `odDemo1-3.scr` performs applet deletion, package deletion, and employs the `AppletEvent.uninstall` method to uninstall an applet. The demo verifies that all transient memory of type `CLEAR_ON_RESET` and `CLEAR_ON_DESELECT` has been returned to the memory manager. The demo also demonstrates the use of the `AppletEvent.uninstall()` method.

  The `odDemo1-3.scr` demo must be run after `odDemo1-2.scr` because the initial state of `cref` must be the same as its final state after running `odDemo1-2.scr`.

▼ Running odDemo1

`odDemo1` runs only in the C-language Java Card RE. This is because the Java Card WDE does not support the object deletion mechanism, applet deletion, or package deletion.

1. **In a command window, run `cref` using this command:**

```
cref -o crefState
```

2. **In a second command window, navigate to the** `$JC_HOME/samples/src/demo/object_deletion` **directory (or** `%JC_HOME%\samples\src\demo\object_deletion` **on the Microsoft Windows 2000 platform) and run APDUTool, using this command:**

   ```
   apdutool -nobanner -noatr odDemo1-1.scr > odDemo1-1.scr.cref.out
   ```

   If the run is successful, the APDUTool log, `odDemo1-1.scr.cref.out` should be identical to the file `odDemo1-1.scr.expected.out`.

3. **In the first command window run** `cref` **using this command:**

   ```
   cref -i crefState -o crefState
   ```

4. **In the second command window, execute APDUTool using this command:**

   ```
   apdutool -nobanner -noatr odDemo1-2.scr > odDemo1-2.scr.cref.out
   ```

   If the run is successful, the apdutool log, `odDemo1-2.scr.cref.out` should be identical to the file `odDemo1-2.scr.expected.out`.

5. **In the first command window run** `cref` **using this command:**

   ```
   cref -i crefState
   ```

6. **In the second command window, execute APDUTool using this command:**

   ```
   apdutool -nobanner -noatr odDemo1-3.scr > odDemo1-3.scr.cref.out
   ```

   If the run is successful, the apdutool log, `odDemo1-3.scr.cref.out` should be identical to the file `odDemo1-3.scr.expected.out`.

Object Deletion Demo2

The Object Deletion Demo 2, `odDemo2`, demonstrates package deletion and checks that persistent memory has been returned to the memory manager. This demo has one script: `odDemo2.scr`. You do not have to run `odDemo1` to run `odDemo2`.

▼ Running odDemo2

`odDemo2` runs only in the C-language Java Card RE. This is because the Java Card WDE does not support the object deletion mechanism, applet deletion, or package deletion.

1. **In a command window, run** `cref` **using this command:**

   ```
   cref
   ```

2. **In a second window, navigate to the** `$JC_HOME/samples/src/demo/object_deletion` **directory (or** `%JC_HOME%\samples\src\demo\object_deletion` **on the Microsoft Windows 2000 platform) and run APDUTool with this command:**

   ```
   apdutool -nobanner -noatr odDemo2.scr > odDemo2.scr.cref.out
   ```

   If the run is successful, the apdutool log, `odDemo2.scr.cref.out` should be identical to the file `odDemo2.scr.expected.out`.

Logical Channels Demo

The Logical Channels Demo, `lcdemo`, demonstrates the behavior of Java Card technology-based logical channels by showing how two applets, which interact with each other, can each be selected for use at the same time.

The logical channels demo mimics the behavior of a wireless device connected to a network service. A connection manager tracks whether the device is connected to the service and whether the connection is local or remote. While it is connected, the user's account is debited on a unit of time basis; the debit rate is based on whether the connection is local or remote. The demo employs two applets to simulate this situation: the `ConnectionManager` applet manages the connection while the `AccountAccessor` applet manages the account.

When the user turns on the device, the `ConnectionManager` applet is selected. Every unit of time, the terminal sends a message containing the area code to the card. When the user wants to use the service, the `AccountAccessor` applet is selected on another logical channel so that the terminal can query the balance. The `AccountAccessor` can return the balance only if the `ConnectionManager` is active. The `ConnectionManager` applet sets the connection and tracks the connection status. Based on the value of an area code variable, the `ConnectionManager` determines whether the connection is local or remote. `AccountAccessor` uses this information to debit the account at the appropriate rate. The connection is disabled when the user completes the call or when the account is depleted.

▼ Running the Logical Channels Demo

The logical channels demo runs only in the C-language Java Card RE. No sample scripts or demos are provided to demonstrate this functionality on Java Card WDE.

1. **In a command window, run** `cref` **using this command:**

   ```
   cref
   ```

2. **In the second command window, navigate to the** `$JC_HOME/samples/src/demo/logical_channels` **directory (or** `%JC_HOME%\samples\src\demo\logical_channels` **on the Microsoft Windows 2000 platform) and execute APDUTool using this command:**

   ```
   apdutool -nobanner -noatr channelDemo.scr > channelDemo.scr.cref.out
   ```

   If the run is successful, the APDUTool log, `channelDemo.scr.cref.out` should be identical to the file `channelDemo.scr.expected.out`.

Demo 2 Cryptography Demo

The Demo 2 Cryptography Demo, `demo2crypto`, is similar to `demo2`, except that it employs a version of `JavaPurse` that uses a DES MAC algorithm. This version of `JavaPurse` is called `JavaPurseCrypto`. All other applets are exactly the same as were used in `demo2`.

Note – There are no cryptography versions of `demo1` or `demo3`.

A DES MAC is a cryptographic signature that uses DES encryption on all or part of a message (APDU). `JavaPurseCrypto` uses the DES MAC to verify several of the APDUs; that is, instead of zeros in the signature currently in `JavaPurse`, there will be a real signature that can be programmatically signed and verified. Other programs that interact with `JavaPurseCrypto` (such as `JavaLoyalty` and `Wallet`) are not affected since all signing/verifying of the signature will occur only within `JavaPurseCrypto`.

▼ Running the demo2crypto Demo

`demo2crypto` runs in the C-language Java Card RE because the Java Card WDE is not able to support the downloading of CAP files.

1. **Run** `cref` **using the following command:**

   ```
   cref
   ```

2. **In a second command window, navigate to the** `$JC_HOME/samples/src/demo/demo2` **directory (or** `%JC_HOME%\samples\src\demo\demo2` **on Windows) and execute APDUTool using this command:**

   ```
   apdutool -nobanner -noatr demo2crypto.scr > demo2crypto.scr.cref.out
   ```

   If the run is successful, the `apdutool` log, `demo2crypto.scr.cref.out`, should be identical to the file `demo2crypto.scr.expected.out`.

Photo Card Demo

The Photo Card Demo, `photocard`, illustrates how you can use the large address space available in the 32-bit version of the Java Card platform reference implementation, version 2.2.1. The demo uses the large address space of the smart card's EEPROM memory to store up to four GIF images. The images are included with the demo.

Running the Photo Card Demo

The Photo Card demo can be run only after `demo2` has successfully completed. This is because the Photo Card applet is downloaded with `demo2.scr`.

1. **Run `cref` with the `-z` option to display the memory statistics for the card:**

   ```
   cref -z -i demoee
   ```

2. **In a separate window, navigate to the `photocard` directory. This will be `$JC_HOME/samples/src/demo/demo2/photocard` directory (on the Solaris or Linux platform) or `%JC_HOME%\samples\src\demo\demo2\photocard` directory (on the Microsoft Windows 2000 platform). Run the `photocard` client program and specify the four supplied GIF images:**

   ```
   photocard duke_magnify.gif duke_pencil.gif duke_wave.gif
       duke_thumbsup.gif > photocard.scr.cref.out
   ```

   If the run is successful, the output in the file `photocard.scr.cref.out` will be the same as contained in file `photocard.scr.expected.out`.

3. **Perform a diff on the individual images to ensure that their contents have not changed.**

CHAPTER 4

Running Applets in an Emulated Card Environment

The Java Card platform Workstation Development Environment ("Java Card Workstation Development Environment" or "Java Card WDE") tool allows the simulated running of a Java Card applet as if it were masked in ROM. It emulates the card environment.

The Java Card WDE is not an implementation of the Java Card virtual machine. It uses the Java virtual machine to emulate the Java Card RE. Class files that represent masked packages must be available on the classpath for Java Card WDE.

For the 2.2.1 release of the Java Card reference implementation, Java Card WDE adds support for Java Card Remote Method Invocation (Java Card RMI).

Here are some of Java Card RE features that are *not* supported by Java Card WDE:

- package installation
- applet instance creation
- persistent card state
- firewall
- transactions
- transient array clearing
- object deletion
- applet deletion
- package deletion

The Java Card WDE tool uses the `jcwde.jar`, `api.jar` (with cryptography extensions), and `apduio.jar` files. The main class for Java Card WDE is `com.sun.javacard.jcwde.Main`. A sample batch and shell script are provided to start Java Card WDE.

Preparing to Run the Java Card WDE Tool

Before you run the Java Card WDE tool, you must ensure that the environment variables are set appropriately and the applets to be configured are listed in a configuration file.

Setting Environment Variables

To set the environment variables correctly, refer to "Setting Environment Variables for the Solaris or Linux Platform" on page 5 or "Setting Environment Variables for Microsoft Windows 2000 Platform" on page 7.

Configuring the Applets in the Java Card WDE Mask

The applets to be configured in the mask during Java Card WDE simulation need to be listed in a configuration file that is passed to the Java Card WDE as a command line argument. Also, the `CLASSPATH` environment variable needs to be set to reflect the location of the class files for the applets to be simulated. In this release, the sample applets are listed in a configuration file called `jcwde.app`. Each entry in this file contains the name of the applet class, and its associated AID.

The configuration file contains one line per installed applet. Each line is a white space(s) separated `{CLASS_NAME AID}` pair, where `CLASS NAME` is the fully qualified Java name of the class defining the applet, and `AID` is an Application Identifier for the applet class used to uniquely identify the applet. `AID` may be a string or hexadecimal representation in form:

`0x`*`XX`*`[:0x`*`XX`*`]`

where the construct `0xXX` is repeated as many times as necessary.

Note that `AID` should be 5 to 16 bytes in length.

For example:

```
com.sun.javacard.samples.wallet.Wallet
    0xa0:0x0:0x0:0x0:0x0:0x62:0x3:0x1:0xc:0x6:0x1
```

Note – The installer applet must be listed first in the Java Card WDE configuration file.

If you write your own applets for public distribution, you should obtain an AID for each of your packages and applets according to the directions in Section 4.2 of the *Virtual Machine Specification for the Java Card™ Platform, Version 2.2.1* (Sun Microsystems, Inc., 2003), and in the *ISO 7816 Specification* Parts 1-6.

Running the Java Card WDE Tool

The general format of the command to run the Java Card WDE and emulate the Java Card RE is:

```
jcwde [-help] [-p port] [-version] [-nobanner] <config-file>
```

where:

TABLE 7 Command Line Options for Java Card WDE

Option	Description
<config-file>	The configuration file described above.
`-help`	Prints a help message.
`-nobanner`	Suppresses all banner messages.
`-p`	Allows you to specify a TCP/IP port other than the default port.
`-version`	Prints the Java Card WDE version number.

When started, Java Card WDE starts listening to APDUs in T=0 format on the TCP/IP port specified by the `-p` port parameter. The default port is 9025.

CHAPTER 5

Converting Java Class Files

The Converter processes class files that make up a Java package. In addition to class files, the Converter can process either version 2.2.*x*, or 2.1.*x* export files. Depending on the command line options, the Converter outputs a CAP file, a Java Card Assembly file, and an export file.

The CAP file is a JAR-format file which contains the executable binary representation of the classes in a Java package. The CAP file also contains a manifest file that provides human-readable information regarding the package that the CAP file represents. For more information on the manifest file and its contents, see Appendix B "CAP File Manifest File Syntax". For more information on the CAP file and its format, see Chapter 6 of the *Virtual Machine Specification for the Java Card™ Platform, Version 2.2.1*.

Note – For more information on the Java Card Assembly file, see Appendix A "Java Card Assembly Syntax Example".

The Converter verifies that class files comply to limitations described in Section 2.2, "Java Card Platform Language Subset" in the *Virtual Machine Specification for the Java Card™ Platform, Version 2.2.1*. It also checks the correctness of export files.

You are responsible for the consistency of your input data. This means that:

- all input class files are compatible with each other.
- export files of imported packages are consistent with class files that were used for compiling the converting package.

If the package to be converted contains remote classes or interfaces, the Converter generates a CAP file for version 2.2.*x* of the Java Card platform, a Java Card Assembly file and an export file. If the package does not contain remote classes or interfaces, the Converter generates files that can be used by version 2.1 of the Java Card platform. To create a CAP file compatible with version 2.1 of the Java Card platform, you must use export files for Java Card API packages from the Java Card Development Kit 2.1.x.

Setting Java Compiler Options

For the most efficient conversion, compile your class files with the SDK Java compiler's `-g` command line option. The `-g` option causes the compiler to generate the `LocalVariableTable` attribute in the class file. The Converter uses this attribute to determine local variable types. If you do not use the `-g` option, the Converter attempts to determine the variable types on its own. This is expensive in terms of processing and might not produce the most efficient code.

Do not compile with the `-O` option. The `-O` option is not recommended on the Java compiler command line, for these reasons:

- this option is intended to optimize execution speed rather than minimize memory usage. Minimizing memory usage is much more important in the Java Card environment.
- the `LocalVariableTable` attribute will not be generated.

Generating the CAP File's Debug Component

If you want to use the Converter's `-debug` option to generate a debug component in the CAP file, then you must first compile your class files with `-g`.

Running the Converter

Command line usage of the Converter is:

```
converter [options]  <package_name> <package_aid> <major_version>.
   <minor_version>
```

The file to invoke the Converter is a shell script (`converter`) on the Solaris or Linux platform, and a batch file (`converter.bat`) on the Microsoft Windows 2000 platform.

Command Line Arguments

The arguments to this command line are:

TABLE 8 Converter Command Line Arguments

Option	Description
<*package_name*>	Fully-qualified name of the package to convert.
<*package_aid*>	5- to 16-decimal, hex or octal numbers separated by colons. Each of the numbers must be byte-length.
<*major_version*>. <*minor_version*>	User-defined version of the package.

Command Line Options

The options in this command line are:

TABLE 9 Converter Command Line Options

Option	Description
`-applet` <*AID*> <*class_name*>	Sets the default applet AID and the name of the class that defines the applet. If the package contains multiple applet classes, this option must be specified for each class.
`-classdir` <*root directory of the class hierarchy*>	Sets the root directory where the Converter will look for classes. If this option is not specified, the Converter uses the current user directory as the root.
`-d` <*root directory for output*>	Sets the root directory for output.
`-debug`	Generates the optional debug component of a CAP file. If the `-mask` option is also specified, the file `debug.msk` will be generated in the output directory. **Note—**To generate the debug component, you must first compile your class files with the Java compiler's `-g` option.
`-exportmap`	Uses the token mapping from the pre-defined export file of the package being converted. The Converter will look for the export file in the exportpath.
`-exportpath` <*list of directories*>	Specifies the root directories in which the Converter will look for export files. The separator character for multiple paths is platform dependent. It is semicolon (`;`) for the Microsoft Windows 2000 platform and colon (`:`) for the Solaris or Linux platform. If this option is not specified, the Converter sets the export path to the Java `classpath`.

TABLE 9 Converter Command Line Options

Option	Description
`-help`	Prints help message.
`-i`	Instructs the Converter to support the 32-bit integer type.
`-mask`	Indicates this package is for a mask, so restrictions on native methods are relaxed.
`-nobanner`	Suppresses all banner messages.
`-noverify`	Suppresses the verification of input and output files. For more information on file verification, see "Verification of Input and Output Files" on page 40.
`-nowarn`	Instructs the Converter not to report warning messages.
`-out` [*CAP*] [*EXP*] [*JCA*]	Instructs the Converter to output the CAP file, and/or the export file, and/or the Java Card Assembly file. By default (if this option is not specified), the Converter outputs a CAP file and an export file.
`-v`, `-verbose`	Enables verbose output. Verbose output includes progress messages, such as "opening file", "closing file", and whether the package requires integer datatype support.
`-V`, `-version`	Prints the Converter version string.

Note – The `-out CAP` and `-mask` options cannot be used together.

Using Delimiters with Command Line Options

If the command line option argument contains a space symbol, you must use delimiters with this argument. The delimiter for the Solaris or Linux platform is a backslash and double quote (`\"`); the delimiter for Microsoft Windows 2000 platform is a double quote (`"`).

In the following sample command line, the converter will check for export files in the `.\export files`, `.\jc22\api_export_files`, and current directories.

For the Solaris or Linux platform:

```
converter -exportpath \"./export files:.:./jc22/api_export_files\"
   MyWallet 0xa0:0x00:0x00:0x00:0x62:0x12:0x34 1.0
```

For the Microsoft Windows 2000 platform:

```
converter -exportpath ".\export files;.;.\jc22\api_export_files"
   MyWallet 0xa0:0x00:0x00:0x00:0x62:0x12:0x34 1.0
```

Using a Command Configuration File

Instead of entering all of the command line arguments and options on the command line, you can include them in a text-format configuration file. This is convenient if you frequently use the same set of arguments and options.

The syntax to specify a configuration file is:

```
converter -config <configuration file name>
```

The `<configuration file name>` argument contains the file path and file name of the configuration file.

For Solaris, Linux, and Microsoft Windows 2000 operating systems, you must use double quote (") delimiters for the command line options that require arguments in the configuration file. For example, if the options from the command line example used in "Using Delimiters with Command Line Options" were placed in a configuration file, the result would look like this:

Solaris or Linux platform

```
-exportpath "./export files:.:./jc22/api_export_files"
   MyWallet 0xa0:0x00:0x00:0x00:0x62:0x12:0x34 1.0
```

Microsoft Windows 2000 platform

```
-exportpath ".\export files;.;.\jc22\api_export_files"
   MyWallet 0xa0:0x00:0x00:0x00:0x62:0x12:0x34 1.0
```

File and Directory Naming Conventions

This section describes the names of input and output files for the Converter, and gives the correct location for these files. With some exceptions, the Converter follows the Java naming conventions for default directories for input and output files. These naming conventions are also in accordance with the definitions in § 4.1 of the *Virtual Machine Specification for the Java Card™ Platform, Version 2.2.1* (Sun Microsystems, Inc. , 2003).

Input File Naming Conventions

The files input to the Converter are Java class files named with the `.class` suffix. Generally, there are several class files making up a package. All the class files for a package must be located in the same directory under the root directory, following

the Java naming conventions. The root directory can be set from the command line using the `-classdir` option. If this option is not specified, the root directory defaults to be the directory from which the user invoked the Converter.

Suppose, for example, you wish to convert the package `java.lang`. If you use the `-classdir` flag to specify the root directory as `C:\mywork`, the command line will be:

```
converter -classdir C:\mywork java.lang <package_aid>
    <package_version>
```

where *<package_aid>* is the application ID of the package, and *<package_version>* is the user-defined version of the package.

The Converter will look for all class files in the `java.lang` package in the directory `C:\mywork\java\lang`

Output File Naming Conventions

The name of the CAP file, export file, and the Java Card Assembly file must be the last portion of the package specification followed by the extensions `.cap`, `.exp`, and `.jca`, respectively.

By default, the files output from the Converter are written to a directory called `javacard`, a subdirectory of the input package's directory.

In the above example, the output files are written by default to the directory `C:\mywork\java\lang\javacard`

The `-d` flag allows you to specify a different root directory for output.

In the above example, if you use the `-d` flag to specify the root directory for output to be `C:\myoutput`, the Converter will write the output files to the directory `C:\myoutput\java\lang\javacard`.

When generating a CAP file, the Converter creates a Java Card Assembly file in the output directory as an intermediate result. If you do not want a Java Card Assembly file to be produced, then omit the option `-out JCA`. The Converter deletes the Java Card Assembly file at the end of the conversion.

Verification of Input and Output Files

By default, the converter invokes the Java Card technology-based off-card verifier ("Java Card off-card verifier") for every input EXP file and on the output CAP and EXP files.

- If any of the input EXP files do not pass verification, then no output files are created.
- If the output CAP or EXP files does not pass verification, then the output EXP and CAP files are deleted.

If you want to bypass verification of your input and output files, use the `-noverify` command line option. Note that if the converter finds any errors, output files will not be produced.

Creating a debug.msk Output File

If you select the `-mask` and `-debug` options, the file `debug.msk` is created in the same directory as the other output files. (Refer to "Command Line Options" on page 37.)

Loading Export Files

A Java Card technology-based export file ("Java Card export file") contains the public API linking information of classes in an entire package. The Unicode string names of classes, methods and fields are assigned unique numeric tokens.

Export files are not used directly on a device that implements a Java Card virtual machine. However, the information in an export file is critical to the operation of the virtual machine on a device. An export file is produced by the Converter when a package is converted. This package's export file can be used later to convert another package that imports classes from the first package. Information in the export file is included in the CAP file of the second package, then is used on the device to link the contents of the second package to items imported from the first package.

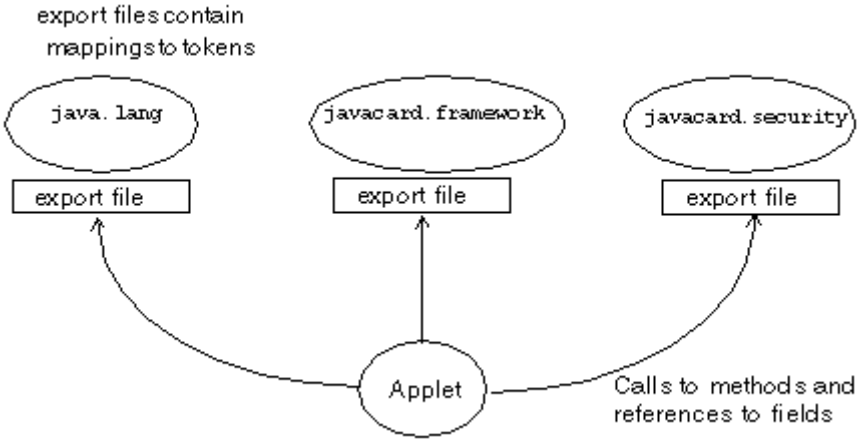
During the conversion, when the code in the currently-converted package references a different package, the Converter loads the export file of the different package.

FIGURE 2 illustrates how an applet package is linked with the `java.lang`, the `javacard.framework` and `javacard.security` packages via their export files.

You can use the `-exportpath` command option to specify the locations of export files. The path consists of a list of root directories in which the Converter looks for export files. Export files must be named as the last portion of the package name followed by the extension `.exp`. Export files are located in a subdirectory called `javacard`, following the Java Card platform's directory naming convention.

For example, to load the export file of the package `java.lang`, if you have specified `-exportpath` as `c:\myexportfiles`, the Converter searches the directory `c:\myexportfiles\java\lang\javacard` for the export file `lang.exp`.

FIGURE 2 Calls between packages go through the export files



Specifying an Export Map

You can request the Converter to convert a package using the tokens in the pre-defined export file of the package that is being converted. Use the `-exportmap` command option to do this.

There are two distinct cases when using the `-exportmap` flag: when the minor version of the package is the same as the version given in the export file (this case is called package reimplementation) and when the minor version increases (package upgrading). During the package reimplementation the API of the package (exportable classes, interfaces, fields and methods) must remain exactly the same. During the package upgrade, changes that do not break binary compatibility with preexisting packages are allowed (See "Binary Compatibility" in Section 4.4 of the *Virtual Machine Specification for the Java Card™ Platform, Version 2.2.1*).

For example, if you have developed a package and would like to reimplement a method (package reimplementation) or upgrade the package by adding new API elements (new exportable classes or new public or protected methods or fields to already existing exportable classes), you must use the `-exportmap` option to preserve binary compatibility with already existing packages that use your package.

The Converter loads the pre-defined export file in the same way that it loads other export files.

CHAPTER 6

Viewing an Export File

The `exp2text` tool is provided to allow you to view any export file in text format.

`exp2text [options] <package_name>`

Where `options` include:

TABLE 10 exp2txt Command Line Options

Option	Description
`-classdir` <*input root directory*>	Specifies the root directory where the program looks for the export file.
`-d` <*output root directory*>	Specifies the root directory for output.
`-help`	Prints help message.

CHAPTER 7

Verifying CAP and Export Files

Off-card verification provides a means for evaluating CAP and export files in a desktop environment. When applied to the set of CAP files that will reside on a Java Card-compliant smart card and the set of export files used to construct those CAP files, the Java Card technology-enabled off-card verifier ("Java Card off-card verifier") provides the means to assert that the content of the smart card has been verified.

The off-card verifier is a combination of three tools. Use these tools for:

- Verifying CAP Files
- Verifying Export Files
- Verifying Binary Compatibility

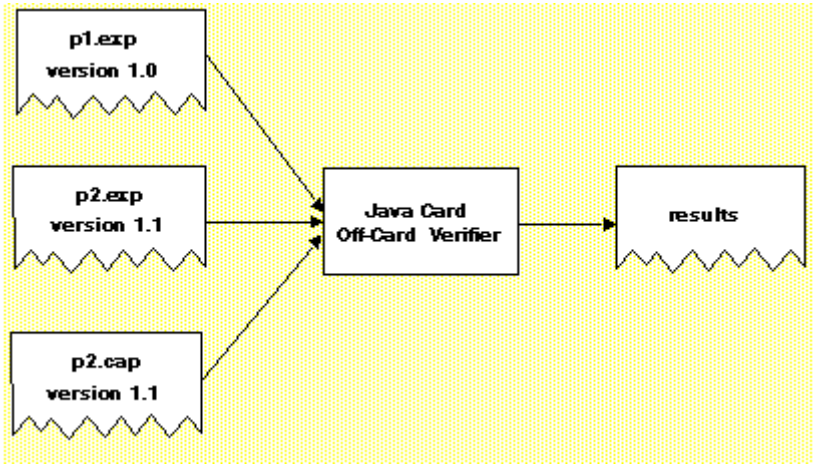
The names of the tools that perform these verifications are `verifycap`, `verifyexp`, and `verifyrev`, respectively. The following sections describe how to use each tool.

Verifying CAP Files

The `verifycap` tool is used to verify a CAP file within the context of package's export file (if any) and the export files of imported packages. This verification confirms whether a CAP file is internally consistent, as defined in Chapter 6 of the *Virtual Machine Specification for the Java Card™ Platform, Version 2.2.1*, and consistent with a context in which it can reside in a Java Card technology-enabled device.

Each individual export file is verified as a single unit. The scenario is shown in FIGURE 3. In the figure, the package `p2` CAP file is being verified. Package `p2` has a dependency on package `p1`, so the export file from package `p1` is also input. The `p2.exp` file is only required if `p2.cap` exports any of its elements.

FIGURE 3 Verifying a CAP file



Running verifycap

Command line usage is:

`verifycap [`*`options`*`] <`*`export files`*`> <`*`CAP file`*`>`

The file to invoke `verifycap` is a shell script (`verifycap`) on the Solaris or Linux platform and a batch file (`verifycap.bat`) on the Microsoft Windows 2000 platform.

Command Line Arguments

The arguments to this command line are:

TABLE 11 verifycap Command Line Arguments

Argument	Description
<export files>	A list of export files of the packages that this CAP file uses.
<CAP file>	Name of the CAP file to be verified.

Command Line Options

For a description of the command line options available for `verifycap`, see "Command Line Options for Off-Card Verifier Tools" on page 52.

Verifying Export Files

The `verifyexp` tool is used to verify an export file as a single unit. This verification is "shallow," examining only the content of a single export file, not including export files of packages referenced by the package of the export file. The verification determines whether an export file is internally consistent and viable as defined in Chapter 5 of the *Virtual Machine Specification for the Java Card™ Platform, Version 2.2.1*. This scenario is illustrated in FIGURE 4.

FIGURE 4 Verifying an export file

Running verifyexp

Command line usage is:

`verifyexp [options] <export file>`

The file to invoke `verifyexp` is a shell script (`verifyexp`) on the Solaris or Linux platform and a batch file (`verifyexp.bat`) on the Microsoft Windows 2000 platform.

Command Line Arguments

The argument to this command line is:

TABLE 12 verifyexp Command Line Argument

Argument	Description
<export file>	Fully qualified path and name of the export file.

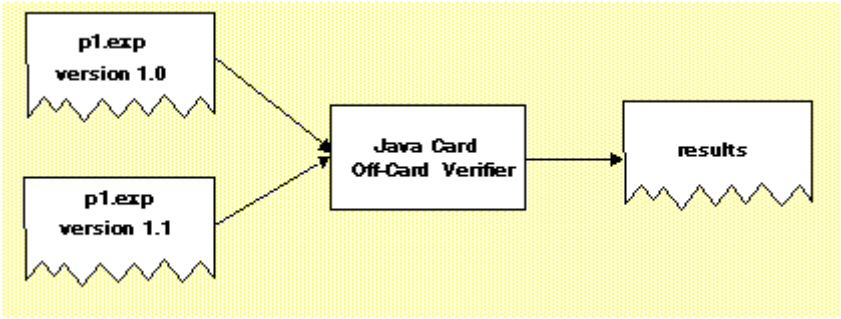
Command Line Options

For a description of the command line options available for `verifyexp`, see "Command Line Options for Off-Card Verifier Tools" on page 52.

Verifying Binary Compatibility

The `verifyrev` tool checks for binary compatibility between revisions of a package by comparing the respective export files. This scenario is illustrated in FIGURE 5. The export files from version 1.0 and 1.1 of package `p1` are input to `verifyrev`. The verification examines whether the Java Card platform version rules, including those imposed for binary compatibility as defined in Section 4.4 of the *Virtual Machine Specification for the Java Card™ Platform, Version 2.2.1*, have been followed.

FIGURE 5 Verifying binary compatibility of export files



Running verifyrev

Command line usage is:

```
verifyrev [options] <export file> <export file>
```

The file to invoke `verifyrev` is a shell script (`verifyrev`) on the Solaris or Linux platform and a batch file (`verifyrev.bat`) on the Microsoft Windows 2000 platform.

Command Line Arguments

The arguments to this command line are:

<export file> <export file>

Where *<export file>* represents the fully qualified path of the export files to be compared.

The second export file name must be the same as the first one with a different path. For example,

```
verifyrev d:\testing\old\crypto.exp d:\testing\new\crypto.exp
```

Command Line Options

For a description of the command line options available for `verifyrev`, see "Command Line Options for Off-Card Verifier Tools".

Command Line Options for Off-Card Verifier Tools

The `verifycap`, `verifyexp`, and `verifyrev`, off-card verifier tools share many of the same command line options. The only exception is the `-package` option which is available for `verifycap` only.

These options exhibit the same behavior regardless of the tool that calls them.

TABLE 13 verifycap, verifyexp, verifyrev Command Line Options

Option	Description
`-help`	Prints help message.
`-nobanner`	Suppresses banner message.
`-nowarn`	Suppresses warning messages.
`-package` <*package name*>	*(Available for verifycap only)* Sets the name of the package to be verified.
`-verbose`	Enables verbose mode.
`-version`	Prints version number and exit.

CHAPTER 8

Generating a CAP File from a Java Card Assembly File

Use the `capgen` tool to generate a CAP file from a given Java Card Assembly file. The CAP file which is generated has the same contents as a CAP file produced by the Converter. The `capgen` tool is a backend to the Converter.

Running capgen

The file to invoke `capgen` is a shell script (`capgen`) on the Solaris or Linux platform, and a batch file (`capgen.bat`) on the Microsoft Windows 2000 platform.

Command line syntax for `capgen` is:

`capgen [-`*`options`*`] <`*`filename`*`>`

where <*`filename`*> is the Java Card Assembly file.

Command Line Options

The option values and their actions are:

TABLE 14 capgen Command Line Options

Option	Description
`-help`	Prints a help message.
`-nobanner`	Suppresses all banner messages.
`-o` <*filename*>	Allows you to specify an output file. If the output file is not specified with the `-o` flag, output defaults to the file `a.jar` in the current directory.
`-version`	Outputs the version information.

CHAPTER 9

Producing a Text Representation of a CAP File

Use the `capdump` tool to produce an ASCII representation of a CAP file.

Running capdump

The file to invoke `capdump` is a shell script (`capdump`) on the Solaris or Linux platform, and a batch file (`capdump.bat`) on the Microsoft Windows 2000 platform.

Command line usage of `capdump` is:

`capdump <filename>`

where `<filename>` is the CAP file.

Output from this command is always written to standard output.

There are no command line options to `capdump`.

CHAPTER 10

Producing a Mask File from Java Card Assembly Files

The `maskgen` tool produces a mask file from a set of Java Card Assembly files produced by the Converter. The format of the output mask file is targeted to a specific platform. The plugins that produce each different `maskgen` output format are called generators. The supported generators are `cref`, which supports the C-language Java Card RE, and `size`, which reports size statistics for the mask. Other generators, which are not supported in this release, include `jref`, which supports the Java programming language Java Card RE and `a51`, which supports the Keil A51 assembly language interpreter.

For more information on the contents of a Java Card Assembly file, see Appendix A "Java Card Assembly Syntax Example".

Command Line for maskgen

The file to invoke `maskgen` is a shell script (`maskgen`) on the Solaris or Linux platform, and a batch file (`maskgen.bat`) on the Microsoft Windows 2000 platform. Usage is:

```
maskgen [options] <generator> <filename> [ <filename> ...]
```

Command Line Arguments

TABLE 15 Command Line Arguments for the maskgen tool

Argument	Description
<*generator*>	Specifies the generator, the plugin that formats the output for a specific target platform. The generators are: • `a51`–Output for the Keil A51 assembly language interpreter (not supported for this release). • `cref`–Output for the C-language Java Card RE interpreter. • `jref`–Output for the Java programming language Java Card RE interpreter (not supported for this release). • `size`–Outputs mask size statistics. In this release, the only supported generator is `cref`.
<*filename*> [<*filename*>...]	Any number of Java Card Assembly files can be input to `maskgen` as a whitespace-separated list. You can also create a text file containing a list of Java Card Assembly file names for a new mask, and prepend an "@" character to the name of this text file as an argument to `maskgen`.

Order of Packages on the Command Line

The Java Card Assembly files that can be listed on the command line can belong to API packages, the installer package, or the user's library and applet packages. The Java Card Assembly files that belong to API packages must be listed first on the command line, followed by the Java Card Assembly files belonging to any applets.

If you include the installer package's Java Card Assembly file on the command line, it must be listed after all of the Assembly files belonging to API packages and before the Assembly files of any other applet packages.

For example:

```
maskgen -nobanner cref API_package_1.jca .... API_package_n.jca
    installer_package.jca applet_package_1.jca ... applet_package_n.
    jca
```

Version Numbers for Processed Packages

The packages that you specify to generate a mask can import other packages. These imported packages must share the same major and minor version number as the specified packages.

For example, presume that you are using `Package A`, version 1.1 to create a mask, and that `Package A` imports `Package B`, version 1.1. Then you must ensure that `Package B`, version 1.1 is listed in the `import` component of the `Package A .jca` file.

Command Line Options

The command line options recognized by `maskgen` are:

TABLE 16 maskgen Command Line Options

Option	Description
`-16bit`	Generates a version of the mask that can be used with 16-bit applications.
`-32bit`	Generates the default version of the mask that can be used with 32-bit applications.
`-c <`*`filename`*`>`	Specifies a configuration file, which contains generator-specific settings. For example, the following line maps a native Java Card API method to a native label: `javacard/framework/JCSystem/beginTransaction()V=beginTransaction_NM` `cref_mask.cfg` and `cref_mask16.cfg` are examples of maskgen configuration files.
`-debuginfo`	Generates debug information for the generated mask. This option is available only with the `jref` generator.
`-help`	Prints a help message.
`-nobanner`	Suppresses all banner messages.
`-o <`*`filename`*`>`	Specifies the file name output from `maskgen`. If the output file is not specified, output defaults to `a.out`.
`-version`	Prints the version number of `maskgen`, then exits.

maskgen Example

This example uses a text file (`args.txt`) to pass command line arguments to `maskgen`:

```
maskgen -o mask.c cref @args.txt
```

where the contents of the file `args.txt` is:

```
first.jca second.jca third.jca
```

This is equivalent to the command line:

```
maskgen -o mask.c cref first.jca second.jca third.jca
```

This command produces an output file `mask.c` that is compiled with a C compiler to produce `mask.o`, which is linked with the C-language Java Card RE interpreter.

(Refer to Chapter 11 "Using the Java Card Reference Implementation" for more information about this target platform.)

CHAPTER 11

Using the Java Card Reference Implementation

The Java Card reference implementation is written in the C programming language and is called the *C-language Java Card Runtime Environment ("C-language Java Card RE")*. It is a simulator that can be built with a ROM mask, much like a real Java Card technology-based implementation. It has the ability to simulate persistent memory (EEPROM), and to save and restore the contents of EEPROM to and from disk files. Applets can be installed in the C-language Java Card RE. The C-language Java Card RE performs I/O via a socket interface, using the TLP-224 protocol, simulating a Java Card technology-compliant smart card in a card reader (CAD).

The C-language Java Card RE supports the following:

- use of up to three logical channels
- `integer` data type
- object deletion
- card reset in case of object allocation during an aborted transaction

In version 2.2.1 of the Development Kit, the C-language Java Card RE is available as a 32-bit implementation. The 32-bit implementation gives you the ability to go beyond the 64KB memory access limitation that was present in previous releases. The 2.2.1 release does provide a 16-bit version of the C-language Java Card RE for backward compatibility with older applications.

Running the Java Card Runtime Environment

The 32-bit implementation of the C-language Java Card RE is supplied as prebuilt executables.

TABLE 17 Name and Location of cref Executables

File Name	Description
`%JC_HOME%/bin/cref.exe`	32-bit implementation of `cref` for the Microsoft Windows 2000 platform.
`$JC_HOME\bin\cref`	32-bit implementation of `cref` for the Solaris or Linux platform.

Installer Mask

The Development Kit Installer, the Java Card virtual machine interpreter, and the Java Card platform framework are built into the installer mask. It can be used as-is to load and run applets. Other than the Installer, it does not contain any applets.

The C-language Java Card RE requires no other files to start proper interpretation and execution of the mask image's Java Card bytecode.

Runtime Environment Command Line

Command line usage of C-language Java Card RE is the same on the Solaris, Linux, and Microsoft Windows 2000 platforms. The syntax is:

`cref [`*`options`*`]`

The output of the simulation is logged to standard output, which can be redirected to any desired file. The output stream can range from nothing, to very verbose, depending on the command line options selected.

Command-line Options

The options are case-sensitive.

TABLE 18 Runtime Environment Command Line Options

Option	Description
`-b`	Dumps a Byte Code Histogram at the end of the execution.
`-e`	Displays the program counter and stack when an exception occurs.
`-h`, `-help`	Prints a help screen.
`-i` <*input filename*>	Specifies a file to initialize EEPROM. Under the Solaris, Linux, and Microsoft Windows 2000 operating systems, file names must be single part--that is, there can be no spaces in the file name.
`-n`	Performs a trace display of the native methods that are invoked.
`-nobanner`	Suppresses the printing of a program banner.
`-nomeminfo`	Suppresses the printing of memory statistics when execution starts.
`-o` <*output filename*>	Saves the EEPROM contents to the named file. Under the Solaris, Linux, and Microsoft Windows 2000 operating systems, file names must be single part—that is, there can be no spaces in the file name.
`-p` <*port number*>	Connects to a TCP/IP port using the specified port number.
`-s`	Suppresses output. Does not create any output unless followed by other flag options.
`-t`	Performs a line-by-line trace display of the mask's execution.
`-version`	Prints only the program's version number. Do not execute.
`-z`	Prints the resource consumption statistics.

Obtaining Resource Consumption Statistics

The C-language Java Card RE provides a command line option (`-z`) for printing resource consumption statistics. This option enables the C-language Java Card RE to print statistics regarding memory usage once at startup and once at shutdown. Although memory usage statistics will vary among Java Card RE implementations, this option provides the applet developer with a general idea of the amount of memory needed to install and execute an applet.

The following output is obtained by running the `demo2` demonstration program with the `-z` command line option.

```
cref -z
```

```
Java Card platform version 2.2.1 C Reference Implementation Simulator (version
    0.41)
32-bit Address Space implementation - no cryptography support
Copyright 2003 Sun Microsystems, Inc. All rights reserved.

Memory configuration
        Type    Base    Size    Max Addr
        RAM     0x0     0x500   0x4ff
        ROM     0x2000  0xa000  0xbfff
        E2P     0x10020 0xffe0  0x1ffff

        ROM Mask size =                 0x566b =        22123 bytes
        Highest ROM address in mask =   0x766a =        30314 bytes
        Space available in ROM =        0x4995 =        18837 bytes
Mask has now been initialized for use
        0 bytecodes executed.
        Stack size: 00384 (0x0180) bytes,          00000 (0x0000) maximum used
        EEPROM use: 05935 (0x172f) bytes consumed, 59569 (0xe8b1) available
Transaction buffer: 00000 (0x0000) bytes consumed, 02560 (0x0a00) available
Clear-On-Reset RAM: 00000 (0x0000) bytes consumed, 00256 (0x0100) available
Clear-On-Dsel. RAM: 00000 (0x0000) bytes consumed, 00128 (0x0080) available
C-language Java Card RE was powered down.
   891495 bytecodes executed.
        Stack size: 00384 (0x0180) bytes,          00244 (0x00f4) maximum used
        EEPROM use: 14839 (0x39f7) bytes consumed, 50665 (0xc5e9) available
Transaction buffer: 00000 (0x0000) bytes consumed, 02560 (0x0a00) available
Clear-On-Reset RAM: 00168 (0x00a8) bytes consumed, 00088 (0x0058) available
Clear-On-Dsel. RAM: 00026 (0x001a) bytes consumed, 00102 (0x0066) available
```

The `demo2` demonstration program downloads and installs several applets and performs several transactions using a subset of the installed applets. Statistics are provided regarding the following resources: EEPROM, transaction buffer, stack usage, clear-on-reset RAM, and clear-on-deselect RAM. The statistics are printed twice, once at C-language Java Card RE start up and once when it shuts down.

This particular example shows the resources used to download and install a set of applications and execute several transactions. More fine-grained statistics could be obtained by limiting the actions during a single session. For example, using a single session to download one application would provide information regarding the resources needed to process the application download. The EEPROM contents at the end of the session could be saved using the `-o` option, and subsequent sessions could be used to measure resource usage for other actions, such as applet installation and execution.

In addition to the command line option, the Java Card API provides programmatic mechanisms for determining resource usage. For more information on these mechanisms, see the `javacard.framework.JCSystem.getAvailableMemory()` method in the *Application Programming Interface for the Java Card™ Platform, Version 2.2.1* .

Reference Implementation Limitations

- The maximum number of remote references that can be returned during one card session is 8.
- The maximum number of remote objects that can be exported simultaneously is 16.
- The maximum number of parameters of type `array` that can be used in remote methods is 8.
- The maximum number of Java Card API packages that the C-language Java Card RE can support is 32.
- The maximum number of library packages that a Java Card system can support is 32.
- The maximum number of applets that a Java Card system can support is 16.

Input and Output

The C-language Java Card RE performs I/O via a socket interface, using the TLP-224 protocol, simulating a Java Card technology-compliant smart card in a card reader (CAD). Use `apdutool` to read script files and send APDUs via a socket to the C-language Java Card RE. See "apdutool Examples" on page 90 for details. Note that you can have the C-language Java Card RE running on one workstation and run `apdutool` on another workstation.

Working with EEPROM Image Files

You can save the state of EEPROM contents, then load it in a later invocation of the C-language Java Card RE. To do this, specify an EEPROM image or "store" file to save the EEPROM contents.

Use the `-i` and `-o` flags to manipulate EEPROM image files at the `cref` command line:

- The `-i` flag, followed by a filename, specifies the initial EEPROM image file that will initialize the EEPROM portion of the virtual machine before Java Card virtual machine bytecode execution begins.

- The `-o` flag, followed by a filename, saves the updated EEPROM portion of the virtual machine to the named file, overwriting any existing file of the same name.

The `-i` and `-o` flags do not conflict with the performance of other option flags. File names used with the `-i` and `-o` flags flags must not contain spaces.

The commit of EEPROM memory changes during the execution of the C-language Java Card RE is not affected by the `-o` flag. Neither standard nor error output is written to the output file named with the -o option.

The following examples show how the `-i` and `-o` option flags can be used in a variety of useful execution scenarios.

Input EEPROM Image File

```
cref -i e2save
```

The C-language Java Card RE attempts to initialize simulated EEPROM from the EEPROM image file named `e2save`. No output file will be created.

Output EEPROM Image File

```
cref -o e2save
```

The C-language Java Card RE writes EEPROM data to the file `e2save`. The file will be created if it does not currently exist. Any existing EEPROM image file named `e2save` is overwritten.

Same Input and Output EEPROM Image File

```
cref -i e2save -o e2save
```

The C-language Java Card RE attempts to initialize simulated EEPROM from the EEPROM image file named `e2save`, and during processing, saves the contents of EEPROM to `e2save`, overwriting the contents. This behavior is much like a real Java Card technology-compliant smart card in that the contents of EEPROM is persistent.

Different Input and Output EEPROM Image Files

```
cref -i e2save_in -o e2save_out
```

The C-language Java Card RE attempts to initialize simulated EEPROM from the EEPROM image file named `e2save_in`, and during C-language Java Card RE processing, writes EEPROM updates to a EEPROM image file named `e2save_out`.

The output file will be created if it does not exist. Using different names for input and output EEPROM image files eliminates much potential confusion. This command line can be executed multiple times with the same results.

Note – Be careful naming your EEPROM image files. The C-language Java Card RE will overwrite an existing file specified as an output EEPROM image file. This can, of course, cause a problem if there is already an identically named file with a different purpose in the same directory.

The Default ROM Mask

Version 2.2.1 of the Java Card reference implementation provides a 32-bit version of the C-language Java Card RE executable: `cref.exe` for the Microsoft Windows 2000 platform, and `cref` for the Solaris or Linux platform. These executables contain only the Java Card RE packages, and an installer applet.

CHAPTER 12

Using the Installer

The Development Kit installer can be used to:

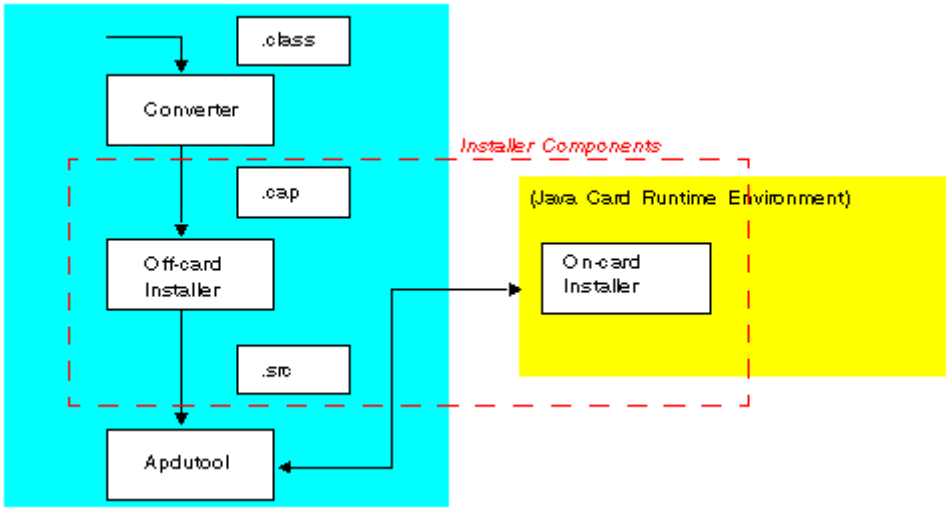
- Dynamically download a Java Card package to a Java Card technology-compliant smart card. During development, the CAP file can be installed in the C-language Java Card RE rather than on a Java Card technology-compliant smart card. The installer is capable of downloading version 2.1 and 2.2 Java Card technology-based CAP files ("Java Card CAP files").
- Perform necessary on-card linking.
- Delete applets and packages from a Java Card technology-compliant smart card. Once the installer is selected, requests for deletion can be sent from the terminal to the Java Card technology-compliant smart card in the form of APDU commands. For more information, see "Deleting Packages and Applets" on page 83.

The installer is not a multiselectable application. On startup, the installer is the default applet on logical channel 0. The default applet on the other logical channels is set to "`No applet selected`".

Installer Components and Data Flow

FIGURE 6 illustrates the components of the installer and how they relate to the rest of Java Card technology. The dotted line encloses the installer components that are described in this chapter.

FIGURE 6 Installer Components



The data flow of the installation process is as follows:

1. **An off-card installer takes a version 2.1 or 2.2 CAP file, produced by the Java Card technology-based converter ("Java Card converter"), as the input, and produces a text file that contains a sequence of APDU commands.**
2. **This set of APDUs is then read by the APDUTool and sent to the on-card installer.**
3. **The on-card installer processes the CAP file contents contained in the APDU commands as it receives them.**
4. **The response APDU from the on-card installer contains a status and optional response data.**

The off-card installer is called `scriptgen`. The on-card installer is simply called "installer" in this document.

For more information about the installer, please see the *Runtime Environment Specification for the Java Card™ Platform, Version 2.2.1* (Sun Microsystems, Inc., 2003).

Running scriptgen

The `scriptgen` tool converts a package contained in a CAP file into a script file. The script file contains a sequence of APDUs in ASCII format suitable for another tool, such as `apdutool`, to send to the CAD. The CAP file component order in the APDU script is identical to the order recommended by the *Virtual Machine Specification for the Java Card™ Platform, Version 2.2.1* (Sun Microsystems, Inc., 2003).

Enter the `scriptgen` command on the command line in this format:

```
scriptgen [options] <capFilePath>
```

where `options` include:

TABLE 19 scriptgen Command Line Options

Option	Description
`-help`	Prints a help message and exits.
`-nobanner`	Suppresses printing of the version number.
`-nobeginend`	Suppresses the output of the "CAP Begin" and "CAP End" APDU commands.
`-o <`*filename*`>`	Specifies an output filename (default is `stdout`).
`-package <`*package_name*`>`	Specifies the name of the package contained in the CAP file. According to the *Virtual Machine Specification for the Java Card™ Platform, Version 2.2.1*, the CAP file can contain components besides the ones required by the package. This option helps to avoid any possible ambiguity in determining which components should be included.
`-version`	Prints the version number and exits.

Note – If the CAP file contains components of multiple packages, you **must** use the `-package <`*package_name*`>` option to specify which package to process.

Note – The APDUtool commands: `powerup;` and `powerdown;` are not included in the output from `scriptgen`.

Installer Applet AID

The on-card installer applet AID is:
`0xa0,0x00,0x00,0x00,0x62,0x03,0x01,0x08,0x01`

Downloading CAP Files and Creating Applets

The installer is invoked by using the APDUtool. (See Chapter 13 "Sending and Receiving APDU Commands.")

Procedures for CAP file download and applet instance creation are described in the following sections:

- Downloading the CAP File
- Creating an Applet Instance

These scenarios are described in the following sections.

▼ Downloading the CAP File

In this scenario, the CAP file is downloaded and applet creation (instantiation) is postponed until a later time. (Refer to the Create Only scenario below.) Follow these steps to perform this installation:

1. **Use `scriptgen` to convert a CAP file to an APDU script file.**
2. **Prepend these commands to the APDU script file:**

```
powerup;
// Select the installer applet
0x00 0xA4 0x04 0x00 0x09 0xa0 0x00 0x00 0x00 0x62 0x03 0x01 0x08 0x01
    0x7F;
```

3. **Append this command to the APDU script file:**

```
powerdown;
```

4. **Invoke APDUTool with this APDU script file path as the argument.**

▼ Creating an Applet Instance

In this scenario, the applet from a previously downloaded CAP file or an applet compiled in the mask is created. For example, follow these steps to create the JavaPurse applet:

1. **Determine the applet AID.**
2. **Create an APDU script similar to this:**

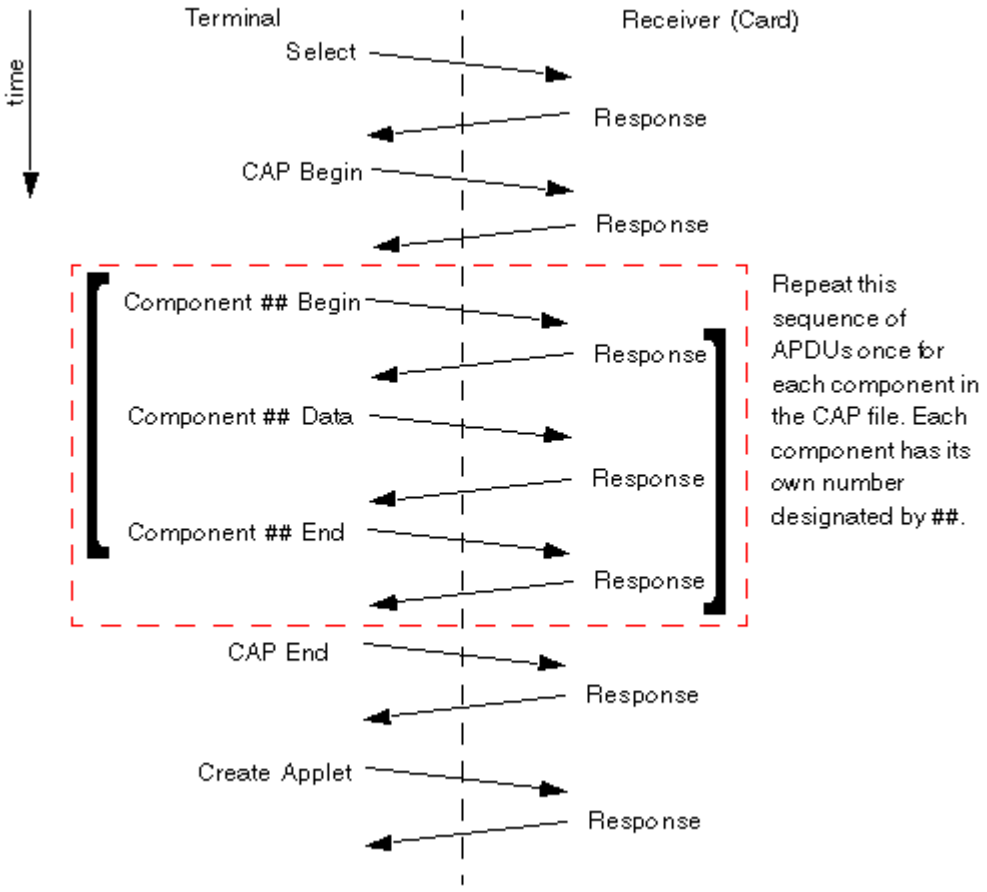
```
powerup;
// Select the installer applet
0x00 0xA4 0x04 0x00 0x09 0xa0 0x00 0x00 0x00 0x62 0x03 0x01 0x08 0x01
    0x7F;
// create JavaPurse
0x80 0xB8 0x00 0x00 0x0b 0x09 0xa0 0x00 0x00 0x00 0x62 0x03 0x01 0x04
    0x01 0x00
0x7F;
powerdown;
```

3. **Invoke APDUTool with this APDU script file path as the argument.**

Installer APDU Protocol

The Installer APDU protocol follows a specific time sequence of events in the transmission of Applet Protocol Data Units as shown in the following figure.

FIGURE 7 Installer APDU Transmission Sequence



APDU Types

There are many different APDU types, which are distinguished by their fields, and field values. The following is a general list of APDUs.

- Select
- Response
- CAP Begin
- CAP End
- Component ## Begin
- Component ## End

- Component ## Data
- Create Applet
- Abort

Descriptions of each of these APDU data types, including their bit frame formats, field names and field values follows.

Note – In the following APDU commands, the *x* in the second nibble of the class byte indicates that the installer can be invoked on channels 0, 1, or 2. For example, `0x8x`.

Select

The table below specifies the field sequence in the Select APDU, which is used to invoke the on-card installer.

TABLE 20 Select APDU Command

0x0*x*, 0xa4, 0x04, 0x00	Lc field	Installer AID	Le field

Response

The table below specifies the field sequence in the Response APDU. A Response APDU is sent as a response by the on-card installer after each APDU that it receives. The Response APDU can be either an Acknowledgment (called an ACK) which indicates that the most recent APDU was received successfully, or it can be a Negative Acknowledgement (called a NAK) which indicates that the most recent APDU was not received successfully and must be either resent or the entire installer transmission must be restarted. The first ACK indicates that the on-card installer is ready to receive. The value for an ACK frame SW1SW2 is 9000, and the value for a NAK frame SW1SW2 is 6XXX.

TABLE 21 Response APDU Command

[optional response data]	SW1SW2

CAP Begin

The table below specifies the field sequence in the CAP Begin APDU. The CAP Begin APDU is sent to the on-card installer, and indicates that the CAP file components are going to be sent next, in sequentially numbered APDUs.

TABLE 22 CAP Begin APDU Command

0x8*x*, 0xb0, 0x00, 0x00	[Lc field]	[optional data]	Le field

CAP End

The table below specifies the field sequence in the CAP End APDU. The CAP End APDU is sent to the on-card installer, and indicates that all of the CAP file components have been sent.

TABLE 23 CAP End APDU Command

0x8*x*, 0xba, 0x00, 0x00	[Lc field]	[optional data]	Le field

Component ## Begin

The table below specifies the field sequence in the Component ## Begin APDU. The double pound sign indicates the component token of the component being sent. The CAP file is divided into many components, based on class, method, etc. The Component ## Begin APDU is sent to the on-card installer, and indicates that component ## of the CAP file is going to be sent next.

TABLE 24 Component ## Begin APDU Command

0x8*x*, 0xb2, 0x##, 0x00	[Lc field]	[optional data]	Le field

Component ## End

The table below specifies the field sequence in the Component ## End APDU. The Component ## End APDU is sent to the on-card installer, and indicates that component ## of the CAP file has been sent.

TABLE 25 Component ## End APDU Command

0x8*x*, 0xbc, 0x##, 0x00	[Lc field]	[optional data]	Le field

Component ## Data

The table below specifies the field sequence in the Component ## Data APDU. The Component ## Data APDU is sent to the on-card installer, and contains the data for component ## of the CAP file.

TABLE 26 Component ## Data APDU Command

0x8*x*, 0xb4, 0x##, 0x00	Lc field	Data field	Le field

Create Applet

The table below specifies the field sequence in the Create Applet APDU. The Create Applet APDU is sent to the on-card installer, and tells the on-card installer to create an applet instance from each of the already sequentially transmitted components of the CAP file.

TABLE 27 Create Applet APDU Command

0x8*x*, 0xb8, 0x00, 0x00	Lc field	AID length field	AID field	parameter length field	[parameters]	Le field

Abort

The table below specifies the data sequence in the Abort APDU. The Abort APDU indicates that the transmission of the CAP file is terminated, and that the transmission is not complete and must be redone from the beginning in order to be successful.

TABLE 28 Abort APDU Command

0x8*x*, 0xbe, 0x00, 0x00	Lc field	[optional data]	Le field

APDU Responses to Installation Requests

The installer sends a response code of `0x9000` to indicate that a command completed successfully. Version 2.2.1 of the Java Card reference implementation provides a number of codes that can be sent in response to unsuccessful installation requests. TABLE 29 describes these codes.

TABLE 29 APDU Responses to Installation Requests

Response Code	Description
0x6402	Invalid CAP file magic number. • **Cause:** An incorrect magic number was specified in the CAP file. • **Solution:** Refer to the *Java™ Virtual Machine Specification* for the correct magic number. Ensure that the CAP file is built correctly, run it through `scriptgen`, and download the resulting script file to the card.
0x6403	Invalid CAP file minor number. • **Cause:** An invalid CAP file minor number was specified in the CAP file. • **Solution**: Refer to the *Java™ Virtual Machine Specification* for the correct minor number. Ensure that the CAP file is built correctly, run it through `scriptgen`, and download the resulting script file to the card.
0x6404	Invalid CAP file major number. • **Cause:** An invalid CAP file major number was specified in the CAP file. • **Solution**: Refer to the *Java™ Virtual Machine Specification* for the correct major number. Ensure that the CAP file is built correctly, run it through `scriptgen`, and download the resulting script file to the card.
0x640b	Integer not supported. • **Cause:** An attempt was made to download a CAP file that requires integer support into a CREF that does not support integers. • **Solution:** Either change the CAP file so that it does not require integer support or build the version of CREF that supports integers.
0x640c	Duplicate package AID found. • **Cause:** A duplicate package AID was detected in CREF. • **Solution:** Choose a new AID for the package to be installed.
0x640d	Duplicate Applet AID found. • **Cause:** A duplicate Applet AID was detected in CREF. • **Solution:** Choose a new AID for the applet to be installed.
0x640f	Installation aborted. • **Cause:** Installation was aborted by an outside command. • **Solution:** Restart the CAP installation from the beginning and check the `INS` bytes in the installation script for the offending command.

TABLE 29 APDU Responses to Installation Requests

Response Code	Description
0x6421	Installer in error state. • **Cause:** A non-recoverable error previously occurred. • **Solution:** Scan the `apdutool` output for previous APDU responses indicating an error. Restart the CAP installation.
0x6422	CAP file component out of order. • **Cause:** Installer unable to proceed because it did not receive a component that is a prerequisite to process the current component. • **Solution:** Check the script file contents for the correct component ordering.
0x6424	Exception occurred. • **Cause:** General purpose error in the installer or applet code. • **Solution:** Check your applet code for errors.
0x6425	Install APDU command out of order. • **Cause:** Installer APDU commands were received out of order. • **Solution:** Check the script file for the order of APDU commands. See "Installer APDU Protocol" on page 73 for more information on the ordering of APDU commands.
0x6428	Invalid component tag number. • **Cause:** An incorrect component tag number was detected during download. • **Solution:** Refer to Chapter 6 in the *Java™ Virtual Machine Specification* for the correct tag number.
0x6436	Invalid install instruction. • **Cause:** An invalid Installer APDU commands was received. • **Solution:** Check the script file for the offending command. See "Installer APDU Protocol" on page 73 for more information on APDU commands.
0x6437	On-card package max exceeded. • **Cause:** Package installation failed because the number of packages that can be stored on the card has been exceeded. • **Solution:** Remove some packages from the CREF.
0x6438	Imported package not found. • **Cause:** A package that is required by the current package was not found. • **Solution:** Download the required package first.
0x643a	On-card applet package max exceeded. • **Cause:** Installation of an applet package failed because the number of applet packages that can be stored on the card has been exceeded. • **Solution:** Remove some applet packages from the CREF.

TABLE 29 APDU Responses to Installation Requests

Response Code	Description
0x6442	Maximum allowable package methods exceeded. • **Cause:** The limit of 128 package methods on the card has been exceeded. • **Solution:** Modify the package to support fewer methods.
0x6443	Applet not found for installation. • **Cause:** An attempt was made to create an applet instance, but the applet code was not installed on the card. • **Solution:** Verify that the applet package has been downloaded to the card.
0x6444	Applet creation failed. • **Cause:** A general purpose error to indicate that an unsuccessful attempt was made to create the applet. • **Solution:** Verify availability of resources on the card, check the applet's `install` method, and so on.
0x644f	Package name is too long. • **Cause:** The package name exceeds the length specified in Section 2.2. 4.1of the *Java™ Virtual Machine Specification*. • **Solution:** Replace the name and rebuild.
0x6445	Maximum allowable applet instances exceeded. • **Cause:** Creation of the applet instance failed because the number of applet instances that can be stored on the card has been exceeded. • **Solution:** Remove some applet instances from the CREF.
0x6446	Memory allocation failed. • **Cause:** The amount of memory available on the card has been exceeded. • **Solution:** Verify the amount of memory that is available on the card. Remove packages, applets, and so on, to create enough space. Check the memory requirements of the applet or package being installed or downloaded.
0x6447	Imported class not found. • **Cause:** A class that is required by the current class was not found. • **Solution:** Download the required class first.

A Sample APDU Script

The following is a sample APDU script to download, create, and select the `HelloWorld` applet.

```
powerup;

// Select the installer applet
```

```
0x00 0xA4 0x04 0x00 0x09 0xa0 0x00 0x00 0x00 0x62 0x03 0x01 0x08 0x01
    0x7F;

// CAP Begin
0x80 0xB0 0x00 0x00 0x00 0x7F;

// com/sun/javacard/samples/HelloWorld/javacard/Header.cap
// component begin
0x80 0xB2 0x01 0x00 0x00 0x7F;
// component data
0x80 0xB4 0x01 0x00 0x16 0x01 0x00 0x13 0xDE 0xCA 0xFF 0xED 0x01 0x02
    0x04 0x00 0x01 0x09 0xA0 0x00 0x00 0x00 0x62 0x03 0x01 0x0C 0x01
    0x7F;
// component end
0x80 0xBC 0x01 0x00 0x00 0x7F;

// com/sun/javacard/samples/HelloWorld/javacard/Directory.cap
0x80 0xB2 0x02 0x00 0x00 0x7F;
0x80 0xB4 0x02 0x00 0x20 0x02 0x00 0x1F 0x00 0x13 0x00 0x1F 0x00 0x0E
    0x00 0x0B 0x00 0x36 0x00 0x0C 0x00 0x65 0x00 0x0A 0x00 0x13 0x00
    0x00 0x00 0x6C 0x00 0x00 0x00 0x00 0x00 0x00 0x01 0x7F;
0x80 0xB4 0x02 0x00 0x02 0x01 0x00 0x7F;
0x80 0xBC 0x02 0x00 0x00 0x7F;

// com/sun/javacard/samples/HelloWorld/javacard/Import.cap
0x80 0xB2 0x04 0x00 0x00 0x7F;
0x80 0xB4 0x04 0x00 0x0E 0x04 0x00 0x0B 0x01 0x00 0x01 0x07 0xA0 0x00
    0x00 0x00 0x62 0x01 0x01 0x7F;
0x80 0xBC 0x04 0x00 0x00 0x7F;

// com/sun/javacard/samples/HelloWorld/javacard/Applet.cap
0x80 0xB2 0x03 0x00 0x00 0x7F;
0x80 0xB4 0x03 0x00 0x11 0x03 0x00 0x0E 0x01 0x0A 0xA0 0x00 0x00 0x00
    0x62 0x03 0x01 0x0C 0x01 0x01 0x00 0x14 0x7F;
0x80 0xBC 0x03 0x00 0x00 0x7F;

// com/sun/javacard/samples/HelloWorld/javacard/Class.cap
0x80 0xB2 0x06 0x00 0x00 0x7F;
0x80 0xB4 0x06 0x00 0x0F 0x06 0x00 0x0C 0x00 0x80 0x03 0x01 0x00 0x01
    0x07 0x01 0x00 0x00 0x00 0x1D 0x7F;
0x80 0xBC 0x06 0x00 0x00 0x7F;

// com/sun/javacard/samples/HelloWorld/javacard/Method.cap
0x80 0xB2 0x07 0x00 0x00 0x7F;
0x80 0xB4 0x07 0x00 0x20 0x07 0x00 0x65 0x00 0x02 0x10 0x18 0x8C 0x00
    0x01 0x18 0x11 0x01 0x00 0x90 0x0B 0x87 0x00 0x18 0x8B 0x00 0x02
    0x7A 0x01 0x30 0x8F 0x00 0x03 0x8C 0x00 0x04 0x7A 0x7F;
```

```
0x80 0xB4 0x07 0x00 0x20 0x05 0x23 0x19 0x8B 0x00 0x05 0x2D 0x19 0x8B
    0x00 0x06 0x32 0x03 0x29 0x04 0x70 0x19 0x1A 0x08 0xAD 0x00 0x16
    0x04 0x1F 0x8D 0x00 0x0B 0x3B 0x16 0x04 0x1F 0x41 0x7F;
0x80 0xB4 0x07 0x00 0x20 0x29 0x04 0x19 0x08 0x8B 0x00 0x0C 0x32 0x1F
    0x64 0xE8 0x19 0x8B 0x00 0x07 0x3B 0x19 0x16 0x04 0x08 0x41 0x8B
    0x00 0x08 0x19 0x03 0x08 0x8B 0x00 0x09 0x19 0xAD 0x7F;
0x80 0xB4 0x07 0x00 0x08 0x00 0x03 0x16 0x04 0x8B 0x00 0x0A 0x7A 0x7F;
0x80 0xBC 0x07 0x00 0x00 0x7F;

// com/sun/javacard/samples/HelloWorld/javacard/StaticField.cap
0x80 0xB2 0x08 0x00 0x00 0x7F;
0x80 0xB4 0x08 0x00 0x0D 0x08 0x00 0x0A 0x00 0x00 0x00 0x00 0x00 0x00
    0x00 0x00 0x00 0x00 0x7F;
0x80 0xBC 0x08 0x00 0x00 0x7F;

// com/sun/javacard/samples/HelloWorld/javacard/ConstantPool.cap
0x80 0xB2 0x05 0x00 0x00 0x7F;
0x80 0xB4 0x05 0x00 0x20 0x05 0x00 0x36 0x00 0x0D 0x02 0x00 0x00 0x00
    0x06 0x80 0x03 0x00 0x03 0x80 0x03 0x01 0x01 0x00 0x00 0x00 0x06
    0x00 0x00 0x01 0x03 0x80 0x0A 0x01 0x03 0x80 0x0A 0x7F;
0x80 0xB4 0x05 0x00 0x19 0x06 0x03 0x80 0x0A 0x07 0x03 0x80 0x0A 0x09
    0x03 0x80 0x0A 0x04 0x03 0x80 0x0A 0x05 0x06 0x80 0x10 0x02 0x03
    0x80 0x0A 0x03 0x7F;
0x80 0xBC 0x05 0x00 0x00 0x7F;

// com/sun/javacard/samples/HelloWorld/javacard/RefLocation.cap
0x80 0xB2 0x09 0x00 0x00 0x7F;
0x80 0xB4 0x09 0x00 0x16 0x09 0x00 0x13 0x00 0x03 0x0E 0x23 0x2C 0x00
    0x0C 0x05 0x0C 0x06 0x03 0x07 0x05 0x10 0x0C 0x08 0x09 0x06 0x09
    0x7F;
0x80 0xBC 0x09 0x00 0x00 0x7F;

// CAP End
0x80 0xBA 0x00 0x00 0x00 0x7F;

// create HelloWorld
0x80 0xB8 0x00 0x00 0x0b 0x09 0xa0 0x00 0x00 0x00 0x62 0x03 0x01 0x03;
0x01 0x00 0x7F;

// Select HelloWorld
0x00 0xA4 0x04 0x00 9 0xA0 0x00 0x00 0x00 0x62 0x03 0x01 0x03 0x01
    0x7F;

powerdown;
```

Deleting Packages and Applets

The Installer in version 2.2.1 of the Java Card reference implementation provides the ability to delete package and applet instances from the card's memory. Once the Installer is selected, it can receive deletion requests from the terminal in the form of ADPU commands. Requests to delete an applet or package cannot be sent from an applet on the card. For more information on package and applet deletion, see the *Runtime Environment Specification for the Java Card™ Platform, Version 2.2.1* (Sun Microsystems, Inc., 2003).

▼ How to Send a Deletion Request

1. **Select the Installer applet on the card.**
2. **Send the ADPU for the appropriate deletion request to the Installer. The requests that you can send are described in the following sections:**
   - Delete Package
   - Delete Package and Applets
   - Delete Applets

   For information on the responses that the ADPU requests can return, see "APDU Responses to Deletion Requests" on page 85.

APDU Requests to Delete Packages and Applets

You can send requests to delete a package, a package and its applets, and individual applets.

Note – In the following APDU commands, the *x* in the second nibble of the class byte indicates that the installer can be invoked on channels 0, 1, or 2. For example, 0x8*x*.

Delete Package

In this request, the Data field contains the size of the package AID and the AID of the package to be deleted. The following is the format of the Delete Package request and the expected response:

TABLE 30 Delete Package Command

0x8*x*, 0xc0, 0x*XX*, 0x*XX*	Lc field	Data field	Le field

where 0x*XX* can be any value for the **P1** and **P2** parameters. The installer will ignore the 0x*XX* values. An example of a delete package request on channel 1 would be:

```
//Delete Package Request:
 0x81 0xC0 0x00 0x00 0x08 0x07 0xa0 0x00 0x00 0x00 0x62 0x12 0x34 0x7F;
```

In this example, `0x07` is the AID length and `0xa0 0x00 0x00 0x00 0x62 0x12 0x34` is the package AID.

Delete Package and Applets

This request is similar to the Delete Package command. In this case the package and applets are removed simultaneously. The data field will contain the size of the package AID and the AID of the package to be deleted. The following is the format of the Delete Packages and Applets request and the expected response:

TABLE 31 Delete Package and Applets Command

0x8*x*, 0xc2, 0x*XX*, 0x*XX*	Lc field	Data field	Le field

where 0x*XX* can be any value for the **P1** and **P2** parameters. The installer will ignore the 0x*XX* values. An example of a package and applets deletion request on channel 1 would be:

```
//Delete Package And Applets request
0x81 0xC2 0x00 0x00 0x08 0x07 0xa0 0x00 0x00 0x00 0x62 0x12 0x34 0x7F;
```

In this example, `0x07` is the AID length and `0xa0 0x00 0x00 0x00 0x62 0x12 0x34` is the package AID.

Delete Applets

In this request, the "#" symbol in the **P1** byte indicates the number of applets to be deleted which can have a maximum value of eight. The **Lc** field contains the size of the data field. Data field contains a list of AID size and AID pairs. The following is the format of the Delete Applet request and the expected response:

TABLE 32 Delete Applet Command

0x8*x*, 0xc4, 0x0#, 0x*XX*	Lc field	Data field	Le field

where 0x*XX* can be any value for the **P2** parameter. The installer will ignore the 0x*XX* values. An example of a applet deletion request on channel 1 would be:

```
//Delete the applet's request for two applets
0x81 0xC4 0x02 0x00 0x12 0x08 0xa0 0x00 0x00 0x00 0x62 0x12 0x34 0x12
    0x08 0xa0 0x00 0x00 0x00 0x62 0x12 0x34 0x13 0x7F;
```

In this example, the "#" symbol is replaced with "2" (0x02) indicating that there are two applets to be deleted. The first applet is 0xa0 0x00 0x00 0x00 0x62 0x12 0x34 0x12 and the second applet is 0xa0 0x00 0x00 0x00 0x62 0x12 0x34 0x13.

APDU Responses to Deletion Requests

When the installer receives the request from the terminal, it can return any of the following responses:

TABLE 33 APDU Responses to Deletion Requests

Response Code	Description
0x6a86	Invalid value for P1 or P2 parameter. • **Cause:** Value for P1 is less than 1 or greater than 8. • **Solution:** Ensure that the value for P1 is between 1 and 8.
0x6443	Applet not found for deletion. • **Cause:** The applet with the specified AID does not exist. • **Solution:** Check and correct the AID.
0x644b	Package not found. • **Cause:** The package with the specified AID does not exist. • **Solution:** Check and correct the AID.

TABLE 33 APDU Responses to Deletion Requests

Response Code	Description
0x644c	Dependencies on package. • **Cause:** Package has other packages dependent on it, or there are some object instances of classes belonging to this package residing in memory. • **Solution:** Determine which packages are dependent and remove them. If there are object instances of classes belonging to this package residing in memory, try the package and applet deletion combination command to remove the package from card memory.
0x644d	One or more applet instances of this package are present. • **Cause:** One or more applet instances of this package are present • **Solution:** Remove the applets first and then try package deletion, or try the package and applet deletion combination command.
0x644e	Package is ROM package. • **Cause:** An attempt was made to delete a package in ROM. • **Solution:** There is no solution to this problem since packages in ROM cannot be deleted.
0x6448	Dependencies on applet. • **Cause:** Other applets are using objects owned by this applet. • **Solution:** Remove references from other applets to this applet's objects, or try to delete the dependent applets along with this applet.
0x6449	Internal memory constraints. • **Cause:** There is not enough memory available for the intermediate structures required by applet deletion. • **Solution:** It may not be possible to recover from this error. One possible thing that can be tried in case of multiple applet deletion is to try to delete applets individually.
0x6452	Cannot delete applet; an applet in the same context is currently active on one of the logical channels. • **Cause:** An attempt was made to delete an applet while another applet in the same context is currently active on one of the logical channels. • **Solution:** In the context of the applet that you are attempting to delete, make sure that no applet is selected on any of the logical channels. Then, re-attempt to delete the applet.
0x6700	Invalid value for `Lc` parameter. • **Cause:** In case of package deletion, the value for `Lc` is less than 6 or greater than 17. In case of applet deletion, the value for `Lc` is less than 7 or greater than 136. • **Solution:** Value of `Lc` in both of these cases depends on the AIDs being passed in the APDU. Make sure the AIDs are correct and value for `Lc` is between 6 and 16 in case of package deletion and between 7 and 135 in case of applet deletion.

The response has the following format:

TABLE 34 APDU Response Format

[optional response data]	SW1SW2

Installer Limitations

- The maximum length of the parameter in the applet creation APDU command is 110.
- The maximum number of packages to be downloaded is 32, including up to 16 applet packages.
- The maximum number of applet instances to be created is 16.
- The maximum length of data in the installer APDU commands is 128.
- No on-card CAP file verification is supported.
- All subsequent APDU commands enclosed in a "`CAP Begin`," "`CAP End`" APDU pair will continue to fail after an error occurs.
- The maximum number of applets that can be deleted using one command is eight.

CHAPTER 13

Sending and Receiving APDU Commands

The apdutool reads a script file containing Application Protocol Data Unit commands (APDUs) and sends them to the C-language Java Card RE (or other Java Card RE) or the Java Card WDE. Each APDU is processed and returned to apdutool, which displays both the command and response APDUs on the console. Optionally, apdutool can write this information to a log file.

Running apdutool

The file to invoke apdutool is a shell script (apdutool) on the Solaris or Linux platform, and a batch file (apdutool.bat) on the Microsoft Windows 2000 platform.

The command line usage for apdutool is:

```
apdutool [-h hostname] [-nobanner] [-noatr] [-o <outputFile>]
    [-p port] [-s serialPort] [-version]
    <inputFile> [<inputFile> ...]
```

The option values and their actions are:

TABLE 35 apdutool Command Line Options

Option	Description
-h	Specifies the host name on which the TCP/IP socket port is found. (See the flag -p.)
-help	Displays online documentation for this command.
-noatr	Suppresses outputting an ATR (answer to reset).
-nobanner	Suppresses all banner messages.

TABLE 35 apdutool Command Line Options

Option	Description
`-o`	Specifies an output file. If an output file is not specified with the `-o` flag, output defaults to standard output.
`-p`	Specifies a TCP/IP socket port other than the default port (which is 9025).
`-s` *`serialPort`*	Specifies the serial port to use for communication, rather than a TCP/IP socket port. For example, *`serialPort`* can be `COM1` on a Microsoft Windows 2000 system and `/dev/term/a` on a Solaris system. Currently, this option is not supported on the Linux platform. To use this option, the `javax.comm` package must be installed on your system. For more information on installing this package, see "Prerequisites for Installing the Binary Release" on page 4. If you enter the name of a serial port that does not exist on your system, `apdutool` will respond by printing the names of available ports.
`-version`	Outputs the version information.
<*`inputFile`*>	Allows you to specify the input script (or scripts).

apdutool Examples

Directing Output to the Console

The following is a command line invocation sample:

```
apdutool example.scr
```

This command runs `apdutool` with the file `example.scr` as input. Output is sent to the console. The default TCP port (9025) is used.

Directing Output to a File

```
apdutool -o example.scr.out example.scr
```

This command runs `apdutool` with the file `example.scr` as input. Output is written to the file `example.scr.out`.

Using APDU Script Files

An APDU script file is a protocol-independent APDU format containing comments, script file commands, and C-APDUs. Script file commands and C-APDUs are terminated with a semicolon (;). Comments can be of any of the three Java-style comment formats (//, /*, or /**).

APDUs are represented by decimal, hex or octal digits, UTF-8 quoted literals or UTF-8 quoted strings. C-APDUs may extend across multiple lines.

C-APDU syntax for `apdutool` is as follows:

```
<CLA> <INS> <P1> <P2> <LC> [<byte 0> <byte 1> ... <byte LC-1>] <LE> ;
```

where:

`<CLA>` :: ISO 7816-4 class byte.
`<INS>` :: ISO 7816-4 instruction byte.
`<P1>` :: ISO 7816-4 P1 parameter byte.
`<P2>` :: ISO 7816-4 P2 parameter byte.
`<LC>` :: ISO 7816-4 input byte count.
`<byte 0> ... <byte LC-1>` :: input data bytes.
`<LE>` :: ISO 7816- 4 expected output length byte. 0 implies 256.

The following script file commands are supported:

TABLE 36 Supported APDU Script File Commands

Command	Description
`delay <Integer>;`	Pauses execution of the script for the number of milliseconds specified by `<Integer>`.
`echo "string";`	Echoes the quoted string to the output file. The leading and trailing quote characters are removed.
`output off;`	Suppresses printing of the output.
`output on;`	Restores printing of the output.
`powerdown;`	Sends a powerdown command to the reader.
`powerup;`	Sends a power up command to the reader. A powerup command must be executed prior to sending any C-APDUs to the reader.

CHAPTER 14

Using Cryptography Extensions

This release provides an implementation of basic security and cryptography classes. These implementations are supported by:

- C-language Java Card RE
- the Java Card platform Workstation Development Environment tool (Java Card WDE)

The support for security and cryptography allows you to:

- generate message digests using the SHA1 algorithm
- generate cryptographic keys on Java Card technology-compliant smart cards for use in the ECC and RSA algorithms
- set cryptographic keys on Java Card technology-compliant smart cards for use in the AES, DES, 3DES, ECC, and RSA algorithms
- encrypt and decrypt data with the keys using the AES, DES, 3DES, and RSA algorithms
- generate signatures using the AES, DES, 3DES, ECC, or SHA and RSA algorithms
- generate sequences of random bytes
- generate checksums

Note – DES is also known as single-key DES. 3DES is also known as triple-DES.

For more information on the SHA1, DES, 3DES, and RSA encryption schemes, see:

- for SHA1—"*Secure Hash Standard*", FIPS Publication 180-1: http://www.itl.nist.gov/
- for DES—"*Data Encryption Standard (DES)*", FIPS Publication 46-2 and "*DES Modes of Operation*", FIPS Publication 81: http://www.itl.nist.gov/

- for RSA—*"RSAES-OAEP (Optional Asymmetric Encryption Padding) Encryption Scheme"*:
  http://www.rsasecurity.com/
- for AES—*"Advanced Encryption Standard (AES)"* FIPs Publication 197:
  http://www.itl.nist.gov/
- for ECC—*"Public Key Cryptography for the Financial Industry: The Elliptic Curve Digital Signature Algorithm"* (ECDSA): X9.62-1998
  http://www.x9.org/
- for Checksum—*"Information technology—Telecommunications and information exchange between systems—High-level data link control (HDLC) procedures"* ISO/IEC-13239:2002 (replaces ISO-3309):
  http://www.iso.org

Supported Cryptography Classes

The implementation of security and cryptography in version 2.2.1 of the Java Card reference implementation supports the use of the following classes:

- `javacardx.crypto.Cipher`
- `javacard.security.Checksum`
- `javacard.security.KeyAgreement`
- `javacard.security.KeyPair`
- `javacard.security.KeyBuilder`
- `javacard.security.MessageDigest`
- `javacard.security.RandomData`
- `javacard.security.Signature`

TABLE 37 lists the cryptography algorithms that are implemented for CREF and Java Card WDE:

TABLE 37 Algorithms Implemented by the Cryptography Classes

Class	Algorithm
`Checksum`	• `ALG_ISO3309_CRC16`—ISO/IEC 3309-compliant 16-bit CRC algorithm. This algorithm uses the generator polynomial: `x^16+x^12+x^5+1`. The default initial checksum value used by this algorithm is 0. This algorithm is also compliant with the frame-checking sequence as specified in section 4.2.5.2 of the ISO/IEC 13239 specification. • `ALG_ISO3309_CRC32`—ISO/IEC 3309-compliant 32-bit CRC algorithm. This algorithm uses the generator polynomial: `X^32+X^26+X^23+X^22+X^16+X^12+X^11+X^10+X^8 +X^7+X^5+X^4+X^2+X+1`. The default initial checksum value used by this algorithm is 0. This algorithm is also compliant with the frame-checking sequence as specified in section 4.2.5.3 of the ISO/IEC 13239 specification.
`Cipher`	• `ALG_DES_CBC_ISO9797_M2`—provides a cipher using `DES` in `CBC` mode. This algorithm uses `CBC` for `DES` and `3DES`. Input data is padded according to the ISO 9797 method 2 (ISO 7816-4, EMV'96) scheme. • `ALG_RSA_PKCS1`—provides a cipher using `RSA`. Input data is padded according to the `PKCS#1` (v1.5) scheme. • `ALG_AES_BLOCK_128_CBC_NOPAD`—provides a cipher using `AES` with block size 128 in CBC mode and does not pad input data.
`KeyAgreement`	• `ALG_EC_SVDP_DH`—elliptic curve secret value derivation primitive, Diffie-Hellman version, as per [IEEE P1363]. • `ALG_EC_SVDP_DHC`—elliptic curve secret value derivation primitive, Diffie-Hellman version, with cofactor multiplication, as per [IEEE P1363].
`KeyBuilder`	the algorithms define the key lengths for: • 128-bit `AES` • 64-bit `DES` • 112-, 128-, 160-, 192-bit `ECC` • 128-bit `DES3` • 512-bit `RSA`
`KeyPair`	the algorithms define the key lengths for: • 112-, 128-, 160-, 192-bit `ECC` • 512-bit `RSA`

TABLE 37 Algorithms Implemented by the Cryptography Classes

Class	Algorithm
`MessageDigest`	message digest algorithm `SHA1`
`RandomData`	pseudo-random number generator with a 48-bit seed, which is modified using a linear congruential formula.
`Signature`	• `ALG_DES_MAC8_ISO9797_M2`—generates an 8-byte `MAC` (most significant 8 bytes of encrypted block) using `DES` or `3DES` in `CBC` mode. This algorithm uses `CBC` for `DES` and `3DES`. Input data is padded according to the ISO 9797 method 2 (ISO 7816-4, EMV'96) scheme. • `ALG_RSA_SHA_PKCS1`—encrypts the 20 byte `SHA1` digest using `RSA`. The digest is padded according to the `PKCS#1` (v1.5) scheme. • `ALG_AES_MAC_128_NOPAD`—generates a 16-byte MAC using `AES` with blocksize 128 in CBC mode and does not pad input data. • `ALG_ECDSA_SHA`—signs/verifies the 20-byte SHA digest using ECDSA.

Instantiating the Classes

Implementations of the cryptography classes extend the corresponding base class with implementations of their abstract methods. All data allocation associated with the implementation instance is performed when the instance is constructed. This is done to ensure that any lack of required resources can be flagged when the applet is installed.

Each cryptography class, except `KeyPair`, has a `getInstance` method which takes the desired algorithm as one of its parameters. The method returns an instance of the class in the context of the calling applet. Instead of using a `getInstance` method, `KeyPair` takes the desired algorithm as a parameter in its constructor.

If you request an algorithm that is not listed in TABLE 37 or that is not implemented in this release, `getInstance` will throw a `CryptoException` with reason code `NO_SUCH_ALGORITHM`.

Temporary RAM Usage by Cryptography Algorithms

The implementation of the RSA and EC cryptography algorithms in CREF optimizes RAM usage. To do this, CREF dynamically allocates temporary memory areas in RAM. These temporary RAM areas are allocated for the duration of a native method call.

These memory areas are used as temporary RAM in the following order:

1. Inside of the Java platform stack.
2. The available DTR (clear-on-deselect) space of the current logical channel.
3. The available RTR (clear-on-reset) space.
4. The available DTR space of other logical channels.

Note that the amount of RAM available in the RTR and non-current DTR can be influenced by applets other than the one currently selected. This means that the current applet which uses the RTR and non-current DTR might fail if more applets are installed on the card.

When execution completes, CREF prints maximum memory usage in each of these areas to help you track the memory requirements of the cryptography algorithms in your own Java Card VM implementations.

CHAPTER 15

Java Card RMI Client-Side Reference Implementation

A Java Card RMI client application runs on a Card Acceptance Device (CAD) terminal which supports a J2SE or J2ME platform. The client application requires a portable and platform independent mechanism to access the Java Card RMI server applet executing on the smart card.

The basic client-side framework is implemented in the package `com.sun.javacard.javax.smartcard.rmiclient`. Refer to *Java Card™ RMI Client Application Programming Interface, Version 2.2.1* (Sun Microsystems, Inc., 2003) .

The reference implementation of Java Card Client-Side RMI API is implemented in the package `com.sun.javacard.ocfrmiclientimpl`, and is based on the Open Card Framework (OCF 1.2) for its card access mechanisms. The Open Card Framework classes provide a Java application platform independent access to a connected smart card.

For a detailed description of OCF 1.2, refer to `http://www.opencard.org/`.

For the Java Card RMI Client API Reference Implementation documentation refer to Appendix D "Reference Implementation of Java Card RMI Client-Side API."

The Java Card Remote Stub Object

Java Card RMI supports two formats for passing remote references. The format for remote references containing the class name requires stubs for remote objects available to the client application.

The standard Java RMIC compiler tool can be used as the stub compilation tool to produce stub classes required for the client application. To produce these stub classes, the RMIC compiler tool must have access to all the non-abstract classes

defined in the applet package which directly or indirectly implement remote interfaces. In addition, it needs to access the `.class` files of all the remote interfaces implemented by them.

If you want the stub class to be Java Card RMI-specific when it is instantiated on the client, it must be customized with a Java Card platform-specific implementation of the `CardObjectFactory` interface.

The standard Java RMIC compiler is used to generate the remote stub objects. `JCRemoteRefImpl`, a Java Card platform-specific implementation of the `java.rmi.server.RemoteRef` interface, allows these stub objects to work with Java Card RMI. The stub object delegates all method invocations to its configured `RemoteRef` instance.

The `com.sun.javacard.ocfrmiclientimpl.JCRemoteRefImpl` class is an example of a `RemoteRef` object customized for the Java Card platform.

For examples of how to use these interfaces and classes, see Chapter 3, "Developing Java Card RMI Applications" in the *Application Programming Notes for the Java Card™ Platform, Version 2.2.1*.

Note – Since the remote object is configured as a Java Card platform-specific object with a local connection to the smart card via the `OCFCardAccessor` object, the object is inherently not portable. A bridge class must be used if it is to be accessed from outside of this client application.

Note – Some versions of the RMIC do not treat `Thowable` as a superclass of `RemoteException`. The workaround is to declare remote methods to throw `Exception` instead.

CHAPTER 16

Localization Support in the Development Kit

This chapter describes the support for localization that the Development Kit version 2.2.1 provides. Items which require localization in the Developers Kit include the Java-based tools, CREF, and the Java-based Java Card RMI sample applications and client framework. The Java language-based programs and the C language-based programs use different localization mechanisms.

Localization Support for Java Utilities

This section describes the mechanisms used to localize the following programs and tools:

- RMI sample programs
- RMI client framework
- `scriptgen`
- `apdutool`
- `converter`
- `maskgen`
- `capdump`
- `exp2text`
- offcard verifier
- Java Card WDE

These Java utilities and programs can be localized in a similar fashion. Each uses the Java resource bundle mechanism. This mechanism allows the user to customize locale-sensitive data for a new locale without rebuilding the application. Refer to the Java™ 2 Standard Edition `java.util.ResourceBundle` class for more information regarding resource bundles.

The Development Kit also provides localization support for Java Card RMI sample applications and client framework. Localizing the client framework and the sample applications can be done in the same way as the Java Card technology-based utilities.

Since none of the Java Card reference implementation utilities or programs require a graphical user interface (GUI) and are not dependent on user input, the majority of the locale-specific data consists of static strings. Localization consists of customizing these strings for the intended locale. Locale-sensitive strings are grouped into `.properties` files (for example, `MessagesBundle.properties`). Localizing an application entails creating a new version of the properties file that contains the translated strings.

▼ Localizing a Java Program to a New Locale

The following steps are required to localize a Java program to a new locale:

1. **Create a new version of the appropriate property file which contains the correct set of strings customized for the intended locale.**
2. **Rename the property file with the appropriate locale identifier appended to the file name (for example, the French version of the `MessagesBundle.properties` file would be `MessagesBundle_fr.properties`).**
3. **Include the location of the property file in the classpath for the Java Card reference implementation utility.**

   When the Java utility is executed in an environment with the same locale as the properties file, the strings contained in that properties file will be used for output.

   For additional information regarding internationalization and localization in Java, please refer to the Java 2 Standard Edition online documentation at `http://java.sun.com/j2se/1.4.1/docs/guide/intl/index.html`.

Localization Support for CREF

Similar to the Java utilities described above, localizing CREF consists of providing the set of static output strings used by CREF correctly customized for the intended locale. Unlike the Java utilities described above, CREF must be rebuilt to localize it to a new locale.

All of the locale-sensitive strings used by CREF are stored in a single C header file, `src/share/c/common/cref_locale.h`. To localize CREF, customize the strings in this file for the new locale and then rebuild CREF.

APPENDIX **A**

Java Card Assembly Syntax Example

This appendix contains an annotated Java Card platform assembly ("Java Card Assembly") file output from the Converter. The comments in this file are intended to aid the developer in understanding the syntax of the Java Card Assembly language, and as a guide for debugging Converter output.

```
/*
 * Java Card Assembly annotated example. The code
 * contained within this example is not an executable
 * program. The intention of this program is to illustrate the
 * syntax and use of the Java Card Assembly directives and commands.
 *
 * A Java Card Assembly file is textual representation of the
 * contents of a CAP file.
 * The contents of a Java Card Assembly file are hierarchically
 * structured. The format of this structure is:
 *
 *     package
 *         package directives
 *         imports block
 *         applet declarations
 *         constant pool
 *         class
 *             field declarations
 *              virtual method tables
 *              interface table
 *              [remote interface table] - only for remote classes
 *             methods
 *                 method directives
 *                 method statements
 *
 * Java Card Assembly files support both the Java single line
 * comments and Java block
 * comments. Anything contained within a comment is ignored.
```

```
 *
 * Numbers may be specified using the standard Java notation.
 * Numbers prefixed
 * with a 0x are interpreted as
 * base-16, numbers prefixed with a 0 are base-8, otherwise
 * numbers are interpreted
 * as base-10.
 *
*/

/*
 * A package is declared with the .package directive. Only one
 * package is allowed
 * inside a Java Card Assembly
 * file. All directives (.package, .class, et.al) are case
 * insensitive. Package,
 * class, field and
 * method names are case sensitive. For example, the .package
 * directive may be written
 * as .PACKAGE,
 * however the package names example and ExAmPle are different.
 */
.package example {

    /*
     * There are only two package directives. The .aid and .version
     * directives declare
     * the aid and version that appear in the Header Component of
     * the CAP file.
     * These directives are required.

    .aid 0:1:2:3:4:5:6:7:8:9:0xa:0xb:0xc:0xd:0xe:0xf;
        // the AIDs length must be
        // between 5 and 16 bytes inclusive
    .version 0.1;        // major version <DOT> minor version

    /*
     * The imports block declares all of packages that this
     * package imports. The data
     * that is declared
     * in this section appears in the Import Component of the
     * CAP file. The ordering
     * of the entries
     * within this block define the package tokens which must be
     * used within this
     * package. The imports
     * block is optional, but all packages except for java/lang
     * import at least
     * java/lang. There should
```

```
 * be only one imports block within a package.
 */

.imports {
    0xa0:0x00:0x00:0x00:0x62:0x00:0x01 1.0;
    // java/lang aid <SPACE>
    // java/lang major version <DOT> java/lang minor version
    0:1:2:3:4:5 0.1;                        // package test2
    1:1:2:3:4:5 0.1;                        // package test3
    2:1:2:3:4:5 0.1;                        // package test4
}

/*
 * The applet block declares all of the applets within
 * this package. The data
 * declared within this block appears
 * in the Applet Component of the CAP file. This section may
 * be omitted if this
 * package declares no applets. There
 * should be only one applet block within a package.
 */

.applet {
    6:4:3:2:1:0 test1;    // the class name of a class within this
                          // package which
    7:4:3:2:1:0 test2;    // contains the method install([BSB)V
    8:4:3:2:1:0 test3;
}

/*
 * The constant pool block declares all of the constant
 * pool's entries in the
 * Constant Pool Component. The positional
 * ordering of the entries within the constant pool block
 * define the constant pool
 * indices used within this package.
 * There should be only one constant pool block within a package.
 *
 * There are six types of constant pool entries. Each of these
 * entries directly
 * corresponds to the constant pool
 * entries as defined in the Constant Pool Component.
 *
 * The commented numbers which follow each line are the constant
 * pool indexes
 * which will be used within this package.
 */

.constantPool {
```

```
    /*
     * The first six entries declare constant pool entries that
     * are contained in
     * other packages.
     * Note that superMethodRef are always declared internal
     * entry.
     */
    classRef    0.0;     // 0    package token 0, class token 0
    instanceFieldRef 1.0.2;// 1  package token 1, class token 0,
                          //   instance field token 2
    virtualMethodRef 2.0.2; // 2 package token 2, class token 0,
                //  instance field token 2
    classRef    0.3;  // 3 package token 0, class token 3
    staticFieldRef  1.0.4;   // 4 package token 1, class token 0,
                //      field token 4
    staticMethodRef  2.0.5;  // 5 package token 2, class token 0,
                //      method token 5

    /*
     * The next five entries declare constant pool entries
     * relative to this class.
     *
    classRef    test0;        // 6
    instanceFieldRef    test1/field1;        // 7
    virtualMethodRef    test1/method1()V;        // 8
    superMethodRef  test9/equals(Ljava/lang/Object;)Z;     // 9
    staticFieldRef    test1/field0;        // 10
    staticMethodRef    test1/method3()V;        // 11
}


/*
 * The class directive declares a class within the Class Component
 * of a CAP file.
 * All classes except java/lang/Object should extend an internal
 * or external
 * class. There can be
 * zero or more class entries defined within a package.
 *
 * for classes which extend a external class, the grammar is:
 * .class modifiers* class_name class_token extends
 * packageToken.ClassToken
 *
 * for classes which extend a class within this package,
 * the grammar is:
 * .class modifiers* class_name class_token extends className
 *
 * The modifiers which are allowed are defined by the Java Card
```

```
 * language subset.
 * The class token is required for public and protected classes,
 * and should not be
 * present for other classes.
 */

.class final public test1 0 extends 0.0 {

    /*
     * The fields directive declares the fields within this class.
     * There should
     * be only one fields
     * block per class.
     */

    .fields {
        public static int field0 0;
        public int field1 0;
    }

    /*
     * The public method table declares the virtual methods within
     * this classes
     * public virtual method
     * table. The number following the directive is the method
     * table base (See the
     * Class Component specification).
     *
     * Method names declared in this table are relative to
     * this class. This
     * directive is required even if there
     * are not virtual methods in this class. This is necessary
     * to establish the
     * method table base.
     */

    .publicmethodtable 1 {
        equals(Ljava/lang/Object;)Z;
        method1()V;
        method2()V;
    }

    /*
     * The package method table declares the virtual methods
     * within this classes
     * package virtual method
     * table. The format of this table is identical to the public
     * method table.
     */
```

```
    .packagemethodtable 0 {}

    .method public method1()V 1 { return; }
    .method public method2()V 2 { return; }
    .method protected static native method3()V 0 { }
    .method public static install([BSB)V 1 { return; }
}

.class final public test9 9 extends test1 {

    .publicmethodtable 0 {
        equals(Ljava/lang/Object;)Z;
        method1()V;
        method2()V;
    }
    .packagemethodtable 0 {}

    .method public equals(Ljava/lang/Object;)Z 0 {
        invokespecial 9;
        return;
    }
}

.class final public test0 1 extends 0.0 {

    .Fields {
        // access_flag, type, name [token [static Initializer]] ;
        public static byte field0 4 = 10;
        public static byte[] field1 0;
        public static boolean field2 1;
        public short field4 2;
        public int field3 0;
    }
    .PublicMethodTable 1 {
        equals(Ljava/lang/Object;)Z;
        abc()V;                  // method must be in this class
        def()V;
        labelTest()V;
        instructions()V;
    }
    .PackageMethodTable 0 {
        ghi()V;                  // method must be in this class
        jkl()V;
    }
    // if the class implements more than one interface, multiple
    // interfaceInfoTables will be present.
.implementedInterfaceInfoTable
```

```
  .interface 1.0 {   // java/rmi/Remote
  }

  .interface RemoteAccount { // The table contains method tokens
  10;  // getBalance()S
  9;  // debit(S)V
  8;  // credit(S)V
  11;  // setAccountNumber([B)V
  12;  // getAccountNumber()[B
  }
}
  .implementedRemoteInterfaceInfoTable { // The table contains
                                         // method tokens
  // excluding java.rmi.Remote
  .interface RemoteAccount { // Contains method tokens
  getBalance()S   10;   // getBalance()S
  debit(S)V       9;   // debit(S)V
  credit(S)V      8;   // credit(S)V
  setAccountNumber([B)V   11;  // setAccountNumber([B)V
  getAccountNumber()[B    12;  // getAccountNumber()[B
  }

  }
      /*
       * Declaration of 2 public visible virtual methods and two
       * package visible
       * virtual methods..
       */
      .method public abc()V 1 {
          return;
      }
      .method public def()V 2 {
          return;
      }
      .method ghi()V 0x80 {               // per the CAP file
                             //specification, method tokens
                          // for package visible methods
          return; // must have the most significant bit set to 1.
      }
      .method jkl()V 0x81 {
          return;
      }


      /*
       * This method illustrates local labels and exception table
       * entries. Labels
       * are local to each
       * method. No restrictions are placed on label names except
```

```
             * that they must
             * begin with an alphabetic
             * character. Label names are case insensitive.
             *
             * Two method directives are supported, .stack and .locals.
             * These
             * directives are used to
             * create the method header for each method. If a method
             * directive is omitted,
             * the value 0 will be used.
             *
             */

            .method public static install([BSB)V 0 {
                .stack 0;
                .locals 0;
10:
11:
12:
13:
14:
15:
                return;

                /*
                 * Each method may optionally declare an
                 * exception table. The start offset,
                 * end offset and handler offset
                 * may be specified numerically, or with a
                 * label. The format of this table
                 * is different from the exception
                 * tables contained within a CAP file. In a
                 * CAP file, there is no end
                 * offset, instead the length from the
                * starting offset is specified. In the Java Card Assembly
                 * file an end offset is specified
                 * to allow editing of the
                 * instruction stream without having to recalculate
                 * the exception table
                 * lengths manually.
                 */

                .exceptionTable {
                    // start_offset end_offset handler_offset
                    // catch_type_index;
                    l0 l4 l5 3;
                    l1 l3 l5 3;
                }
            }
```

```
        /*
         * Labels can be used to specify the target of a
         * branch as well.
         * Here, forward and backward branches are
         * illustrated.
         */

        .method public labelTest()V 3 {

L1:             goto L2;


L2:             goto L1;


            goto_w L1;


            goto_w L3;



L3:             return;
        }

        /*
         * This method illustrates the use of each Java Card platform
         * instruction for version 2.2.1.
         * Mnenomics are case insensitive.
         *
         * See the Java Card virtual machine specification for
         * the specification of
         * each instruction.
         */

        .method public instructions()V 4 {

            aaload;
            aastore;
            aconst_null;
   aload 0;
   aload_0;
   aload_1;
   aload_2;
   aload_3;
   anewarray 0;
   areturn;
   arraylength;
```

```
astore 0;
astore_0;
astore_1;
astore_2;
astore_3;
athrow;
baload;
bastore;
bipush 0;
bspush 0;
checkcast 10 0;
checkcast 11 0;
checkcast 12 0;
checkcast 13 0;
checkcast 14 0;
dup2;
dup;
dup_x 0x11;
getfield_a 1;
getfield_a_this 1;
getfield_a_w 1;
getfield_b 1;
getfield_b_this 1;
getfield_b_w 1;
getfield_i 1;
getfield_i_this 1;
getfield_i_w 1;
getfield_s 1;
getfield_s_this 1;
getfield_s_w 1;
getstatic_a 4;
getstatic_b 4;
getstatic_i 4;
getstatic_s 4;
goto 0;
goto_w 0;
i2b;
i2s;
iadd;
iaload;
iand;
iastore;
icmp;
iconst_0;
iconst_1;
iconst_2;
iconst_3;
iconst_4;
iconst_5;
```

```
iconst_m1;
idiv;
if_acmpeq 0;
if_acmpeq_w 0;
if_acmpne 0;
if_acmpne_w 0;
if_scmpeq 0;
if_scmpeq_w 0;
if_scmpge 0;
if_scmpge_w 0;
if_scmpgt 0;
if_scmpgt_w 0;
if_scmple 0;
if_scmple_w 0;
if_scmplt 0;
if_scmplt_w 0;
if_scmpne 0;
if_scmpne_w 0;
ifeq 0;
ifeq_w 0;
ifge 0;
ifge_w 0;
ifgt 0;
ifgt_w 0;
ifle 0;
ifle_w 0;
iflt 0;
iflt_w 0;
ifne 0;
ifne_w 0;
ifnonnull 0;
ifnonnull_w 0;
ifnull 0;
ifnull_w 0;
iinc 0 0;
iinc_w 0 0;
iipush 0;
iload 0;
iload_0;
iload_1;
iload_2;
iload_3;
ilookupswitch 0 1 0 0;
impdep1;
impdep2;
imul;
ineg;
instanceof 10 0;
instanceof 11 0;
```

```
instanceof 12 0;
instanceof 13 0;
instanceof 14 0;
invokeinterface 0 0 0;
invokespecial 3;   // superMethodRef
invokespecial 5;   // staticMethodRef
invokestatic 5;
invokevirtual 2;
ior;
irem;
ireturn;
ishl;
ishr;
istore 0;
istore_0;
istore_1;
istore_2;
istore_3;
isub;
itableswitch 0 0 1 0 0;
iushr;
ixor;
jsr 0;
new 0;
newarray 10;
newarray 11;
newarray 12;
newarray 13;
newarray boolean[];        // array types may be decared
 numerically or
newarray byte[];        // symbolically.
newarray short[];
newarray int[];
nop;
pop2;
pop;
putfield_a 1;
putfield_a_this 1;
putfield_a_w 1;
putfield_b 1;
putfield_b_this 1;
putfield_b_w 1;
putfield_i 1;
putfield_i_this 1;
putfield_i_w 1;
putfield_s 1;
putfield_s_this 1;
putfield_s_w 1;
putstatic_a 4;
```

```
putstatic_b 4;
putstatic_i 4;
putstatic_s 4;
ret 0;
return;
s2b;
s2i;
sadd;
saload;
sand;
sastore;
sconst_0;
sconst_1;
sconst_2;
sconst_3;
sconst_4;
sconst_5;
sconst_m1;
sdiv;
sinc 0 0;
sinc_w 0 0;
sipush 0;
sload 0;
sload_0;
sload_1;
sload_2;
sload_3;
slookupswitch 0 1 0 0;
smul;
sneg;
sor;
srem;
sreturn;
sshl;
sshr;
sspush 0;
sstore 0;
sstore_0;
sstore_1;
sstore_2;
sstore_3;
ssub;
stableswitch 0 0 1 0 0;
sushr;
swap_x 0x11;
sxor;

     }
 }
```

```
    .class public test2 2 extends 0.0 {

        .publicMethodTable 0 {}
   equals(Ljava/lang/Object;)Z;
        .packageMethodTable 0 {}
        .method public static install([BSB)V 0 {
   .stack 0;
   .locals 0;
}
   return;
        }
    }

    .class public test3 3 extends test2 {

    /*
    * Declaration of static array initialization is done the same way
    * as in Java
    * Only one dimensional arrays are allowed in the
    * Java Card platform
    * Array of zero elements, 1 element, n elements
    */
    .fields {
        public static final int[] array0 0 = {}; //  [I
        public static final byte[] array1 1 = {17}; //  [B
        public static short[] arrayn 2 = {1,2,3,...,n}; //  [S
    }

        .publicMethodTable 0 {}
   equals(Ljava/lang/Object;)Z;
        .packageMethodTable 0 {}
        .method public static install([BSB)V 0 {
   .stack 0;
   .locals 0;
   return;
        }
    }

    .interface public test4 4 extends 0.0 {
    }

}
```

APPENDIX **B**

CAP File Manifest File Syntax

One of the files generated by the Converter is the CAP file. The CAP file utilizes the JAR file format, and contains a set of components which describe a Java package. In addition to the components, the CAP file also contains the manifest file: `META-INF/MANIFEST.MF`. The manifest file provides additional human-readable information regarding the contents of the CAP file and the package that it represents. This information can be used to facilitate the distribution and processing of the CAP file.

The information in the manifest file is presented in *name:value* pairs. These *name:value* pairs are described in TABLE 38.

TABLE 38 Name:Value Pairs in the `MANIFEST.MF` File

Name	Value
`Java-Card-CAP-Creation-Time`	Creation time of CAP file. For example: Tue Jan 15 11:07:55 PST 2002 The format of the time stamp is operating system-dependent.
`Java-Card-Converter-Version`	The version of the converter tool. For example: `1.3`.
`Java-Card-Converter-Provider`	Provider of the converter tool. For example: Sun Microsystems, Inc.
`Java-Card-CAP-File-Version`	CAP file *major.minor* version. For example: 2.1.
`Java-Card-Package-Version`	The *major.minor* version of package. For example: `1.0`
`Java-Card-Package-AID`	AID for the package. For example: 0xa0:0x00:0x00:0x00:0x62: 0x03:0x01:0x0c:0x07

TABLE 38 Name:Value Pairs in the `MANIFEST.MF` File

Name	Value
`Java-Card-Package-Name`	The fully-qualified package name in dot (.) format. For example: `javacard.framework`
`Java-Card-Applet-<`*n*`>-AID`	The AID for applet *n*. For example: `0xa0:0x00:0x00:0x00:0x62:0x03:0x01:0x0c:0x07:0x05`
`Java-Card-Applet-<`*n*`>-Name`	Simple class name for applet *n*. For example: `MyApplet`
`Java-Card-Import-Package-<`*n*`>-AID`	The AID for imported package *n*. For example: `0xa0:0x00:0x00:0x00:0x62:0x00:0x01`
`Java-Card-Import-Package-<`*n*`>-Version`	The *major.minor* version of imported package *n*. For example: `1.0`
`Java-Card-Integer-Support-Required`	Can be `TRUE` or `FALSE`. The value is `TRUE` if the package requires integer support.

Note the following additional information about the properties in the manifest file:

- The names `Java-Card-Applet-<`*n*`>-AID` and `Java-Card-Applet-<`*n*`>-Name` refer to the same applet.
- The converter assigns numbers for the `Java-Card-Applet-<`*n*`>-NAME` and `Java-Card-Applet-<`*n*`>-AID` names in sequential order, beginning with 1.
- The names `Java-Card-Imported-Package-<`*n*`>-AID` and `Java-Card-Imported-Package-<`*n*`>-Version` refer to the same package.
- The converter assigns numbers for the `Java-Card-Imported-Package-<`*n*`>-AID` and `Java-Card-Imported-Package-<`*n*`>-AID` names in sequential order, beginning with 1.

Sample Manifest File

The following code sample illustrates the manifest file that the converter generates when it converts package `jcard.applications`. This package contains two applets: `MyClass1` and `MyClass2`.

```
Manifest-Version: 1.0
Created-By: 1.3.1 (Sun Microsystems Inc.)

Java-Card-CAP-Creation-Time: Tue Jan 15 11:07:55 PST 2002
Java-Card-Converter-Version: 1.3
```

```
Java-Card-Converter-Provider: Sun Microsystems, Inc.
Java-Card-CAP-File-Version: 2.1
Java-Card-Package-Version: 1.0
Java-Card-Package-Name: jcard.applications
Java-Card-Package-AID: 0xa0:0x00:0x00:0x00:0x62:0x03:0x01:0x0c:0x07
Java-Card-Applet-1-Name: MyClass1
Java-Card-Applet-1-AID:
   0xa0:0x00:0x00:0x00:0x62:0x03:0x01:0x0c:0x07:0x05
Java-Card-Applet-2-Name: MyClass2
Java-Card-Applet-2-AID:
   0xa0:0x00:0x00:0x00:0x62:0x03:0x01:0x0c:0x07:0x06
Java-Card-Imported-Package-1-AID: 0xa0:0x00:0x00:0x00:0x62:0x00:0x01
Java-Card-Imported-Package-1-Version: 1.0
Java-Card-Imported-Package-2-AID: 0xa0:0x00:0x00:0x00:0x62:0x01:0x01
Java-Card-Imported-Package-2-Version: 1.1
Java-Card-Integer-Support-Required: TRUE
```

APPENDIX C

Using the Large Address Space

Allowing your applications to take advantage of the large address capabilities of the Java Card reference implementation, version 2.2.1, requires careful planning and programming. Some size limitations still exist within the Reference Implementation. The way that you structure large applications, as well as applications that manage large amounts of data, determines how the large address space can be exploited.

The following sections describe two of the ways in which you can take advantage of large memory storage in smart cards:

Programming Large Applications and Libraries

The key to writing large applications for the Java Card platform is to divide the code into individual package units. The most important limitation on a package is the 64KB limitation on the maximum component size. This is especially true for the Method component: if the size of an application's Method component exceeds 64KB, then the Java Card converter will not process the package and will return an error.

You can overcome the component size limitation by dividing the application into separate application and library components. The Java Card platform has the ability to support library packages Library packages contain code which can be linked and reused by several applications. By dividing the functionality of a given application into application and library packages, you can increase the size of the components. Keep in mind that there are important differences between library packages and applet packages:

- In a library package, all public fields are available to other packages for linking.
- In an applet package, only interactions through a shareable interface are allowed by the firewall.

Therefore, you should not place sensitive or exclusive-use code in a library package. It should be placed in an applet package, instead.

Handling a Package as a Separate Code Space

Several applications and API functionality can be installed in the smart card simultaneously by handling each package as a separate code space. This technique will let you exceed the 64KB limit, and provide full Java Card API functionality and support for complex applications requiring larger amounts of code.

Storing Large Amounts of Data

The most efficient way to take advantage of the large memory space is to use it to store data. Today's applications are required to securely store ever-growing amounts of information about the cardholder or network identity. This information includes certificates, images, security keys, and biometric and biographic information.

This information sometimes requires large amounts of storage. Before 2.2.1, versions of the Java Card reference implementation had to save downloaded applications or user data in valuable persistent memory space. Sometimes, the amount of memory space required was insufficient for some applications. However, the memory access schemes introduced with version 2.2.1 allow applications to store large amounts of information, while still conforming to the Java Card specification.

The Java Card specification does not impose any requirements on object location or total object heap space used on the card. It specifies only that each object must be accessible by using a 16-bit reference. It also imposes some limitations on the amount of information an individual object is capable of storing, by using the number of fields or the count of array elements. Because of this loose association, it is possible for any given implementation to control how an object's information is stored, and how much data these objects can collectively hold.

The Java Card reference implementation, version 2.2.1, allows you to use all of the available persistent memory space to store object information. By allowing you to separate data storage into distinct array and object types, this reference implementation allows you to store the large amounts of data demanded by today's applications.

Example: The photocard Demo Applet

The `photocard` demo applet (included with the Java Card reference implementation, version 2.2.1) is an example of an application that takes advantage of the large address space capabilities.

The `photocard` applet performs a very simple task: it stores pictures inside the smart card and retrieves them by using a Java Card RMI interface. For more information on the `photocard` demo applet and how to run it, see "Photo Card Demo" on page 30.

```
public interface PhotoCard extends Remote {

    public static final short NO_SPACE_AVAILABLE = (short)0x6000;
    public static final short NO_PHOTO_STORED  = (short)0x6001;
    public static final short INVALID_PHOTO_ID = (short)0x6002;
    public static final short INVALID_ARGUMENT   = (short)0x6003;
    public static final short MAX_SIZE           = (short)0x7FFF;
    public static final short MAX_PHOTO_COUNT    = (short)4;
    public static final short MAX_BUFFER_BYTES   = (short)96;


    public short requestPhotoStorage(short size)
        throws RemoteException, UserException;

    public void loadPhoto(short photoID, byte[] data,
        short size, short offset, boolean more)
        throws RemoteException, UserException;

    public void deletePhoto(short photoID)
        throws RemoteException, UserException;

    public short getPhotoSize(short photoID)
        throws RemoteException, UserException;

    public byte[] getPhoto(short photoID, short offset, short size)
        throws RemoteException, UserException;
}
```

To store the images, an array of arrays has been defined:

```
// Array containing photo objects
    private Object[] photos;
```

Each image is stored inside an array, and each array can grow up to 32,767 elements in size.

```
for (short i = (short)0; i < (short)MAX_PHOTO_COUNT;i++) {
    byte[] thePhoto = (byte[])photos[i];

    if (photos[i] == null) {
        photos[i] = new byte[size];
        return (short)(i + 1);
    }
}
UserException.throwIt(NO_SPACE_AVAILABLE);
```

The array can be randomly accessed, as needed. In this implementation, the arrays are defined as byte arrays, however, they could also have been defined as integer arrays.

```
byte[] selPhoto = (byte[])photos[(short)(photoID - (short)1)];
...
Util.arrayCopy(selPhoto, offset, buffer, (short)0, size);
return buffer;
```

The collection of arrays (more than two arrays would be required in this case) can easily hold far more than 64KB of data. Storing this amount of information should not be a problem, provided that enough mutable persistent memory is configured in the C-language Java Card RE.

Notes on the photocard Applet

The `photocard` applet employs a collection of arrays to store large amounts of data. The arrays allow the applet to take advantage of the platform's capabilities by transparently storing data.

The coding and design of applications that use the large address space to access memory must adhere to the target platform's requirements.

As smart cards have limited resources, code cannot be guaranteed to behave identically on different cards. For example, if you run the `photocard` applet on a card with less mutable persistent memory available for storage, then it might run out of memory space when it attempts to store the images. A given set of inputs might not produce the same set of outputs in a C-language Java Card RE with different characteristics. The applet code must account for any different implementation-specific behavior.

APPENDIX **D**

Reference Implementation of Java Card RMI Client-Side API

This appendix contains documentation for the Reference Implementation of the Java Card RMI client-side API.

Package ocfrmiclientimpl

The package `com.sun.javacard.ocfrmiclientimpl` contains implementations of the classes and interfaces from the package `com.sun.javacard.javax.smartcard.rmiclient`. It also contains implementations of classes and interfaces inherited from `java.rmi.server.RemoteRef` and `opencard.core.service.CardType`.

This implementation of `ocfrmiclientimpl` requires that an OCF framework is installed on the terminal.

- **class JCCardObjectFactory**–An implementation of the `CardObjectFactory` that processes the data returned from the card in the format defined in the *Runtime Environment (JCRE) Specification for the Java Card™ Platform, Version 2.2.1*. Any object references must contain class names.
- **class JCCardProxyFactory**–The `JCCardProxyFactory` class is similar to `JCCardObjectFactory`, but processes references containing lists of names. `JCCardProxyFactory` uses the SDK1.4.1 proxy mechanism to generate proxies dynamically.
- **class JCRemoteRefImpl**–An implementation of interface `java.rmi.server.RemoteRef`. These remote references can work with stubs generated by the RMIC compiler with the `-v1.2` option.

The main method is:

```
public Object invoke(Remote remote, Method method, Object[]
params, long unused) throws IOException, RemoteException,
Exception
```

This method prepares the outgoing APDU, passes it to `CardAccessor`, and then uses `CardObjectFactory` to parse the returned APDU and instantiate the returned object or throw an exception.

- **class OCFCardAccessor**–A simple implementation of the `CardAccessor` interface. It passes the APDU to an underlying `CardTerminal` and returns the result received from the `CardTerminal`. Here, `CardTerminal` is the OpenCard Framework's representation of a physical card terminal.

  A client program usually supplies its own `CardAccessor` which extends `OCFCardAccessor` and performs additional transformations and checks of the data.
- **class JavaCardType**–A tagging (empty) class; used to notify the OCF that the client framework expects to communicate with the Java Card API class `OCFCardAccessorFactory`. This class extends `opencard.core.service.CardType`.
- **class OCFCardAccessorFactory**–A factory returning an instance of the `OCFCardAccessor`. Required to register the `OCFCardAccessor` with the OCF.

API Documentation

The remainder of this appendix contains API documentation for the reference implementation of the Java Card RMI client-side API, package `com.sun.javacard.ocfrmiclientimpl`. This package demonstrates remote stub customization using the RMIC compiler generated stubs and OCF-based card access for Java Card applets.

Overview

Package Summary	
Packages	
com.sun.javacard.ocfrmiclientimpl	Provides implementation of classes and interfaces defined in com.sun.javacard.javax.smartcard.rmiclient.

Class Hierarchy

```
java.lang.Object
    com.sun.javacard.javax.smartcard.rmiclient.CardObjectFactory
        com.sun.javacard.ocfrmiclientimpl.JCCardObjectFactory
        com.sun.javacard.ocfrmiclientimpl.JCCardProxyFactory
    opencard.core.service.CardService
        com.sun.javacard.ocfrmiclientimpl.OCFCardAccessor (implements com.sun.javacard.
javax.smartcard.rmiclient.CardAccessor)
    opencard.core.service.CardServiceFactory
        com.sun.javacard.ocfrmiclientimpl.OCFCardAccessorFactory
    opencard.core.service.CardType
        com.sun.javacard.ocfrmiclientimpl.JavaCardType
    com.sun.javacard.ocfrmiclientimpl.JCRemoteRefImpl (implements java.rmi.server.
RemoteRef, java.lang.reflect.InvocationHandler)
```

Package
com.sun.javacard.ocfrmiclientimpl

Description

Provides implementation of classes and interfaces defined in `com.sun.javacard.javax.smartcard.rmiclient`.

This implementation depends on the OCF1.2 as the client framework.

Class Summary	
Classes	
`JavaCardType`	An instance of this class is used by the `OCFCardAccessorFactory` to denote that the smart card currently being accessed may be Java Card-compliant.
`JCCardObjectFactory`	Processes the data returned from the card in the format defined for Java Card RMI.
`JCCardProxyFactory`	Processes the data returned from the card in the format defined for Java Card RMI.
`JCRemoteRefImpl`	Represents a reference to a card object.
`OCFCardAccessor`	Passes APDUs between client program and `CardTerminal`.
`OCFCardAccessorFactory`	The OCFCardAccessorFactory class creates the `OCFCardAccessor` instance which is used by terminal client applications to initiate and conduct a Java Card RMI based dialogue with the smart card.

com.sun.javacard.ocfrmiclientimpl
JavaCardType

Declaration

```
public class JavaCardType extends opencard.core.service.CardType

java.lang.Object
  |
  +--opencard.core.service.CardType
        |
        +--com.sun.javacard.ocfrmiclientimpl.JavaCardType
```

Description

An instance of this class is used by the `OCFCardAccessorFactory` to denote that the smart card currently being accessed may be Java Card technology-compliant.

Member Summary	
Constructors	
	`JavaCardType()` Creates new `JavaCardType`

Inherited Member Summary

Fields inherited from class `CardType`
`UNSUPPORTED`
Methods inherited from class `CardType`
`getInfo(), getType(), setInfo(Object)`
Methods inherited from class `Object`
`clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(), wait()`

Constructors

JavaCardType()

```
public JavaCardType()
```

Creates new `JavaCardType`

com.sun.javacard.ocfrmiclientimpl
JCCardObjectFactory

Declaration

```
public class JCCardObjectFactory extends com.sun.javacard.javax.smartcard.rmiclient.
    CardObjectFactory

java.lang.Object
  |
  +--com.sun.javacard.javax.smartcard.rmiclient.CardObjectFactory
        |
        +--com.sun.javacard.ocfrmiclientimpl.JCCardObjectFactory
```

Description

Processes the data returned from the card in the format defined for Java Card RMI. Object references must contain class names. Extends `CardObjectFactory`.

Member Summary	
Constructors	
	`JCCardObjectFactory(com.sun.javacard.javax.smartcard.rmiclient.CardAccessor ca)` The constructor.
Methods	
protected java.rmi.Remote	`getRemoteObject(byte[] buffer, int tagOffset)` Creates the stub instance for object reference returned from the card, assuming the card returned a reference with class name.
byte	`getRemoteRefFormat()` Returns constant `REF_FORMAT_CLASS` defined in class `com.sun.javacard.javax.smartcard.rmiclient.CardObjectFactory`.

Inherited Member Summary
Fields inherited from class `CardObjectFactory`
`REF_FORMAT_CLASS, REF_FORMAT_INTERFACES, REF_FORMAT_NONE, cardAccessor`
Methods inherited from class `CardObjectFactory`

Inherited Member Summary
`getINSByte(), getObject(byte[], int, Class), setINSByte(byte)`
Methods inherited from class `Object`
`clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(), wait()`

Constructors

JCCardObjectFactory(CardAccessor)

```
public JCCardObjectFactory(com.sun.javacard.javax.smartcard.rmiclient.CardAccessor
            ca)
```

The constructor.

Parameters:

`ca` - The `CardAccessor` of the current card session. It is passed to the Stubs/Proxies created by the Factory.

Methods

getRemoteObject(byte[], int)

```
protected java.rmi.Remote getRemoteObject(byte[] buffer, int tagOffset)
            throws Exception
```

Creates the stub instance for object reference returned from the card, assuming the card returned a reference with class name.

Overrides: `getRemoteObject` in class `CardObjectFactory`

Parameters:

`buffer` - APDU buffer

`tagOffset` - Offset to tag

Returns: The resulting stub.

Throws:

`java.lang.Exception` - Failed to instantiate a stub

getRemoteRefFormat()

```
public byte getRemoteRefFormat()
```

Returns constant `REF_FORMAT_CLASS` defined in class `com.sun.javacard.javax.smartcard.rmiclient.CardObjectFactory`.

Overrides: `getRemoteRefFormat` in class `CardObjectFactory`

Returns: REF_FORMAT_CLASS value defined above

com.sun.javacard.ocfrmiclientimpl
JCCardProxyFactory

Declaration

```
public class JCCardProxyFactory extends com.sun.javacard.javax.smartcard.rmiclient.
    CardObjectFactory

java.lang.Object
  |
  +--com.sun.javacard.javax.smartcard.rmiclient.CardObjectFactory
        |
        +--com.sun.javacard.ocfrmiclientimpl.JCCardProxyFactory
```

Description

Processes the data returned from the card in the format defined for Java Card RMI. Object references must contain lists of interface names. Extends `CardObjectFactory`.

Member Summary	
Constructors	
	`JCCardProxyFactory(com.sun.javacard.javax.smartcard.rmiclient.CardAccessor ca)` Constructor for the factory.
Methods	
`protected java.rmi.Remote`	`getRemoteObject(byte[] buffer, int tagOffset)` Creates the stub instance for object reference returned from the card, assuming the card returned a reference with list of interface names.
`byte`	`getRemoteRefFormat()` Returns constant `REF_FORMAT_INTERFACES` defined in class `com.sun.javacard.javax.smartcard.rmiclient.CardObjectFactory`.

Inherited Member Summary
Fields inherited from class `CardObjectFactory`
`REF_FORMAT_CLASS, REF_FORMAT_INTERFACES, REF_FORMAT_NONE, cardAccessor`
Methods inherited from class `CardObjectFactory`
`getINSByte(), getObject(byte[], int, Class), setINSByte(byte)`

Inherited Member Summary
Methods inherited from class `Object` `clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(), wait()`

Constructors

JCCardProxyFactory(CardAccessor)

```
public JCCardProxyFactory(com.sun.javacard.javax.smartcard.rmiclient.CardAccessor
            ca)
```

Constructor for the factory.

Parameters:

`ca` - `CardAccessor` for the current session.

Methods

getRemoteRefFormat()

```
public byte getRemoteRefFormat()
```

Returns constant `REF_FORMAT_INTERFACES` defined in class `com.sun.javacard.javax.smartcard.rmiclient.CardObjectFactory`.

Overrides: `getRemoteRefFormat` in class `CardObjectFactory`

Returns: The format constant.

getRemoteObject(byte[], int)

```
protected java.rmi.Remote getRemoteObject(byte[] buffer, int tagOffset)
            throws Exception
```

Creates the stub instance for object reference returned from the card, assuming the card returned a reference with list of interface names.

Overrides: `getRemoteObject` in class `CardObjectFactory`

Parameters:

`buffer` - `APDU` buffer.

`tagOffset` - Offset to tag.

Returns: The instance of the proxy.

Throws:

`java.lang.Exception` - Thrown if the proxy instance cannot be instantiated

com.sun.javacard.ocfrmiclientimpl
JCRemoteRefImpl

Declaration

```
public class JCRemoteRefImpl implements java.rmi.server.RemoteRef, java.lang.reflect.
    InvocationHandler

java.lang.Object
  |
  +--com.sun.javacard.ocfrmiclientimpl.JCRemoteRefImpl
```

All Implemented Interfaces: `java.io.Externalizable`, `java.lang.reflect.InvocationHandler`, `java.rmi.server.RemoteRef`, `java.io.Serializable`

Description

Represents a reference to a card object. This class is a Java Card RMI implementation of the `RemoteRef` interface. It is used in conjunction with Java RMIC generated stubs or dynamically generated proxies for Java Card RMI method invocations.

Member Summary	
Constructors	
	`JCRemoteRefImpl(short objID, java.lang.String a_string, com.sun.javacard.javax.smartcard.rmiclient.CardAccessor cA, com.sun.javacard.javax.smartcard.rmiclient.CardObjectFactory cOF)` Creates new `JCRemoteRefImpl`
Methods	
`void`	`done(java.rmi.server.RemoteCall remoteCall)` Deprecated and not implemented
`java.lang.String`	`getRefClass(java.io.ObjectOutput objectOutput)` Unsupported operation.
`java.lang.Object`	`invoke(java.lang.Object obj, java.lang.reflect.Method method, java.lang.Object params)` This method is used by dynamically generated proxies.
`void`	`invoke(java.rmi.server.RemoteCall remoteCall)` Deprecated and not implemented
`java.lang.Object`	`invoke(java.rmi.Remote remote, java.lang.reflect.Method method, java.lang.Object params, long unused)` This method is used by rmic-generated stubs.

Member Summary	
java.rmi.server.RemoteCall	newCall(java.rmi.server.RemoteObject remoteObject, java.rmi.server.Operation operation, int param, long param3) Deprecated and not implemented
void	readExternal(java.io.ObjectInput objectInput) Unsupported operation.
boolean	remoteEquals(java.rmi.server.RemoteRef remoteRef) Compares two remote objects for being identical.
int	remoteHashCode() Unsupported operation.
java.lang.String	remoteToString() String representation of remote object.
void	writeExternal(java.io.ObjectOutput objectOutput) Unsupported operation.

Inherited Member Summary

Fields inherited from interface `RemoteRef`

`packagePrefix, serialVersionUID`

Methods inherited from class `Object`

`clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(), wait()`

Constructors

JCRemoteRefImpl(short, String, CardAccessor, CardObjectFactory)

```
public JCRemoteRefImpl(short objID, java.lang.String a_string, com.sun.javacard.
            javax.smartcard.rmiclient.CardAccessor cA, com.sun.javacard.javax.
            smartcard.rmiclient.CardObjectFactory cOF)
```

Creates new `JCRemoteRefImpl`

Parameters:

`objID` - 2 byte Object ID from card remote reference descriptor

`a_string` - Anticollision string for the class of the remote object

`cA` - `CardAccessor`

`cOF` - `CardObjectFactory`

Methods

getRefClass(ObjectOutput)

```
public java.lang.String getRefClass(java.io.ObjectOutput objectOutput)
```

Unsupported operation.

Specified By: `getRefClass` in interface `RemoteRef`

Parameters:

invoke(Remote, Method, Object[], long)

```
public java.lang.Object invoke(java.rmi.Remote remote, java.lang.reflect.
            Method method, java.lang.Object[] params, long unused)
            throws IOException, RemoteException, Exception
```

This method is used by rmic-generated stubs.

Specified By: `invoke` in interface `RemoteRef`

Parameters:

`remote` - Reference to the stub - not used.

`method` - `java.reflect.Method` object containing information about the method to be invoked.

`params` - Array of parameters. Primitives are wrapped.

`unused` - rmic-generated hash of the method. Not used.

Returns: The result returned from the card.

Throws:

`java.io.IOException` - If a communication error occurred.

`java.rmi.RemoteException` - If an RMI error occurred.

`java.lang.Exception` - Exception corresponding to the one that was thrown on the card.

remoteHashCode()

```
public int remoteHashCode()
```

Unsupported operation.

Specified By: `remoteHashCode` in interface `RemoteRef`

Returns: A number which is the same for all objects.

remoteToString()

```
public java.lang.String remoteToString()
```

String representation of remote object.

Specified By: `remoteToString` in interface `RemoteRef`

Returns: A `String` representation of the remote object.

readExternal(ObjectInput)

```
public void readExternal(java.io.ObjectInput objectInput)
            throws IOException, ClassNotFoundException
```

Unsupported operation.

Specified By: `readExternal` in interface `Externalizable`

Parameters:

Throws:

`java.io.IOException`, `java.lang.ClassNotFoundException`

writeExternal(ObjectOutput)

```
public void writeExternal(java.io.ObjectOutput objectOutput)
            throws IOException
```

Unsupported operation.

Specified By: `writeExternal` in interface `Externalizable`

Parameters:

Throws:

`java.io.IOException`

newCall(RemoteObject, Operation[], int, long)

```
public java.rmi.server.RemoteCall newCall(java.rmi.server.RemoteObject
            remoteObject, java.rmi.server.Operation[] operation, int param,
            long param3)
            throws RemoteException
```

Deprecated. Deprecated and not implemented

Specified By: `newCall` in interface `RemoteRef`

Parameters:

Throws:

`java.rmi.RemoteException`

invoke(RemoteCall)

```
public void invoke(java.rmi.server.RemoteCall remoteCall)
            throws Exception
```

Deprecated. Deprecated and not implemented

Specified By: `invoke` in interface `RemoteRef`

Parameters:

Throws:
`java.lang.Exception`

remoteEquals(RemoteRef)

```
public boolean remoteEquals(java.rmi.server.RemoteRef remoteRef)
```

Compares two remote objects for being identical.

Specified By: `remoteEquals` in interface `RemoteRef`

Parameters:
`remoteRef` - `RemoteRef` to the other remote object.

Returns: `true` if corresponding remote objects are identical.

done(RemoteCall)

```
public void done(java.rmi.server.RemoteCall remoteCall)
            throws RemoteException
```

Deprecated. Deprecated and not implemented

Specified By: `done` in interface `RemoteRef`

Parameters:

Throws:
`java.rmi.RemoteException`

invoke(Object, Method, Object[])

```
public java.lang.Object invoke(java.lang.Object obj, java.lang.reflect.
            Method method, java.lang.Object[] params)
            throws IOException, RemoteException, Throwable
```

This method is used by dynamically generated proxies.

Specified By: `invoke` in interface `InvocationHandler`

Parameters:
`obj` - The reference to the Proxy - not used.

`method` - `Method` object containing information about the method.

`params` - Array of parameters for the method.

Returns: The result returned from the card.

Throws:

`java.io.IOException` - If a communication error occurred.

`java.rmi.RemoteException` - If an RMI error occurred.

`java.lang.Throwable` - Exception corresponding to the one that was thrown on the card.

com.sun.javacard.ocfrmiclientimpl
OCFCardAccessor

Declaration

```
public class OCFCardAccessor extends opencard.core.service.CardService implements com.
    sun.javacard.javax.smartcard.rmiclient.CardAccessor

java.lang.Object
  |
  +--opencard.core.service.CardService
        |
        +--com.sun.javacard.ocfrmiclientimpl.OCFCardAccessor
```

All Implemented Interfaces: `com.sun.javacard.javax.smartcard.rmiclient.CardAccessor`

Description

Passes APDUs between client program and `CardTerminal`. Client programs usually supply their own `CardAccessor` extending this class and performing additional transformations and checks of the data. Implements the `CardAccessor` interface.

Member Summary	
Constructors	
	`OCFCardAccessor()` Creates new OCFCardAccessor
Methods	
`byte[]`	`exchangeAPDU(byte[] sendData)` Exchanges APDU with the card.
`short`	`getSessionIdentifier()` A number identifying the current session.

Inherited Member Summary
Methods inherited from class `CardService`
`allocateCardChannel()`, `getCHVDialog()`, `getCard()`, `getCardChannel()`, `initialize(CardServiceScheduler, SmartCard, boolean)`, `releaseCardChannel()`, `setCHVDialog(CHVDialog)`, `setCardChannel(CardChannel)`

Inherited Member Summary
Methods inherited from class `Object` `clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(), wait()`

Constructors

OCFCardAccessor()

```
public OCFCardAccessor()
```

Creates new OCFCardAccessor

Methods

exchangeAPDU(byte[])

```
public byte[] exchangeAPDU(byte[] sendData)
            throws IOException
```

Exchanges APDU with the card.

Specified By: `exchangeAPDU` in interface `CardAccessor`

Parameters:
`sendData` - The outgoing `APDU`

Returns: The response `APDU`

Throws:
IOException - if a communication error occurs.

`java.io.IOException`

getSessionIdentifier()

```
public short getSessionIdentifier()
```

A number identifying the current session.

Specified By: `getSessionIdentifier` in interface `CardAccessor`

Returns: session ID.

com.sun.javacard.ocfrmiclientimpl
OCFCardAccessorFactory

Declaration

```
public class OCFCardAccessorFactory extends opencard.core.service.CardServiceFactory

java.lang.Object
  |
  +--opencard.core.service.CardServiceFactory
        |
        +--com.sun.javacard.ocfrmiclientimpl.OCFCardAccessorFactory
```

Description

The OCFCardAccessorFactory class creates the `OCFCardAccessor` instance which is used by terminal client applications to initiate and conduct a Java Card RMI based dialogue with the smart card. The methods in this class are intended to be invoked by the OCF `CardServiceRegistry` class. Java Card RMI Client applications should access the `SmartCard` class to obtain instances of `OCFCardAccessor`.

See Also: `OCFCardAccessor`

Member Summary	
Constructors	
	`OCFCardAccessorFactory()` Creates new OCFCardAccessorFactory
Methods	
`protected opencard.core.service.CardType`	`getCardType(opencard.core.terminal.CardID cid, opencard.core.service.CardServiceScheduler scheduler)` This method examines the `CardID` object (containing the ATR returned by the Card) and checks if the card could be Java Card technology-compliant. If so, this method returns a `JavaCardType` object.
`protected java.util.Enumeration`	`getClasses(opencard.core.service.CardType type)` If the input parameter is a `JavaCardType` object, this method returns an enumeration object with the `OCFCardAccessor` object listed.

Inherited Member Summary

Methods inherited from class `CardServiceFactory`
`getCardServiceInstance(Class, CardType, CardServiceScheduler, SmartCard, boolean), getClassFor(Class, CardType), newCardServiceInstance(Class, CardType, CardServiceScheduler, SmartCard, boolean)`
Methods inherited from class `Object`
`clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(), wait()`

Constructors

OCFCardAccessorFactory()

```
public OCFCardAccessorFactory()
```

Creates new OCFCardAccessorFactory

Methods

getCardType(CardID, CardServiceScheduler)

```
protected opencard.core.service.CardType getCardType(opencard.core.terminal.CardID
            cid, opencard.core.service.CardServiceScheduler scheduler)
```

This method examines the `CardID` object (containing the ATR returned by the Card) and checks if the card could be Java Card technology-compliant. If so, this method returns a `JavaCardType` object.

Overrides: `getCardType` in class `CardServiceFactory`

Parameters:

`cid` - `CardID` received from a Card `Slot`.

`scheduler` - `CardServiceScheduler` that can be used to communicate with the card to determine its type.

Returns: A `JavaCardType` if the factory can instantiate services for this card. `CardType.UNSUPPORTED` if the factory does not know the card.

getClasses(CardType)

```
protected java.util.Enumeration getClasses(opencard.core.service.CardType type)
```

If the input parameter is a `JavaCardType` object, this method returns an enumeration object with the `OCFCardAccessor` object listed. Subclasses of this class may add subclasses of `OCFCardAccessor` to this list by using the `add` method.

Overrides: `getClasses` in class `CardServiceFactory`

Parameters:

`type` - The `CardType` of the smart card for which the enumeration is requested.

Returns: An `Enumeration` of `OCFCardAccessor` class objects

Index

D

E

G

I

J

N

O

P

R

S

U

V

W