Java Card™ 2.1.2 Development Kit User's Guide

For the Binary Release

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303 USA
650 960-1300 fax 650 969-9131

Revision 1.0
Apr 11, 2001

Contents

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 *Java Card‘ 2.1.2 Development Kit User’s Guide* contains information on how to install and use the Java Card Development Kit tools comprising this 2.1.2 release.

Who Should Use This Book

The *Java Card™ 2.1.2 Development Kit User’s Guide* is targeted at developers who are creating applets using the *Java Card™ 2.1.1 Application Programming Interface*, Sun Microsystems, Inc., and also at developers who are considering creating a vendor-specific framework based on the Java Card 2.1.1 technology specifications.

Note – Even though this release of the Development Kit is version 2.1.2, the Java Card technology specifications are unchanged from version 2.1.1.

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 technology 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 Java Card Development Kit," provides an overview of the Java Card Development Kit and the tools in the kit.

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

Chapter 3, "Java Card Samples and Demonstrations," shows the suggested sequence of steps to run the two demonstration masks included with this release.

Chapter 4, "Using the JCWDE," provides an overview of the Java Card Workstation Development Environment and details of how to run it.

Chapter 5, "Using the Converter," provides an overview of the Converter and details of how to run it.

Chapter 6, "Using the Off-Card Verifier," provides an overview of the offcard verifier tool and details of running it.

Chapter 7, "Using capgen," describes how to use the capgen utility.

Chapter 8, "Using capdump," describes how to use the capdump utility.

Chapter 9, "Using maskgen," describes how to use the maskgen utility.

Chapter 10, "Using the C-JCRE," describes how to use the C-JCRE interpreter.

Chapter 11, "Using the Installer," describes how to download and create applets using the installer.

Chapter 12, "Using the APDUTool," describes using this tool to send APDUs to the C-JCRE.

Appendix A, "JCA Syntax Example," describes the JCA output of the Converter using a commented example file.

Appendix B, "Java Card CAP File Debug Component Format," describes the format for the Java Card Debug custom CAP file component.

Related Books

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

- *Java Card™ 2.1.1 Application Programming Interface*, Sun Microsystems, Inc.
- *Java Card™ 2.1.1 Virtual Machine Specification*, Sun Microsystems, Inc.
- *Java Card™ 2.1.1 Runtime Environment (JCRE) Specification*, Sun Microsystems, Inc.
- *Java Card™ 2.1.2 Off-Card Verifier White Paper*, Sun Microsystems, Inc.
- *The Java™ Programming Language (Java Series), Second Edition* by Ken Arnold and James Golsing. Addison-Wesley, 1998, ISBN 0-201-31006-6.
- *The Java™ Virtual Machine Specification (Java Series), Second Edition* by Tim Lindholm and Frank Yellin. Addison-Wesley, 1999, ISBN 0-201-43294-3
- *The Java Class Libraries: An Annotated Reference (Java Series)* by Patrick Chan and Rosanna Lee. Addison-Wesley, ISBN: 0201634589
- *ISO 7816 Specification* Parts 1-6

You can download the Java Card 2.1.1 specifications from Sun's web site: `http://java.sun.com/products/javacard`

What Typographic Changes Mean

The following table describes the typographic changes used in this book.

TABLE P-1 Typographic Conventions

Typeface or Symbol	Meaning	Example

`AaBbCc123`
or
`classes`
	The names of commands, files, items of source code, and directories; on-screen computer output	Edit your `.login` file.
		Use `ls -a` to list all files.
		`machine_name% You have mail.`

<AaBbCc123>	Command-line placeholder: replace with a real name or value	To delete a file, type `rm` <filename>.
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in *User's Guide*. These are called *class* options. You *must* be root to do this.

CHAPTER 1

Introduction to the Java Card Development Kit

The Java Card Development Kit is a suite of tools for designing Java Card technology-based implementations and for developing applets based on the *Java Card™ 2.1.1 Application Programming Interface*.

Note – Even though this release of the Development Kit is version 2.1.2, the Java Card technology specifications are unchanged from version 2.1.1. You can download the Java Card 2.1.1 specifications from Sun's web site: `http://java.sun.com/products/javacard`

FIGURE 1-1 on page 2 shows two main data flows through the Java Card Development Kit 2.1.2 components. One flow is for mask production. The other is for CAP file (converted applet) production and installation. Both flows start with Java source being compiled and input to the Converter. Both flows end with Java Card bytecode running in a JCRE (Java Card Runtime Environment).

Mask production refers to embedding the Java Card virtual machine, runtime environment and applets in the read-only memory of a smart card during manufacture. CAP file production and installation refers to the process of downloading and installing applets to a smart card after manufacture.

Any implementation of a JCRE contains a Java Card Virtual Machine (VM), the Java Card Application Programming Interface (API) classes (and industry-specific extensions), and support services.

In this release, the only JCRE provided is written in the C language (C-JCRE).

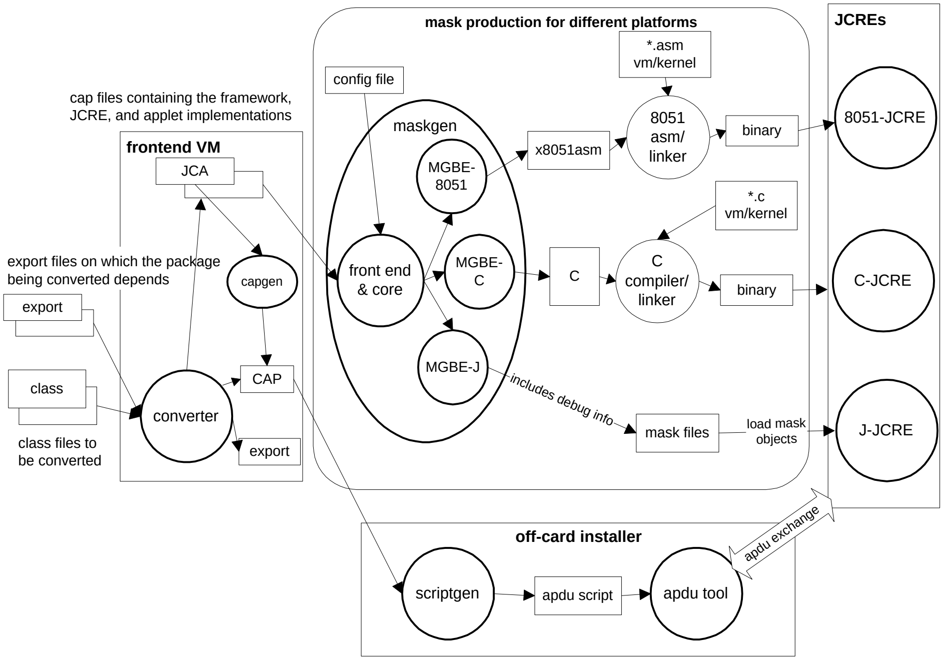


FIGURE 1-1 Java Card 2.1.2 Tool Architecture

Mask Production Flow

The Converter can convert classes that comprise a Java package to a JCA file. When producing a mask, `maskgen` takes a set of JCA files, one for each Java Card package in the mask, and produces output for incorporation into a mask. The mask generator is designed with plugable back-ends. When a new Java Card Runtime Environment (JCRE) is created, a new mask generator back-end (MGBE) is provided that produces output specific to a target. The output is source code appropriate to that target. For example, 8051 assembly language for 8051-based hardware, or C language source code for input to the C-JCRE.

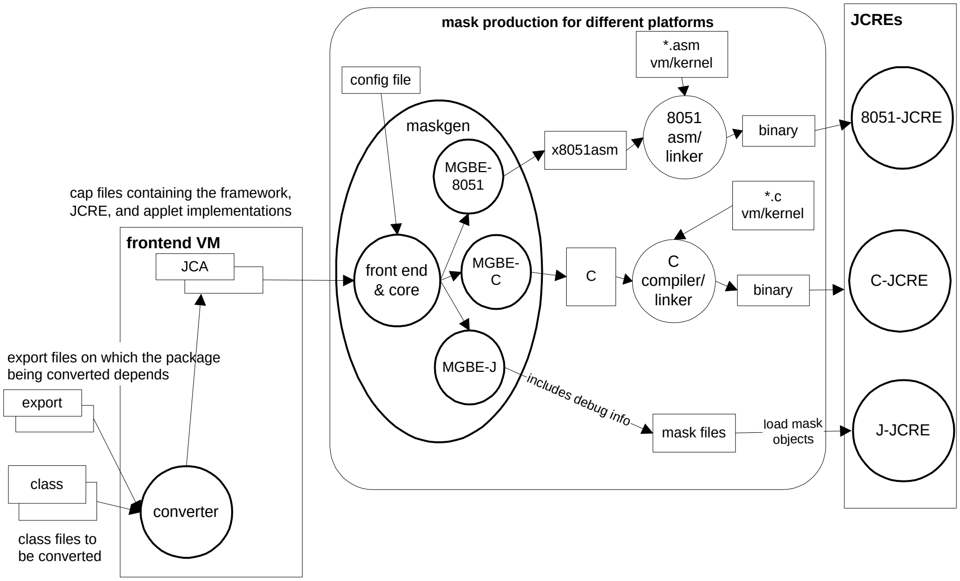
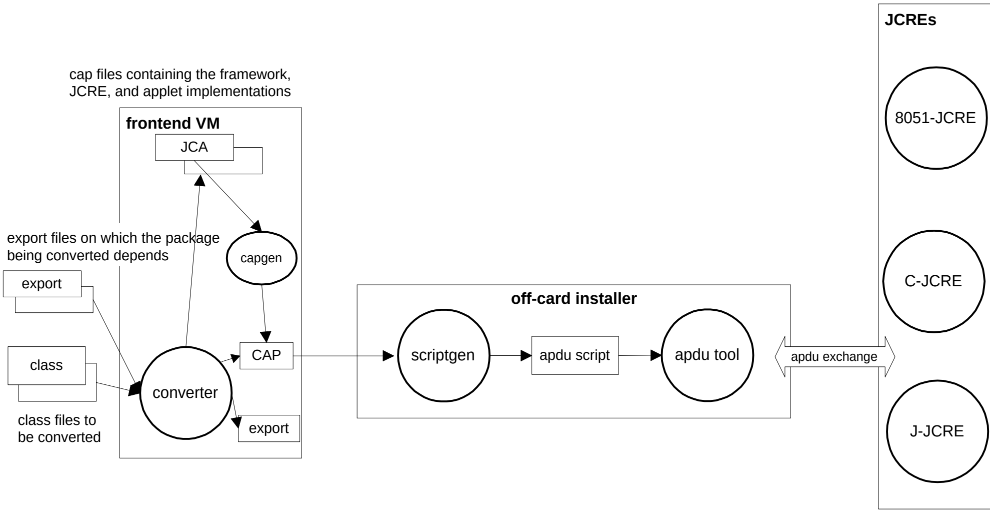


FIGURE 1-2 Java Card 2.1.2 Maskgen Tool Architecture

CAP File Flow

The capgen tool takes a JCA file as input to produce a CAP file. The Converter can also take all the classes in a package and convert them into a CAP file directly. 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 JCRE implementation. For more information on the APDUTool, refer to Chapter 12, “Using the APDUTool.”



JCREs
cap files containing the framework, JCRE, and applet implementations
frontend VM
JCA
8051-JCRE
export files on which the package being converted depends
capgen
C-JCRE
export
off-card installer
CAP
scriptgen
apdu script
apdu tool
apdu exchange
class
converter
class files to be converted
export
J-JCRE

FIGURE 1-3 Java Card 2.1.2 CAP Tool Architecture

CHAPTER **2**

Installation

This release is provided for Solaris (versions 2.6, 2.7 and 2.8) and for the Microsoft Windows NT 4.0 (with Service Pack 4) platforms as compressed Zip archives.

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 Development Kit (JDK) from `http://java.sun.com/j2se/` . Supported JDK versions are 1.2.2 and 1.3 (the latest).

   If you are installing JDK on Solaris, make sure that all the required patches are installed. To get more information, refer to the product documentation available at
   `http://www.sun.com/solaris/java`

   You must set the environment variable `JAVA_HOME` before you run any scripts or batch files. (Refer to "Setting Environment Variables for Solaris" on page 7 or "Setting Environment Variables for Windows" on page 9.)

2. Obtain `javax.comm`

   The Java Communications API 2.0 contains a package `javax.comm`, which is needed to run the Java Card Development Kit. Please visit Sun's web site at `http://java.sun.com/products/javacomm` to obtain the package. Follow the instructions provided in the file `Readme.html` to install the package.

Installing the Java Card Development Kit Binaries

Unzip the file provided with this release. To unpack the file, use the appropriate unzip utility. For Windows, this is Winzip (available from http://www.winzip.com). For Solaris, the appropriate unzip utility is unzip.

Note – There are separate sections below covering installation for the Windows and Solaris platforms. If you are a Windows user, to avoid confusion when reading a Solaris procedure, substitute the \ character for / in paths.

Similarly, for an environment variable such as $JC21_HOME in Solaris (csh), the Windows equivalent reference would be %JC21_HOME%.

Solaris Installation Procedure

The release is contained in a file named java_card_kit-2_1_2-solsparc.zip .

1. Save the file in a convenient installation location of your choice: for example, in the directory /javacard.
2. `% cd /javacard`
3. `% unzip java_card_kit-2_1_2-solsparc.zip`
4. `% ls -F`

   ```
    java_card_kit-2_1_2-solsparc.zip    java_card_kit-2_1_2/
   ```

5. `% cd java_card_kit-2_1_2`
6. `% pwd`

   ```
   /javacard/java_card_kit-2_1_2
   ```

   This directory is now the root of the development kit installation. You should set the environment variable JC21_HOME equal to this directory. (Refer to “Setting Environment Variables for Solaris” on page 7.)

7. `% ls`

   ```
    api21_export_files/  bin/
    doc/                 lib/
    samples/             COPYRIGHT
   ```

```
README                    RELEASENOTES
```

The descriptions of these items are as follows:

api21_export_files	Directory contains the export files for the Java Card 2.1.1 API packages.
bin	Directory contains all shell scripts for running the tools (such as the apdutool, capdump, converter and so forth), and the cref binary executable.
doc	Documentation includes the present document, the *Java Card™ 2.1.2 Development Kit User's Guide*, and the *Java Card™ 2.1.2 Off-Card Verifier White Paper.*
lib	Directory contains all Java jar files required for tools. It also contains the api21.jar file that is necessary to write Java Card applets and libraries.
samples	Directory contains sample applets and demonstration programs.
COPYRIGHT	Contains the copyright notice for the product.
README, RELEASENOTES	Contains important information about this release.

Setting Environment Variables for Solaris

Set the environment variable JC21_HOME to the installation directory. For example (using csh), if you unzipped the release in the directory /javacard:

```
setenv JC21_HOME /javacard/java_card_kit-2_1_2
```

Or, if you unzipped the installation into a different directory, define the environment variable JC21_HOME accordingly.

Next, set the environment variable JAVA_HOME to the directory where you installed your Java development tools. For example,

```
setenv JAVA_HOME /usr/java1.3
```

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

```
setenv PATH .:$JC21_HOME/bin:$PATH
```

We suggest you automate these environment settings:

Create a csh script file (named for example javacard_env.cshrc) which includes the above setenv statements.

Once the script file is created, run it from the command prompt:

```
% source javacard_env.cshrc
```

This is useful in running the Java Card Development Kit tools and in running the samples and demonstrations (refer to Chapter 3, “Java Card Samples and Demonstrations”).

Windows Installation Procedure

The release is contained in a file named java_card_kit-2_1_2-win.zip .

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

2. `C:\> winzip32 java_card_kit-2_1_2-win.zip`

   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.)

3. `C:\> cd java_card_kit-2_1_2`

   This directory is now the root of the development kit installation. You should set the environment variable JC21_HOME equal to this directory. (Refer to “Setting Environment Variables for Windows,” below.)

4. `C:\java_card_kit-2_1_2> dir /w`

   ```
   [api21_export_files] [bin]
   [doc]                [lib]
   [samples]            COPYRIGHT.txt
   README.txt           RELEASENOTES.txt
   ```

The descriptions of these items are as follows:

api21_export_files	Directory contains the export files for the Java Card 2.1.1 API packages.
bin	Directory contains all batch files for running the tools (such as the apdutool, capdump, the converter and so forth), and the cref binary executable.
doc	Documentation includes the present document, the *Java Card™ 2.1.2 Development Kit User’s Guide*, and the *Java Card™ 2.1.2 Off-Card Verifier White Paper.*

`lib`	Directory contains all Java jar files required for tools. It also contains the `api21.jar` file that is necessary to write Java Card applets and libraries.
`samples`	Directory contains sample applets and demonstration programs.
`COPYRIGHT.txt`	Contains the copyright notice for the product.
`README.txt`, `RELEASENOTES.txt`	Contains important information about this release.

Setting Environment Variables for Windows

Set the environment variable `JC21_HOME` to the installation directory. For example, if you unzipped the release in the root directory of the C volume:

```
set JC21_HOME=c:\java_card_kit-2_1_2
```

Or, if you unzipped the installation into a different directory, define the environment variable `JC21_HOME` accordingly.

Next, set the environment variable `JAVA_HOME` to the directory where you installed your Java development tools. For example,

```
set JAVA_HOME=d:\java\jdk13
```

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

```
set PATH=%JC21_HOME%\bin;%PATH%
```

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

```
@echo off
set JC21_HOME=C:\java_card_kit-2_1_2
set JAVA_HOME=d:\java\jdk13
set PATH=.;%JC21_HOME%\bin;%PATH%
```

Sample Programs and Demonstrations

All samples are contained in the `samples` directory.

Solaris:

1. `% cd $JC21_HOME/samples`

2. `% ls`

```
build_samples    classes/
src/
```

Windows:

1. `C:> cd %JC21_HOME%\samples`

2. `C:\java_card_kit-2_1_2\samples> dir /w`

```
build_samples.bat    [classes]
[src]
```

The descriptions of these items are given in the table below.

Note – If you are a Windows user, substitute the `\` character for `/` in the paths.

`classes`	Directory contains pre-built sample java applet classes.
`build_samples` or `build_samples.bat`	A script or batch file to automate building samples only.
`src`	Directory contains the sources for the sample applets that belong to the packages `com.sun.javacard.samples.*`.
`src/demo`	Directory contains the demonstration masks, APDU scripts and expected APDU outputs for the demonstrations.

CHAPTER 3

Java Card Samples and Demonstrations

Five sample applets are provided with this release, which illustrate the use of the Java Card API. They are: `HelloWorld`, `JavaPurse`, `JavaLoyalty`, `NullApp` and `Wallet`. Also included is `SampleLibrary`. The sources and associated class files are also provided.

Included with this release are three demonstration programs. These illustrate very important scenarios of applet masking and post-manufacture installation.

Preliminaries

Prior to using the demonstrations, you must first build the sample programs.

Script File for 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 `$JC21_HOME/samples/build_samples` (for a Solaris installation) or `%JC21_HOME%\samples\build_samples.bat` (for a Windows installation).

Running the Script

The command line syntax for the script is:

```
build_samples [options]
```

The following table describes possible values for *options*:

Value of *options*	Description
[no options]	Builds all targets.
`-help`	Prints out a help message and exits.
`-clean`	Removes all files produced by the build script.

Setting Environment Variables

The `build_samples` script uses the environment variables `JC21_HOME` and `JAVA_HOME`. To correctly set these environment variables, refer to "Setting Environment Variables for Solaris" on page 7 or "Setting Environment Variables for Windows" on page 9.

Building the Sample Applets

Run the script without parameters to build the samples:

```
build_samples
```

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

Preparing

1. A `classes` directory is created as a peer to `src` under the `samples` directory.
2. Class files from `api21.jar` are extracted into this `classes` directory. This is necessary, since the Converter cannot read JAR files.
3. The Java Card 2.1.1 API export files are also copied to the same structure under `classes`.

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:

```
javac -g src/com/sun/javacard/samples/HelloWorld/*.java
```

Converting the Class Files

The next step is to convert the Java class files.

For each class file, a corresponding configuration file is copied to the classes directory. This file, which has an .opt extension, is fed to the converter tool. (The .opt file is necessary because some operating systems don't support many command-line option arguments.)

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 JCA (*.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, "Using the Converter."

The Demonstrations

There are two demonstration masks and three demonstration programs in the demo directory.

The first demonstration (demo1) contains the installer, the JavaPurse, JavaLoyalty, Wallet and SampleLibrary packages as part of the ROM mask image.

The second demonstration (demo2) contains only the installer in the mask image and downloads four CAP files into the simulator (SampleLibrary, JavaPurse, JavaLoyalty, and Wallet).

The third demonstration (demo3) is designed to run after successful completion of demo2, and illustrates the second time power-up of an already initialized mask.

Files in the demo Directory

The files provided are:

jcwde.app	lists all the applets (and their IDs) to be loaded into the simulated mask
*.in files	mask configuration files
*.cfg file	platform-specific configuration file
*.scr files	demonstration apdutool script files
*.expected.out files	files for comparison with apdutool output when the demos are run

Running scriptgen to Generate Scripts for apdutool

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
```

As it creates a script file for each package, the build script concatenates the output to demo2.scr. At the end, the build script adds the APDU script file AppletTest.scr. This APDU script is included in the mask to exercise the other applets, so that you can see each of them invoked when the simulation is run.

Running the Demonstrations

demo1 runs in the JCWDE.

demo2 runs in the C-JCRE simulator. This is because the JCWDE is not able to support downloading of CAP files.

demo3 runs in the C-JCRE simulator, due to the need to restore the virtual machine state after the initial run.

Running demo1

1. The command is:

```
jcwde jcwde.app
```

2. In a separate command window, 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.

Running demo2

1. Run cref using the following command:

```
cref -o demoee
```

2. In a separate command window, 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 10, “Using the C-JCRE.”)

Note – The APDU script for demo2 contains commands for package installation. The file demo2.scr.expected.out included in this release is correct for JDK 1.3. If you are running JDK 1.2.2, expect a different bytecode sequence.

Running demo3

demo3 should be run after demo2.

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 10, “Using the C-JCRE.”)

2. In a separate command window, 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.

CHAPTER 4

Using the JCWDE

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

The JCWDE tool executable consists of the `jcwde.jar`, `api21.jar`, and `apduio.jar` files. The main class for JCWDE is `com.sun.javacard.jcwde.Main`.

A sample batch and shell script are provided to start JCWDE.

Preliminaries

Make sure that the `JC21_HOME` environment variable is set, as detailed in "Setting Environment Variables for Solaris" on page 7 or "Setting Environment Variables for Windows" on page 9. Also, the `CLASSPATH` environment variable needs to be set to reflect the location of the classfiles for the applets to be simulated.

Configuring the Applets in the JCWDE Mask

The applets to be configured in the mask during JCWDE simulation need to be listed in a configuration file that is passed to the JCWDE as a command line argument. 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 0xXX[:0xXX][1]. Note that AID should be 5 to 16 bytes in length.

For example:

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

Note – The installer applet must be listed first in the JCWDE 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 §4.2 of the *Java Card™ 2.1.1 Virtual Machine Specification*, Sun Microsystems, Inc., and in *ISO 7816 Specification* Parts 1-6.

Running the JCWDE Tool

The general format of the command to run the JCWDE is as follows:

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

where:

the flag -p allows you to specify a TCP/IP port other than the default port;

the flag -version prints the JCWDE version number;

the flag -nobanner suppresses all banner messages;

the flag -help prints out a help message; and

<config-file> is the configuration file described above.

When started, JCWDE starts listening to APDUs in T=0 format on the TCP/IP port specified by the –p port parameter. The default port is 9025.

[1] Repeat the construct :0xXX as many times as necessary.

CHAPTER **5**

Using the Converter

The Converter loads and processes class files that make up a Java package. The Converter outputs are a CAP file and an export file.

Another Converter output is a JCA (Java Card Assembly) file, which you then input to `capgen` to produce a CAP file. A JCA file is a human-readable ASCII file to aid testing and debugging. If you are converting code that is intended for the ROM of a smart card, and especially if it contains native methods, you will input the JCA to `maskgen` to produce a mask file. (For more information on using `maskgen`, refer to Chapter 9, “Using maskgen.”)

Java Compiler Options

The `class` files should be compiled with `-g` option of the JDK Java compiler command line. Don’t use `-O` option.

This is because the Converter determines the local variable types by checking the `LocalVariableTable` attribute within the class file. This attribute is generated in the class file only if the `-g` option is used at the Java compiler command line.

The `-O` option is not recommended at the Java compiler command line, for two reasons. This option is intended to optimize execution speed rather than minimize memory usage. The latter is much more important in Java Card technology. Also, if the `-O` option is used, the `LocalVariableTable` attribute won’t be generated even if the `-g` option is used.

File and Directory Naming Conventions

This section details 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 *Java Card™ 2.1.1 Virtual Machine Specification*, Sun Microsystems, Inc.

Input Files

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 Files

The name of the CAP file, export file, and the JCA 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 JCA file in the output directory as an intermediate result. If the JCA file is not a desired output, then omit the `- out JCA`. The Converter then deletes the JCA file at the end of the conversion.

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 23.)

Loading Export Files

A Java Card export file contains the public API linking information (public classes, public and protected methods and fields) 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 5-1 on page 22 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 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`.

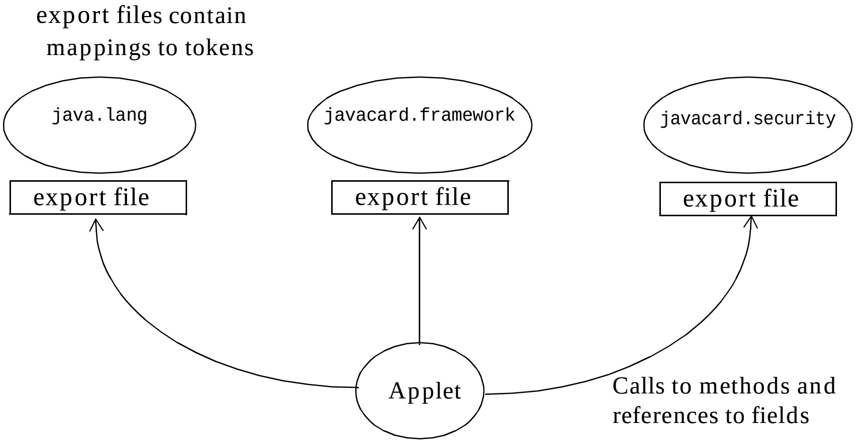


FIGURE 5-1 Calls between packages go through the export files

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 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 *Java Card™ 2.1.1 Virtual Machine Specification*).

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 of the currently-converted package the same way it loads other export files.

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 UNIX® platform, and a batch file (`converter.bat`) on the Microsoft Windows NT platform.

Command Line Arguments

The arguments to this command line are:

`<package_name>`

the 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:

```
-classdir    <the root directory of the class hierarchy>
```

Set 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.

`-i`

Instruct the Converter to support the 32-bit integer type.

`-exportpath   <List of directories>`

These are 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 NT platform and colon (:) for the UNIX® platform.

If this option is not specified, the Converter sets the `exportpath` to the Java `classpath`.

`-exportmap`

Use 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`.

`-applet <AID> <class_name>`

Set 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.

`-d   <the root directory for output>`

Set the root directory for output.

`-out [CAP] [EXP] [JCA]`

Tell the Converter to output the CAP file, and/or the export file, and/or the JCA file.

By default (if this option is not specified), the Converter outputs a CAP file and an export file.

`-V, -version`

Print the Converter version string.

`-v, -verbose`

Enable verbose output.

`-mask`

Indicates this package is for a mask, so restrictions on native methods are relaxed. This option must be specified if a mask is to be generated out of this package using `maskgen`.

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

```
-help
```

Print help message.

```
-nowarn
```

Instruct the Converter not to report warning messages.

```
-nobanner
```

Suppress all banner messages.

```
-debug
```

Generates the debug component described in Appendix B, “Java Card CAP File Debug Component Format.” If the `-mask` option is also specified, the file `debug.msk` will be generated in the output directory.

Command Configuration File

You could also include all the command line arguments and options in a configuration file. 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.

Viewing an Export File

The `exp2text` tool is provided to allow you to view any binary export file in human-readable (ASCII) format.

```
exp2text  [options] <package_name>
```

Where options include:

```
-classdir <input root directory>
```

specify the root directory where the program looks for the export file.

`-d <output root directory>`

specify the root directory for output.

`-help`

Print help message.

CHAPTER **6**

Using the Off-Card Verifier

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 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: VerifyCap, VerifyExp and VerifyRev, which provide functionality for verifying CAP files, export files and the binary compatibility between two versions of a package respectively. The following sections explain the usage of each tool.

VerifyCap

VerifyCap is used to verify a CAP file within the context of the export file(s) it imports, and the export file, if any, that it exports. This verification confirms whether a CAP file is internally consistent, as defined in Chapter 6 of the *Java Card*™ *2.1.1 Virtual Machine Specification*, and consistent with a context in which it may reside in a Java Card enabled device. The context is represented by various export file(s). Each individual export file is also verified as a single unit. The scenario is shown in the figure below. `p2.exp` is optional, since `p2.cap` may not export any of its elements.

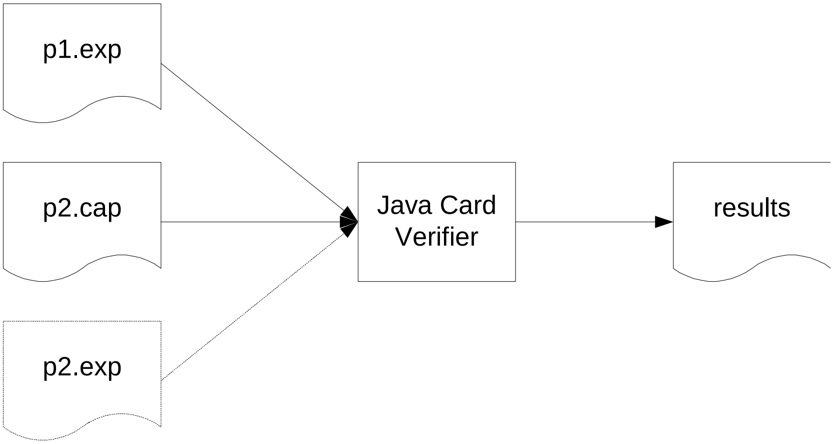


FIGURE 6-1 Verifying a CAP file

Running VerifyCap

Command line usage is:

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

The file to invoke VerifyCap is a shell script (`verifycap`) on the UNIX® platform and a batch file (`verifycap.bat`) on the Microsoft Windows NT platform.

Command Line Arguments

The arguments to this command line are:

```
<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

The options in this command line are:

`-help`

Print help message.

`-nobanner`

Suppress banner message.

`-package <pkg>`

Set the name of the package to be verified.

`-verbose`

Turn on verbose mode.

`-version`

Print version number and exit.

VerifyExp

VerifyExp 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 checks determine whether an export file is internally consistent and viable as defined in Chapter 5 of the *Java Card™ 2.1.1 Virtual Machine Specification*. This scenario is shown in the figure below.

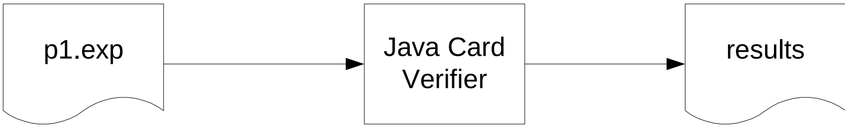


FIGURE 6-2 Verifying an export file

Running VerifyExp

Command line usage is:

```
verifyexp [options] <export_file>
```

The file to invoke VerifyExp is a shell script (`verifyexp`) on the UNIX® platform and a batch file (`verifyexp.bat`) on the Microsoft Windows NT platform.

Command Line Arguments

The arguments to this command line are:

```
<export_file>
```

Fully qualified path and name of the export file

Command Line Options

The options in this command line are:

`-help`

Print help message.

`-nobanner`

Suppress banner message.

`-verbose`

Turn on verbose mode.

`-version`

Print version number and exit.

VerifyRev

VerifyRev checks for binary compatibility between revisions of a package by comparing the respective export files. This verification examines whether the Java Card version rules, including those imposed for binary compatibility as defined in Section 4.4 of the *Java Card™ 2.1.1 Virtual Machine Specification*, have been followed.

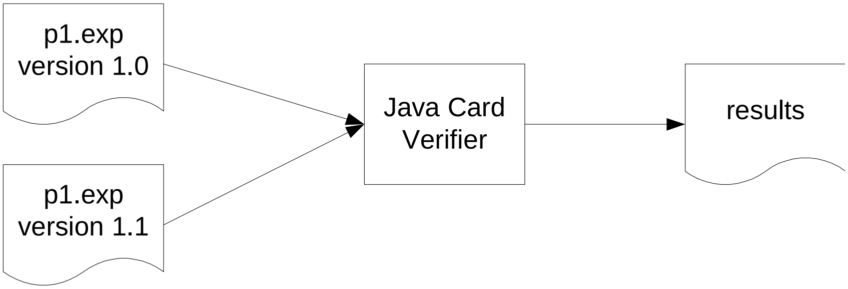


FIGURE 6-3 Verifying binary compatibility of export files

Running VerifyRev

Command line usage is:

```
verifyrev [options] <1st export_file> <2nd export_file>
```

The file to invoke VerifyRev is a shell script (`verifyrev`) on the UNIX® platform and a batch file (`verifyrev.bat`) on the Microsoft Windows NT platform.

Command Line Arguments

The arguments to this command line are:

```
<1st export_file>
<2nd export_file>
```

Where `<1st export_file>` and `<2nd export_file>` are the fully qualified paths of the two different 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

The options in this command line are:

`-help`

Print help message.

`-nobanner`

Suppress banner message.

`-verbose`

Turn on verbose mode.

`-version`

Print version number and exit.

CHAPTER 7

Using capgen

capgen is a backend to the Converter. It produces a CAP file from a JCA file.

Command Line for capgen

The file to invoke capgen is a shell script (capgen) on the UNIX® platform, and a batch file (capgen.bat) on the Microsoft Windows NT platform.

Command line syntax for capgen is:

```
capgen [-options] <infile>
```

where <infile> is the JCA file.

The option values and their actions are:

The flag -o 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.

The flag -version outputs the version information.

The flag -help displays online documentation for this command.

The flag -nobanner suppresses all banner messages.

CHAPTER 8

Using capdump

capdump produces an ASCII representation of a CAP file.

Command Line for capdump

The file to invoke capdump is a shell script (capdump) on the UNIX® platform, and a batch file (capdump.bat) on the Microsoft Windows NT platform.

Command line usage of capdump is:

```
capdump <infile>
```

where <infile> is the CAP file.

Output from this command is always written to standard output.

There are no command line options to capdump.

CHAPTER 9

Using maskgen

maskgen produces a mask file from a set of JCA 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-JCRE, and size, which reports size statistics for the mask. Other generators, which are not supported in this release, include jref, which supports the J-JCRE and a51, which supports the Keil A51 assembly language interpreter.

Command Line for maskgen

The file to invoke maskgen is a shell script (maskgen) on the UNIX® platform, and a batch file (maskgen.bat) on the Microsoft Windows NT platform. Usage is:

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

Arguments

<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-JCRE interpreter.

jref - output for the J-JCRE interpreter (not supported for this release).

size - outputs mask size statistics.

`<infile> [ <infile> ...]`

Any number of `.jca` files can be input to `maskgen` as a whitespace-separated list. On the Microsoft Windows platform, there is a limit of nine positional arguments to a command. If you need more than nine arguments on a command line, create a text file containing a list of `.jca` file names, and prepend an "@" character to the name of this text file as an argument to `maskgen`.

Options

`-c <config file>`

The -c flag specifies a configuration file,[1] which allows you to tailor the format of the output file produced by the specified generator (for example, assembly language or C code). A configuration file for the cref generator is provided in the `demo` directory.

`-debuginfo`

This option allows you to generate debug information for the generated mask.

`-o <outfile>`

This option allows you to specify 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.

`-help`

Displays online documentation for `maskgen`.

`-nobanner`

Suppresses all banner messages.

[1] The configuration file contains target-specific information. For example, the following line maps a native Java Card method to a native label:

```
javacard/framework/JCSystem/beginTransaction()V=beginTransaction_NM
```

Example

An example of containing command line arguments in a text file is:

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

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

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

CHAPTER 10

Using the C-JCRE

The *C language Java Card Runtime Environment* (C-JCRE) is the Java Card reference implementation. It is a simulator that can be built with a ROM mask, much like a real Java Card 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-JCRE. The C-JCRE performs I/O via a socket interface, using the TLP-224 protocol, simulating a Java Card in a card reader (CAD).

Highlights of Changes

1. Not dependent on third party APIs.

The implementation has eliminated the use of OCAPI™. Note that there is no equivalent to OCAPI. There is no new code that directly replaces OCAPI.

Benefits include:

- size reduction
- performance improvement due to fewer layers
- easier to debug

2. Implementation is very portable.

The C-JCRE is written in C, which is a very portable language. For Win32 users, it can be built using MinGW.

The code has been restructured to allow building for multiple targets, including Win32, Solaris, and 8051. A shell script (for Solaris) and a batch file (for Win32) are provided. There is greater use of macros, which facilitates porting to platforms with different memory addressing schemes (such as support for XRAM), different native stack sizes, or different endianness. The intent of these changes is to make it easier to port the implementation.

3. Optimized stream interface to EEPROM for writes.

This version implements a stream interface for writing to EEPROM.

This interface allows the low-level (hardware-dependent) layer to use RAM to buffer writes until a full EEPROM page is completed, before writing it to EEPROM. This avoids additional page writes in some cases.

4. Transactions

An “optimistic” transaction scheme is used. The original data is logged to a transaction buffer. New data is written in-place. In case of an abort or failure, the transaction buffer is used to reliably restore the original data. The optimized (non-atomic) EEPROM stream interface is used, with resulting performance improvement.

5. I/O

The existing socket-based TLP-224 I/O scheme is supported for Win32 and UNIX versions.

6. API

It implements the 2.1.1 Java Card specifications.

7. Improved runtime tracing

The runtime stack trace has been made more readable. There are additional trace options.

8. Simpler store file model.

This version does not create or use a default store file.

Running the C-JCRE

Installer Mask

The C-JCRE is supplied as a prebuilt executable, `cref.exe` for Windows, and `cref` for Solaris. The Java Card Development Kit Installer, the Java Card Virtual Machine interpreter and Java Card framework are built into the mask. It can be used as-is to load and run applets. Other than the Installer, it does not contain any applets.

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

Running the C-JCRE

Command line usage of C-JCRE is the same on Win32 and Solaris. 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.

`-b`

Dump a Byte Code Histogram at the end of the execution.

`-e`

Display the program counter and the stack when an exception occurs.

`-h, -help`

Print out a help screen.

`-i <input filename>`

Use the named file to initialize EEPROM.

`-n`

Do a trace display of the native methods that are invoked.

`-nobanner`

Do not print program banner.

`-nomeminfo`

Do not print memory statistics at the start of execution.

`-o <output filename>`

Save the EEPROM contents to the named file.

`-s`

Silent mode. Do not create any output unless followed by other flag options.

`-t`

Do a line-by-line trace display of the mask's execution.

`-version`

Print only the program's version number. Do not execute.

`-z`

Print the resource consumption statistics at termination.

I/O

The C-JCRE performs I/O via a socket interface, using the TLP-224 protocol, simulating a Java Card in a card reader (CAD). Use `apdutool` to read script files and send APDUs via a socket to the C-JCRE. See "apdutool Syntax" on page 72 for details. Note that you can have the C-JCRE running on one workstation and run `apdutool` on another workstation.

Store Files

The user can save the state of EEPROM contents, so the state can be loaded in a later invocation of the C-JCRE. This is made possible by specifying a *store file* to save the EEPROM contents. The `-i` and `-o` option flags are used to manipulate store files at the `cref` command line.

The `-i` flag allows the user to specify the initial store file from which to initialize the EEPROM portion of the virtual machine before JCVM bytecode execution commences.

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

The commit of EEPROM memory changes during the execution of the C-JCRE is not affected by the `-o` flag. There is no conflict between the use of either or both of these flags with the other option flags. Neither standard nor error output is written to the output file named with the `-o` option.

Note – The previous (2.1.1) version of the C-JCRE would create or input a store file with the default name of `store`. That file was created if it did not exist. No output file was created. This default behavior was sometimes misleading. The current version does not use or create a default store file.

Use of the `-i` and `-o` option flags permits a variety of useful execution scenarios. For example:

Input Store File

```
C:\>cref -i e2save
```

The C-JCRE attempts to initialize simulated EEPROM from the store file named `e2save`. No output file will be created.

Output Store File

```
C:\>cref -o e2save
```

The C-JCRE writes EEPROM data to the file `e2save`. The file will be created if it does not currently exist. Any existing store file named `e2save` is overwritten.

Same Input and Output Store File

```
C:\>cref -i e2save -o e2save
```

The C-JCRE attempts to initialize simulated EEPROM from the store 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 in that the contents of EEPROM is persistent.

Different Input and Output Store Files

```
C:\>cref -i e2save_in -o e2save_out
```

The C-JCRE attempts to initialize simulated EEPROM from the store file named e2save_in, and during C-JCRE processing, writes EEPROM updates to a store file named e2save_out. The output file will be created if it does not exist. Using different names for input and output store files eliminates much potential confusion. This command line can be executed multiple times with the same results.

Note – Be careful naming your store files. The C-JCRE will overwrite an existing file specified as an output store 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

The C-JCRE executable supplied in this release (cref for Solaris and cref.exe for Windows) contains the demo2.c mask (renamed as mask.c). demo2.c is an example mask that contains only the Installer applet. See Chapter 3, “Java Card Samples and Demonstrations” for details about demo2.

Internal Operation of the C-JCRE

C-JCRE is targeted towards Java Card Virtual Machine developers.

Java Card VM Developers

VM developers are usually concerned with the development of the infrastructure of the Java Card technology, and running mask code within this infrastructure. They need to implement the following:

- bytecode routines
- bytecode support routines
- runtime data structures
- i/o routines
- native method support routines
- applet and package tables
- token tables
- firewall support

Each of these components is implemented in the C-JCRE.

Java Card VM Byte Code Routines

The Java Card VM byte codes described in the *Java Card™ 2.1.1 Virtual Machine Specification*, Sun Microsystems, Inc. are all represented by a single 8 bit value. The byte code fetched by the interpreter is used to index into an array of function pointers. For the reference implementation, there are 185 valid opcodes, numbering from 0 through 184 consecutively. The `impdep1` and `impdep2` opcodes, numbered 254 and 255 respectively, are not implemented. Byte code values above 184 cause an internal error to occur within the C-JCRE.

Byte code support routines

Where appropriate, several byte code support routines are used to facilitate common processing requirements shared by some of the byte code routines. The trade off is a minor speed degradation in exchange for major space savings. The routines are platform independent, and are not intended for use as an API by developers modifying the Reference Implementation source code. Excluded from this grouping are any and all platform-specific and all Hardware Abstraction Layer support routines. While some of these routines serve as support routines to the byte code routine implementations, they do constitute part of the internal API, and require platform-dependent implementations.

Run Time Data Structures

The internal run time data structures are influenced by both the CAP file design and the JCVM byte code definitions that are specified in the *Java Card™ 2.1.1 Virtual Machine Specification*, Sun Microsystems, Inc.. The need to conserve both EEPROM and RAM space requires a comprehensive and concise set of structures.

EEPROM Write Optimization Using Streams

The C-JCRE implements a stream interface for EEPROM writes. This is a useful technique for dealing with the problem of EEPROM being organized into pages. The following is an example of how E2 streams are used:

```
/* Initialize the Array’s Object Header fields */
  /* Global = 0; JCRE_EntryPointObj = 0; TransientMode = 0;
              TransientObj = 0; */
  /* Don’t mind atomicity: this block is not yet accessible */
  E2P_stream_open(arrayref);
  E2P_stream_write_u8(Header_ArrayType_Reference);
  E2P_stream_write_u8(cc);
  E2P_stream_write_u16(Object_PA_CLASS_OFFSET);
  E2P_stream_write_u16(count);

  /* Initialize the Array’s Object Header fields */
  E2P_stream_write_u16(eltclass);
  /* init contents to zeros */
  E2P_stream_fill_array(count * SIZE_REF, 0);
  E2P_stream_flush();
```

Note the advantages of the stream interface:

- It avoids constructing a struct in RAM and copying it.
- It avoids calculating address offsets to construct a struct.
- It permits optimization of the EE writes.

C-JCRE Execution Model

The execution model of the C-JCRE interpreter is shown in FIGURE 10-1. The interpreter supports the following functionality:

- emulation of a processor stack
- support of EEPROM heap for post-issuance applets
- support of a RAM heap
- processing of smart card APDUs

Shown as input at the right edge of the diagram is the bytecode stream to be executed.

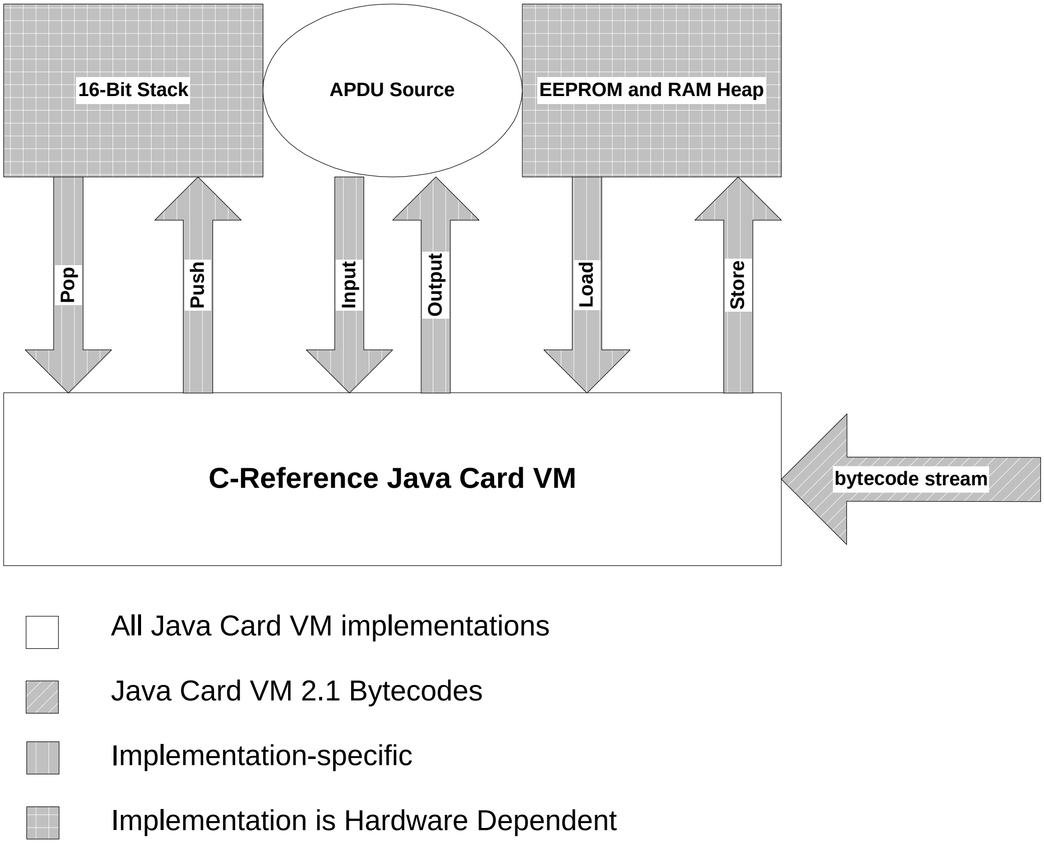


FIGURE 10-1 C-JCRE Execution Model

C-JCRE Internal Structure

The internal structure of the C-JCRE interpreter is shown in FIGURE 10-2.

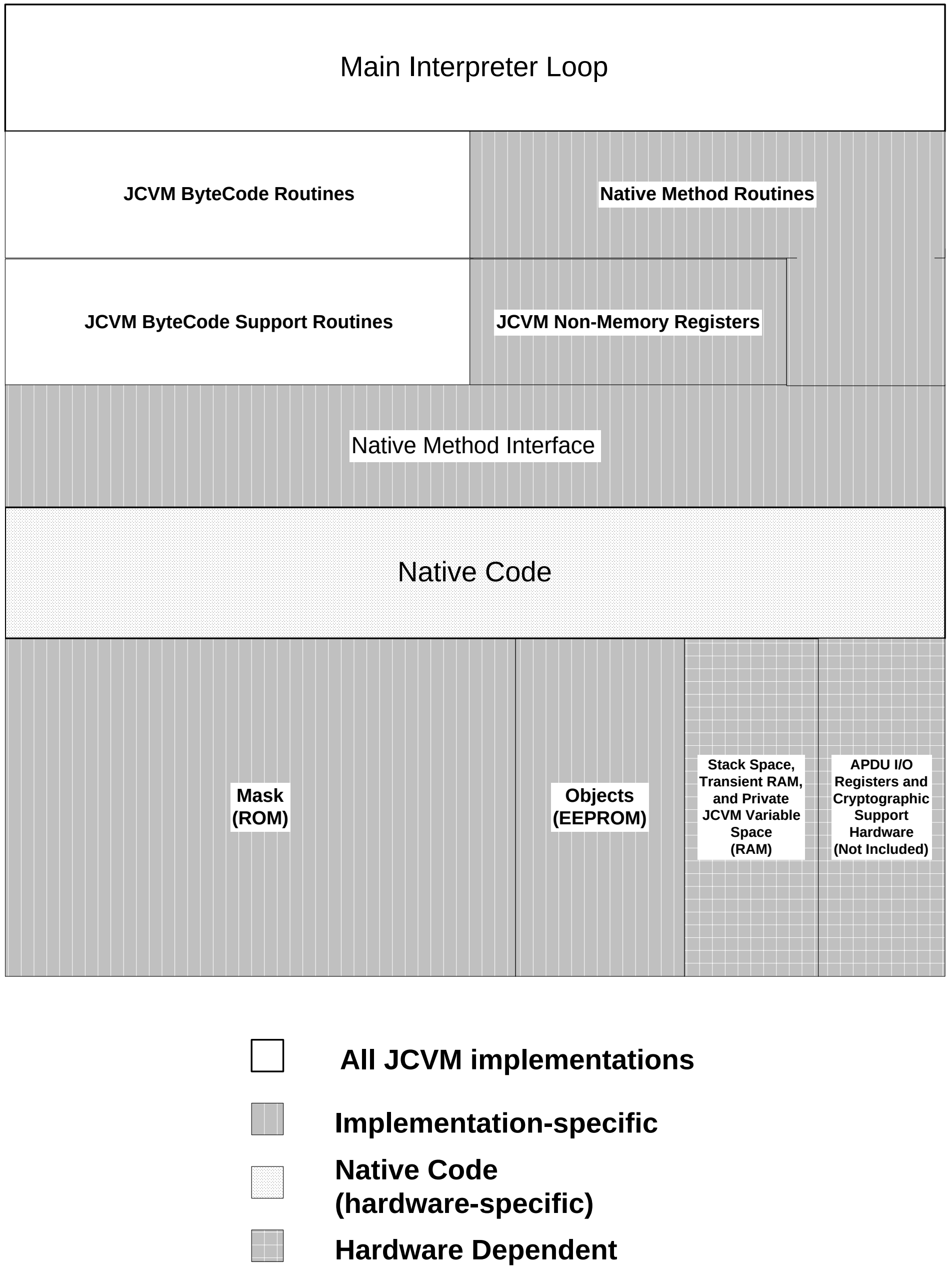

Main Interpreter Loop
JCVM ByteCode Routines
Native Method Routines
JCVM ByteCode Support Routines
JCVM Non-Memory Registers
Native Method Interface
Native Code
Mask (ROM)
Objects (EEPROM)
Stack Space, Transient RAM, and Private JCVM Variable Space (RAM)
APDU I/O Registers and Cryptographic Support Hardware (Not Included)
All JCVM implementations
Implementation-specific
Native Code (hardware-specific)
Hardware Dependent

FIGURE 10-2 C-JCRE Functional Block Diagram

The Interpreter's Native Platform Components

Hardware Dependent Components

The hardware dependent components include the physical memories (or their simulated representations), the native method API routines that support access to the memories and to the I/O registers (simulated or not). The C-JCRE includes hardware-dependent routines that perform the basic memory and I/O access routines.

Stack and Heap Memories

The default Stack and Heap memory areas within the Reference Implementation are defined as static byte arrays.

Transient Objects

Transient objects are defined in the *Java Card™ 2.1.1 Runtime Environment (JCRE) Specification*. Space for the fields of a transient object, once allocated, is reserved for the lifetime of the transient object with which they are associated.

OCAPI™ API Layer

The OCAPI™ layer, which was used in the previous release of the C-JCRE, is no longer used.

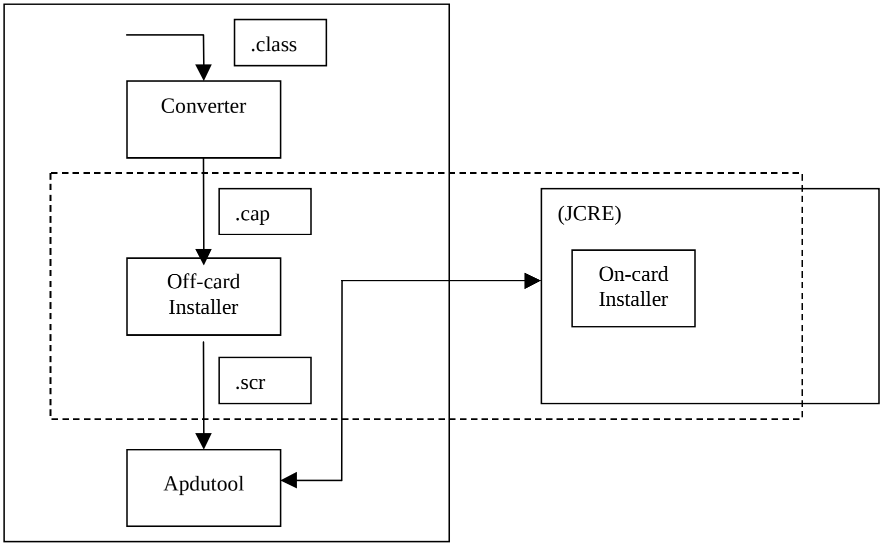
CHAPTER 11

Using the Installer

Overview

The Java Card installer's role is to dynamically download a Java Card package declared in a CAP file to a Java Card-enabled smart card, and to perform necessary on-card linking. During development, the CAP file can be installed in the C-JCRE rather than on a Java Card-enabled smart card.

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



The data flow of the installation process is as follows:

1. An off-card installer takes a CAP file, produced by the 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 downloaded to the on-card installer in the JCRE.

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 *Java Card™ 2.1.1 Runtime Environment (JCRE) Specification*, Sun Microsystems, Inc.

How to Use Scriptgen

Scriptgen is a tool to convert a CAP file into a script file that 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 *Java Card™ 2.1.1 Virtual Machine Specification*, Sun Microsystems, Inc.

Scriptgen Usage:

`scriptgen [options] <capFilePath>`

where `options` include:

`-help`

Print help message and exit.

`-o <filename>`

Output filename (default is `stdout`).

`-version`

Print version number and exit.

`-nobanner`

Do not print version number.

`-nobeginend`

Do not output "CAP Begin" and "CAP End" APDU commands.

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

Installer Applet AID

The on-card installer applet AID field value is: 0xa0,0,0,0,0x62,3,1,8,1

How to Use the Installer

The installer is invoked using the APDUtool. (See Chapter 12, “Using the APDUTool.”)

There are three CAP file installation scenarios supported by the installer:

- Download Only
- Create Only
- Download and Create

These three scenarios are described in the next three sections.

Scenario 1: Download Only

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.) Steps to perform for this kind of installation process are:

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.

Scenario 2: Create Only

In this scenario, the applet from a previously downloaded CAP file or an applet compiled in the mask is created. Steps to perform this creation of the JavaPurse applet are:

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.

Scenario 3: Download and Create

In this scenario, an applet is downloaded in a CAP file, and an applet defined in that CAP file is created. Perform these steps to install, download and create the JavaPurse applet:

1. Determine the applet AID.
2. Convert the JavaPurse CAP file to an APDU script using `scriptgen`‘s `-nobeginend` command option, so that the “CAP Begin” and “CAP End” APDU commands will not be part of the script.
3. Prepend the following to the APDU script:

```
powerup;
// Select the installer applet
0xA4 0x04 0x00 0x09 0xa0 0x00 0x00 0x00 0x62 0x03 0x01 0x08 0x01 0x7F;
// CAP Begin
0x80 0xB0 0x00 0x00 0x00 0x7F;
```

4. Append these APDUs at the end of the script:

```
// CAP End
0x80 0xBA 0x00 0x00 0x00 0x7F;
// create JavaPurse
0x80 0xB8 0x00 0x00 0x0b 0x09 0xa0 0x00 0x00 0x00 0x62 0x03 0x01 0x04
0x01 0x00
0x7F;

powerdown;
```

5. Invoke APDUTool with this APDU script path as the argument.

Note – To install more than one applet contained in the same CAP file, repeat the Create Only steps above for each additional applet.

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.

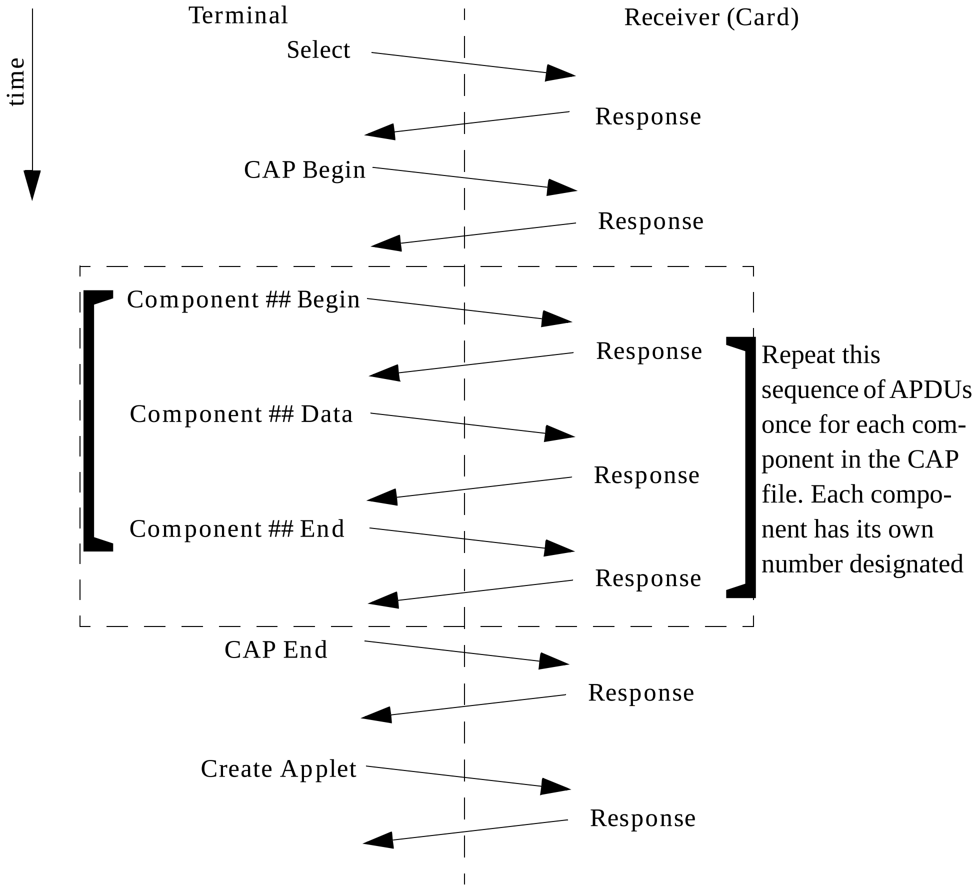


FIGURE 11-1 Installer APDU Transmission Sequence

The following frame (data unit) formats are used in the Installer APDU protocol.

Protocol Data Unit 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 (ACK or NAK)
- 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.

Select

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

Table 1: Select APDU

00, 0xa4, 04, 00	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 values for an ACK frame SW1SW2 are 6XXX, and the values for a NAK frame SW1SW2 are 9000.

Table 2: Response APDU

[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 3: CAP Begin APDU

0x80, 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 4: CAP End APDU

0x80, 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 5: COMPONENT ## Begin APDU

0x80, 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 6: COMPONENT ## End APDU

0x80, 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 7: COMPONENT ## Data APDU

0x80, 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 from each of the already sequentially transmitted components of the CAP file.

Table 8: Create Applet APDU

0x80, 0xb8, 0x00, 0x00	Lc field	AID length field	AID	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 9: Abort APDU

0x80, 0xbe, 0x00, 0x00	Lc field	[optional data]	Le field

Installer Error Response APDUs

```
/**
 * Response status : Invalid CAP file magic number = 0x6402
 */
static final short ERROR_CAP_MAGIC = 0x6402;

/**
 * Response status : Invalid CAP file minor number = 0x6403
 */
static final short ERROR_CAP_MINOR = 0x6403;

/**
 * Response status : Invalid CAP file major number = 0x6404
 */
static final short ERROR_CAP_MAJOR = 0x6404;

/**
 * Response status : Integer not supported = 0x640b
 */
static final short ERROR_INTEGER_UNSUPPORTED = 0x640b;

/**
 * Response status : Duplicate package AID found = 0x640c
 */
static final short ERROR_DUP_PKG_AID = 0x640c;

/**
 * Response status : Duplicate Applet AID found = 0x640d
 */
static final short ERROR_DUP_APPLET_AID = 0x640d;

/**
 * Response status : Installation aborted = 0x640f
 */
static final short ERROR_ABORTED = 0x640f;

/**
 * Response status : Installer in error state = 0x6421
 */
static final short ERROR_STATE = 0x6421;

/**
 * Response status : CAP file component out of order = 0x6422
 */
static final short ERROR_COMP_ORDER = 0x6422;
```

```
    /**
     * Response status : Exception occurred = 0x6424
     */
    static final short ERROR_EXCEPTION = 0x6424;

    /**
     * Response status : Install APDU command out of order = 0x6425
     */
    static final short ERROR_COMMAND_ORDER = 0x6425;

    /**
     * Response status : Invalid component tag number = 0x6428
     */
    static final short ERROR_COMP_TAG = 0x6428;

    /**
     * Response status : Invalid install instruction = 0x6436
     */
    static final short ERROR_INSTRUCTION = 0x6436;

    /**
     * Response status : Import package not found = 0x6438
     */
    static final short ERROR_IMPORT_NOT_FOUND = 0x6438;

    /**
     * Response status : Illegal package identifier = 0x6439
     */
    static final short ERROR_PKG_ID = 0x6439;

    /**
     * Response status : Maximum allowable package methods exceeded
= 0x6442
     */
    static final short ERROR_PKG_METHOD_MAX_EXCEEDED = 0x6442;

    /**
     * Response status : Applet not found = 0x6443
     */
    static final short ERROR_APPLET_NOT_FOUND = 0x6443;

    /**
     * Response status : Applet creation failed = 0x6444
     */
    static final short ERROR_APPLET_CREATION = 0x6444;

    /**
    * Response status : Maximum allowable instances exceeded  = 0x6445
```

```
     */
    static final short ERROR_INSTANCE_MAX_EXCEEDED = 0x6445;

    /**
     * Response status : Memory allocation failed = 0x6446
     */
    static final short ERROR_ALLOCATE_FAILURE = 0x6446;

    /**
     * Response status : Import class not found = 0x6447
     */
    static final short ERROR_IMPORT_CLASS_NOT_FOUND = 0x6447;
```

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;
```

Installer Requirements

The on-card installer applet must be the first applet in the JCRE.

Installer Limitations

- The maximum length of the parameter in applet creation APDU command is 14.
- The maximum number of packages to be downloaded is 16 minus the number of ROM packages.
- The maximum number of applets to be downloaded is 16 minus the number of ROM applets.
- The maximum length of data in the installer APDU commands is 32.
- 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.

CHAPTER 12

Using the APDUTool

The APDUTool reads a script file containing APDUs and sends them to the C-JCRE (or other JCRE) or the JCWDE. Each APDU is processed by the JCRE or JCWDE and returned to the APDUTool, which displays both the command and response APDUs on the console. Optionally, the APDUTool can write this information to a log file.

Command Line for apdutool

The file to invoke apdutool is a shell script (apdutool) on the UNIX® platform, and a batch file (apdutool.bat) on the Microsoft Windows NT platform.

The command line usage for apdutool is:

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

The option values and their actions are:

The flag -h allows you to specify the host name on which the TCP/IP socket port is found. (See the flag -p.)

The flag -noatr suppresses outputting an ATR (answer to reset).

The flag -nobanner suppresses all banner messages.

The flag -o allows you to specify an output file. If an output file is not specified with the -o flag, output defaults to standard output.

The flag -p allows you to specify a TCP/IP socket port other than the default port (which is 9025).

The flag -serialPort allows you to specify input from a serial COM port rather than a TCP/IP socket port.

The flag `-version` outputs the version information.

The `<inputFile>` argument allows you to specify the input script (or scripts).

The flag `-help` displays online documentation for this command.

apdutool Syntax

The following is a command line invocation sample:

```
apdutool example.scr
```

This command runs the APDUTool with the file `example.scr` as input. Output goes to the console.

```
apdutool –o example.scr.out example.scr
```

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

The 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 ';'. Comments may 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:

powerUp;

Send a power up command to the reader. A powerUp command must be executed prior to sending any C-APDUs to the reader.

powerDown;

Send a power down command to the reader.

echo "string";

Echo the quoted string to the output file. The leading and trailing quote characters are removed.

delay <Integer>;

Pause execution of the script for the number of milliseconds specified by `<Integer>`.

APPENDIX **A**

JCA Syntax Example

This appendix contains an annotated JCA file output from the Converter. The comments in this file are intended to aid the developer in understanding the syntax of the JCA language, and as a guide for debugging Converter output.

Note – To get an html file with the BNF grammar for the JCA syntax, use the Java jjdoc tool with the file $JC21SRC_HOME/src/share/java/tools/converter/com/sun/javacard/jcasm/Parser.jj> .

```
/*
 * JCA (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 JCA directives and commands.
 *
 * A JCA file is textual representation of the contents of a CAP file. The contents
 * of a JCA 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
 *          methods
 *              method directives
 *              method statements
 *
 * JCA 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 JCA
 * 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.
       .InterfaceInfoTable 0.0 {
          0;                         // index in public method table of method
          1;                         // index in public method table of method
       }
       .InterfaceInfoTable 0.0 {
          1;                         // index in public method table of method
       }

       /*
        * 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;
l0:       nop;
l1:       nop;
l2:       nop;
l3:         nop;
l4:         nop;
l5:         nop;
            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 JCA 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;
            nop;
            nop;
L2:         goto L1;
            nop;
            nop;
            goto_w L1;
            nop;
            nop;
            goto_w L3;
            nop;
            nop;
            nop;
L3:         return;
        }

        /*
```

```
 * This method illustrates the use of each Java Card 2.1.1 instruction.
 * 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 Java Card
      * 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**

Java Card CAP File Debug Component Format

This appendix specifies the format for the Java Card Debug custom CAP file component. The Debug component contains all the metadata necessary for debugging a package on a suitably instrumented Java Card Virtual Machine. It is not required for executing Java Card software in a non-debug environment.

Identifying a Debug Component

The Debug component format conforms to the constraints specified in Section 6.1.2 of *Java Card™ 2.1.1 Virtual Machine Specification*. The details for identifying a Debug component are as follows:

Component Name	`Debug.cap`
Component AID	`0xA0:00:00:00:62:03:05:01:01`
Component Tag	`0xDB`

The debug_component Structure

This is the top-level structure of the debug component. It starts with some basic information about the component itself and the package. There is a list of all the strings used in the component. These strings are referenced by index from items in the class, field, and method info structures. That information is followed by a list of `class_debug_info` records that contains all the debug data for the package's classes, fields, and methods.

```
debug_component {
    u1 tag
    u2 size
    u1 component_AID_length
    u1 component_AID[component_AID_length]
    u1 minor_version
    u1 major_version
    u2 string_count
    utf8_info strings[string_count]
    u2 package_name_index
    u2 class_count
    class_debug_info classes[class_count]
}
```

The items in the `debug_component` structure are:

Value	Description
`tag`	The one-byte tag required to identify custom components. The value must be `0xDB`.
`size`	The number of bytes in the component, excluding the `tag` and `size` items.
`component_AID_length`	The number of bytes in the `component_AID` item.
`component_AID`	The AID of this custom component. It must match the AID in the directory entry for this component.
`major_version`, `minor_version`	The major and minor version number of this custom component. The format described by this document is 1.0.
`string_count`	The number of strings in the `strings` table.
`strings`	A table of all the strings used in this component. The various `<name>_index` items that occur through this component are all unsigned two-byte indices into this table. The table is zero-based.
`package_name_index`	Contains the index into the `strings` table. The `strings` table at that entry must be the fully-qualified name of the package in this CAP file.
`class_count`	The number of classes in the `classes` table.
`classes`	Contains `class_debug_info` structures for all the classes in this package.

The utf8_info Structure

The component contains a table of the strings used to name the contents of the package. Each string is represented as a utf8_info datatype. The various items in the component such as name_index and type_index are indices into this table and refer to the string at that index. Indices are zero-based, unsigned, two-byte values.

```
utf8_info {
    u2 length
    u1 bytes[length]
}
```

The items in the utf8_info structure are:

Value	Description
length	The number of bytes in the string.
bytes	The bytes of the string.

The class_debug_info Structure

The class_debug_info structure contains all of the debugging information for a class or interface. It also contains tables of debugging information for all its classes' fields and methods.

```
class_debug_info {
    u2 name_index
    u2 access_flags
    u2 location
    u2 superclass_name_index
    u2 source_file_index
    u1 interface_count
    u2 field_count
    u2 method_count
    u2 interface_names_indexes[interface_count]
    field_debug_info fields[field_count]
    method_debug_info methods[method_count]
}
```

The items in the class_debug_info structure are:

Value	Description
name_index	Contains the index into the the fully-qualified name of this class.
access_flags	A two-byte mask of modifiers that apply to this class. The modifiers are: Name Value ACC_PUBLIC 0x0001 ACC_FINAL 0x0010 ACC_INTERFACE 0x0200 ACC_ABSTRACT 0x0400 ACC_SHAREABLE 0x0800
location	The zero-based byte offset of the class_info record for this class in the Class component.
superclass_name_index	Contains the index to the fully-qualified name of the superclass of this class.
source_file_index	Contains the index to the name of the source file in which this class is defined.
interface_count	The number of indexes in the interface_names_indexes table.
field_count	The number of field_debug_info structures in the fields table.
method_count	The number of method_debug_info structures in the methods table.
interface_names_indexes	Contains the indexes to the names of all the interfaces directly implemented by this class.
fields	Contains field_debug_info structures for all the fields in this class.
methods	Contains method_debug_info structures for all the methods in this class.

The field_debug_info Structure

The field_debug_info structure describes a field in a class. It can describe either an instance field, a static field, or a constant (primitive final static) field. The contents union will have the form of a token_var if the field is an instance field, a location_var if it is a static field, or a const_value if it is a constant.

```
field_debug_info {
    u2 name_index
    u2 descriptor_index
    u2 access_flags
    union {
        {
            u1 pad1
            u1 pad2
            u1 pad3
            u1 token
        } token_var
        {
            u2 pad
            u2 location
        } location_var
        u4 const_value
    } contents
}
```

The items in the `field_debug_info` structure are:

Value	Description
`name_index`	Contains the index to the short name of the field (ex: "`applets`").
`descriptor_index`	Contains an index to the type of the field. Class types are fully-qualified (ex: "`[Ljavacard/framework/Applet;`").

`access_flags`	A two-byte mask of modifiers that apply to this field.
	
	`Name` `Value`
	`ACC_PUBLIC` `0x0001`
	`ACC_PRIVATE` `0x0002`
	`ACC_PROTECTED` `0x0004`
	`ACC_STATIC` `0x0008`
	`ACC_FINAL` `0x0010`

`contents`	A `field_debug_info` structure can describe an instance field, a static field, or a static final field (a constant). Constants can be either primitive data or arrays of primitive data. Depending on the kind of field described, the `contents` item is interpreted in different ways. The kind and type of the field can be determined by examining the field's descriptor and access flags.

Value	Description
token_var	If the field is an instance field, this value is the instance field token of the field. The pad1, pad2, and pad3 items are padding only; their values should be ignored.
location_var	If the field is a non-final static field or a final static field with an array type (a constant array), this value is the zero-based byte offset of the location for this field in the static field image. The pad item is padding only; its value should be ignored.
const_value	If the field is a final static field of type byte, boolean, short, or int, this value is interpreted as a signed 32-bit constant.

The method_debug_info Structure

The method_debug_info structure describes a method of a class. It can describe methods that are either virtual or non-virtual (static or initialization methods).

```
method_debug_info {
    u2 name_index
    u2 descriptor_index
    u2 access_flags
    u2 location
    u1 header_size
    u2 body_size
    u2 variable_count
    u2 line_count
    variable_info variable_table[variable_count]
    line_info line_table[line_count]
}
```

The items in the method_debug_info structure are:

Value	Description
name_index	Contains the index to the short name of the method (ex: "lookupAID").
descriptor_index	Contains an index to the argument and return types of the method (essentially the signature without the method name). Class types are fully-qualified (ex: "([BSB)Ljavacard/framework/AID;")
access_flags	A two-byte mask of modifiers that apply to this method. Name Value ACC_PUBLIC 0x0001 ACC_PRIVATE 0x0002 ACC_PROTECTED 0x0004 ACC_STATIC 0x0008 ACC_FINAL 0x0010 ACC_NATIVE 0x0100 ACC_ABSTRACT 0x0400 Note: The ACC_NATIVE flag is only valid for methods located in a card mask. It cannot be used for methods contained in a CAP file.
location	The zero-based byte offset of the location for this method in the method component of the CAP file. Abstract methods have a location of zero.
header_size	The size in bytes of the header of the method. Abstract methods have a header_size of zero.
body_size	The size in bytes of the body of the method, not including the method header. Abstract methods have a body_size of zero.
variable_count	The number of variable_info entries in the variable_table item.
line_count	The number of line_info entries in the line_table item.
variable_table	Contains the variable_info structures for all variables in this method.
line_table	Contains the line_info structures that map bytecode instructions of this method to lines in the class's source file.

The variable_info Structure

The variable_info structure describes a single local variable of a method. It indicates the index into the local variables of the current frame at which the local variable can be found, as well as the name and type of the variable. It also indicates the range of bytecodes within which the variable has a value.

```
variable_info {
    u1 index
    u2 name_index
    u2 descriptor_index
    u2 start_pc
    u2 length
}
```

The items in the variable_info structure are:

Value	Description
index	The index of the variable in the local stack frame, as used in load and store bytecodes. If the variable at index is of type int, it occupies both index and index + 1.
name_index	Contains the index to the short name of the local variable (ex: "applets").
descriptor_index	Contains the index to the type of the local variable. Class types are fully-qualified (ex: "[Ljavacard/framework/Applet;").
start_pc	First bytecode in which the variable is in-scope and valid.
length	Number of bytecodes in which the variable is in-scope and valid. The value of start_pc + length will be either the index of the next bytecode after the valid range, or the first index beyond the end of the bytecode array.

The line_info Structure

Each line_info item represents a mapping of a range of bytecode instructions to a particular line in the source file that contains the method. The range of instructions is from start_pc to end_pc, inclusive. The source_line is the one-based line number in the source file.

```
line_info {
    u2 start_pc
    u2 end_pc
    u2 source_line
}
```

The items in the `line_info` structure are:

Value	Description
`start_pc`	The first bytecode in the range, zero-based count within the method.
`end_pc`	The last bytecode in the range, zero-based count within the method.
`source_line`	Line number in the source file. One-based count within the files.