

SmartCard Framework Developer's Guide

SDK/J Authentication Package Version:1.0



Copyright © 2008-2010 Ricoh Co., Ltd.

Terms of Use and Trademarks

1. The contents of this book may be changed without notice in the future.
2. The copying, reproducing, changing, quoting, reprinting, or distributing a part/all of this book are prohibited.
3. We make no warranty, express or implied, regarding this document and the sample codes described in this document. We will not be held responsible for any of our customer's losses, damages resulting from lost profits, or claims from any third party on using this document and the sample codes described in this document.
4. Trademarks

Ethernet® is a registered trademark of Xerox Corporation.

PostScript® and Acrobat® are registered trademarks of Adobe Systems Incorporated in the United States and/or other countries.

Microsoft® and Windows® are registered trademarks of Microsoft Corporation in the United States and other countries.

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

Red Hat is a registered trademark of Red Hat, Inc. in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States and other countries, or both.

Java, JVM(CVM) and CDC are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Eclipse is a trademark of eclipse.org in the United States, other countries, or both.

OSGi(TM) is a trademark, registered trademark, or service mark of The Open Services Gateway Initiative in the US and other countries.

Apache is a registered trademark of The Apache Software Foundation in the United States, other countries, or both.

Other product names used herein are for identification purposes only and might be trademarks of their respective companies. We disclaim any and all rights in those marks.

Contents

1. Introduction	2
1.1. Target Readers.....	2
1.2. Software Requirements	2
1.3. Restrictions	2
2. Architecture Outline	3
2.1. CardManager (CardManager class)	3
2.2. CardService (concrete CardService class)	4
2.3. CardServiceRegister (CardServiceRegistry class)	4
2.4. CardServiceBundle (RegisterBundle class)	4
3. Application Development	5
3.1. Writing an application	5
3.2. Creating a CardService	7
3.3. Registering a CardService.....	9
4. Application Installation	11
4.1. DALP file settings	11
4.2. Installing the application	11
5. Appendix	12
5.1. How to check the operation of a SmartCard Framework application by using an emulator	12
Change History	14

1. Introduction

This document describes how to use the SmartCard Framework v1.

The SmartCard Framework v1 (hereinafter referred to as “the SCF”) is a software framework that allows a Device SDK Type-J (hereinafter referred to as “SDK/J”) application to use a SmartCard card.

1.1. Target Readers

This document is intended for the application developers who are experienced in SDK/J application development and are familiar with the basics of SmartCard.

1.2. Software Requirements

Use of the SCF requires an operation environment where:

- PC/SC daemon is enabled.

* About the setting of PC/SC daemon, see “SDK/J Authentication Package Settings Guide”.

1.3. Restrictions

Restrictions on using the SCF:

- The operation will not be guaranteed when the SCF and the OpenCard Framework are used together on a same execution environment at the same time.
- The operation will not be guaranteed when more than one card reader device or other USB device are connected to the USB host at the same time.

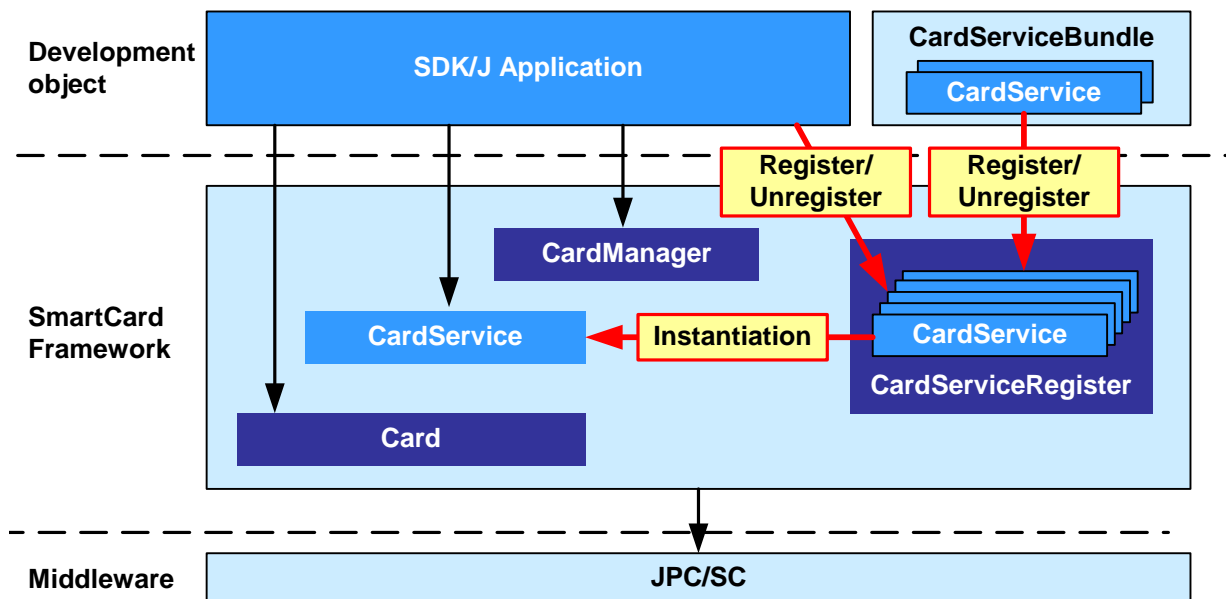
2. Architecture Outline

The outline drawing below shows how SCF components and their related software components are deployed.

In the SCF, application logic and card-specific logic can be implemented separately.

This aspect of the SCF allows you to hide the detailed specification of a card from application developers, and helps you to improve the interoperability of an application.

This section explains SCF components and related Java classes.

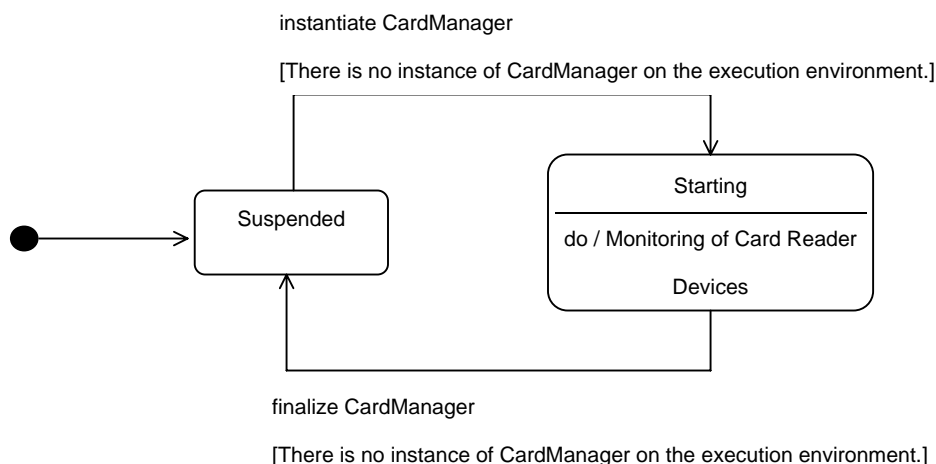


2.1. CardManager (CardManager class)

The CardManager serves as an interface between the SCF and applications.

The SCF creates a daemon thread when the first CardManager is instantiated, and destroys it when the last CardManager is finalized.

The daemon thread will monitor the states of card reader devices being connected to the target device.



2.2. CardService (concrete CardService class)

The CardService abstracts the specification of a card.

Usually this is implemented by a card vender or a card issuer who knows the detailed specification of a card.

A CardService will become available after it is registered with the CardServiceRegister.

Registration/Deletion of a CardService is usually implemented by the application or the CardServiceBundle.

2.3. CardServiceRegister (CardServiceRegistry class)

The CardServiceRegister holds the available CardServices.

At any time, only one instance of the CardServiceRegistry class is present on the execution environment.

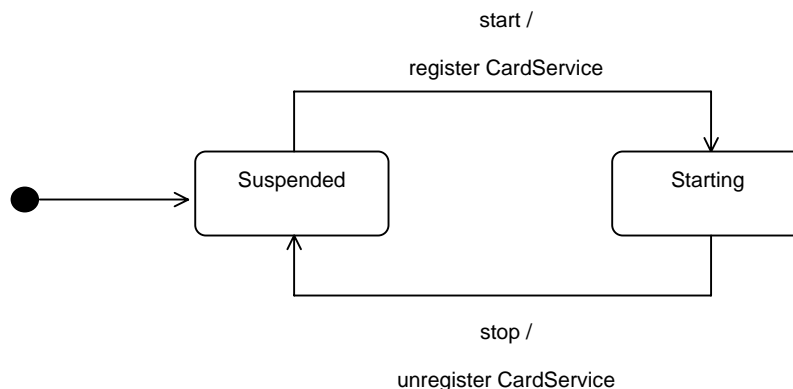
On receiving a request for a CardService, the SCF will search for an appropriate CardService in the CardServiceRegister, and will return an instance of it.

2.4. CardServiceBundle (RegisterBundle class)

The CardServiceBundle serves as a container of CardServices and can register/unregister CardServices.

The developer of a CardService offers an appropriate CardServiceBundle together with the CardService.

Each CardService in a CardServiceBundle will be available only when the CardServiceBundle is active.



3. Application Development

This section explains how to write a SmartCard application using SCF and shows some concise sample codes.

Use **scard.jar** to develop a SCF application.

Details on SDK/J application development and exception handling are not described here.

3.1. Writing an application

3.1.1. Controlling a SmartCard card

Before you can control a SmartCard, you must obtain an instance of the Card class.

One possible way to obtain an instance of the Card class is using the `waitForCard` method of the `CardManager` class. The `waitForCard` method will wait until a SmartCard is inserted into the card reader device or a time-out occurs.

waitForCard Sample Code

```
.
.
/** Instantiate the CardManager. */
CardManager cm = new CardManager();

/** Call the waitForCard method. */
Card card = cm.waitForCard(CardManager.INFINITY);

.
.
```

Another possible way to obtain an instance of the Card class is using the `CardEvent` class.

Create an implementation of the `CardEventListener` in which handling of SmartCard events is described.

Then register it with an instance of the `CardManager` class.

An instance of the Card class can be obtained by the `getCard` method of the `Slot` object obtained by the `getSlot` method of the `CardEvent` class.

CardEvent Sample Code

```
.
.
/** Instantiate the CardManager. */
CardManager cm = new CardManager();

Card card = null;
/** Register the Listener with the CardManager. */
cm.addListener(new CardEventListener() {

    public void inserted (CardEvent event) {
        /** Implement the handling of card insertion. */
        card = event.getSlot().getCard();
    }

    public void removed (CardEvent event) {
        /** Implement the handling of card removal. */
        card = null;
    }

});
.
```

After an instance of the Card class is obtained, access to the SmartCard card will become possible by using the methods of the Card class. The following sample code shows how to use the obtained Card instance.

For details, see the Javadoc.

Card Instance Sample Code

```

        .
        .
        /* Obtain the information of the card by using a Card.Info object. */
        Slot slot    = card.getInfo().getSlot();           // The slot is obtained.
        int protocol = card.getInfo().getProtocol();       // The protocol is obtained.
        byte[] atr    = card.getInfo().getAtr();           // The ATR is obtained.

        /* Access to the card by using a Card.IO object. */
        byte[] response = card.getIO().transmit(
            new byte[] { (byte)0x80, (byte)0x20, (byte)0x00, (byte)0x00 }); // A command is
        sent to the card.
        .
        .

```

3.1.2. Using a CardService

By using a CardService for implementation of the card access feature, you can develop an application more easily without caring about the detailed specification of a card. (Use of a CardService service requires registration of the CardService with the CardServiceRegister. The method to do this is described later in this document.)

A CardService can be obtained by the getCardService method of the Card class. Specify the interface of the CardService service to the parameter of the getCardService method.

FileAccessService Sample Code

```

        .
        .
        /** Obtain the FileAccessService from the Card instance. */
        FileAccessService service =
            (FileAccessService) card.getCardService(FileAccessService.class);

        /** Use the services offered by the FileAccessService. */
        .
        .

```

AppletService Sample Code

```

        .
        .
        /** Obtain the AppletService from the Card instance. */
        AppletService service =
            (AppletService) card.getCardService(AppletService.class);

        /** Use the services offered by the AppletService. */
        .
        .

```


3.2. Creating a CardService

This section explains how to implement a CardService.

3.2.1. Creating a CardService

To create a CardService, go through the following steps.

STEP-1 Defining the interface of the CardService

Create the interface that defines the services offered by the CardService.

Note: This step can be skipped when you use the card service interface prepared by the SCF.

```
public interface AuthenticationService {

    public byte[] getUsername() throws CardServiceException;
    public byte[] getPassword() throws CardServiceException;

}
```

STEP-2 Creating the class that inherits the CardService class

All CardServices must inherit the CardService class of SCF.

```
/** Inherit the CardService class. */
public MyAuthenticationService extends CardService {

}
```

STEP-2.1 Overriding the initialize method

The initialize method is called only once after the CardService is instantiated by the SCF.

Specify the Card object to be associated with this Card Service to the parameter of this method.

When the Card object is not supported by the CardService, the UnsupportedCardException exception must be thrown.

```
public MyAuthenticationService extends CardService {

    public synchronized void initialize(Card card)
    throws UnsupportedCardException, CardServiceException {
        super.initialize(card);

        /**
         * If the card is unsupported, throw the UnsupportedCardException exception.
         */
    }

}
```

STEP-2.2 Implementing the interface of the CardService

Implement the interface that defines the services offered by this CardService. (See STEP-1.)

The interface defined here is used as the key when the user uses this CardService.

```
/** Implement the interface that defines the services. */
public MyAuthenticationService extends CardService implements AuthenticationService
{

    public synchronized void initialize(Card card)
    throws UnsupportedOperationException, CardServiceException {
        .
        .
        .
    }

    public byte[] getUsername() throws CardServiceException {
        /**
         * Implement the getUsername method.
         */
    }

    public byte[] getPassword() throws CardServiceException {
        /**
         * Implement the getPassword method.
         */
    }

}
```

3.3. Registering a CardService

This section describes how to register/unregister a CardService.

3.3.1. Registering a CardService by means of the CardServiceBundle

When the CardService is used by more than one application, create a CardServiceBundle and register/unregister the CardService by using the created CardServiceBundle.

Create a new class that inherits the RegisterBundle class and override the getCardServices method.

The newly created class here will be the BundleActivator of the CardServiceBundle.

The getCardServices method should return the Class of the CardService to register with the CardServiceRegister. The CardServiceBundle will register the CardService returned by the getCardServices method with the CardServiceRegister when it is activated, and will remove the CardService from the CardServiceRegister when it is stopped.

```
public class MyRegister extends RegisterBundle {
    /**
     * Override the getCardServices method and return the CardService service.
     */
    protected Class[] getCardServices() throws Exception {
        return new Class[] {
            MyAuthenticationService.class
        };
    }
}
```

The jar file of the CardServiceBundle is comprised of the newly created class, CardServices, and CardService interfaces.

When you use this CardServiceBundle, install only the CardService interface at the time of installation of the application. They must be installed by using an option-jar tag. (See section 4.)

3.3.2. Registering a CardService from the application

Use this method only when the CardService is used by a single application.

The CardService service must be registered with the CardServiceRegister before it is used by the application.

```
.
.
/** Register the CardService service with the CardServiceRegister. */
new CardServiceRegister().add(MyAuthenticationService.class);
.
.
```

The CardService service must be removed from the CardServiceRegister when it is no longer used by the application.

```
.  
.  
/** Remove the CardService service from the CardServiceRegister. */  
new CardServiceRegister().remove(MyAuthenticationService.class);  
.  
.
```

In this case, when you register a CardService from an application, install both the concrete CardService and its CardService interface at the time of installation of the application.

4. Application Installation

This chapter describes how to install an SDK/J application that employs the SCF.

4.1. DALP file settings

About DALP format, see the SDK/J Developer's Guide.

Since the SCF jar file (scard.jar) is deployed in SDK/J SD card, jar file description of the scard.jar is unnecessary.

The following is a configuration example of the DALP file of an application that uses the SCF.

```

    :
    :
    <resources>
      <dsdk version = "2.0"/>
      <jar href="./123456789.jar" basepath="current" main="true"/>
      <option-jar href="./cardservice.jar" basepath="current" main="false"/> *
      <encode-file>123456789</encode-file>
    </resources>
    :
    :

```

* Use option-jar tag for installation of the CardService interfaces if they are provided by a CardServiceBundle.

4.2. Installing the application

The installation procedure is the same as a normal SDK/J application.

For details, see the User's Guide of the SDK/J.

Note: When the option-jar tag is used in the DALP file, it is needed to reboot the target device after the installation.

5. Appendix

5.1. How to check the operation of a SmartCard Framework application by using an emulator

<< Note >>

This chapter is described assuming that the Embedded Software Architecture Emulator 4.13d is used for the operation check. Therefore, if you use other emulators, the operation check may not be performed properly.

Go through the following steps and you can check the operation of an SCF application by using the Embedded Software Architecture Emulator 4.13d.

STEP-1 SCF Establishing the necessary operation environment

First, establish the necessary operation environment for the operation check.

1. Installing a Card Reader Driver

Install a card reader driver if there is no such driver available.

Be sure to install a card reader driver supporting PC/SC.

2. Locating the jpcsc.dll to “cdc-dsdk4” directory of the emulator

Locate the jpcsc.dll, which is the JNI necessary to use PC/SC in Java, to <emulator_installpath>\cdc-dsdk4 directory. The jpcsc.dll can be downloaded from MUSCLE project's web site.

MUSCLE project : <http://www.linuxnet.com/>

3. Adding the jpcsc.jar and scard.jar to the classpath of the emulator

Add the jpcsc.jar, which is the API necessary to use PC/SC in Java, and the jar file of the SCF to the classpath of the emulator. The jpcsc.jar can be downloaded from MUSCLE project's web site. Edit -classpath option of the activation batch file of the emulator as follows.

```

:
:

SET MYCLASSPATH=-classpath
.\resource\sdk\emulator\common\emulatorCommon.jar;.\mnt\sd2\sdk\common\jars\dsdk\dsd
kCommon.jar;.\lib\jh.jar;.\lib\kxml2-2.3.0.jar;.\mnt\sd2\sdk\common\framework.jar;.\
mnt\sd2\sdk\common\jars\smartcard\scard.jar;.\jpcsc.jar

:
:
```

STEP-2 Adding the jar file that is installed by the option-jar to the classpath of the emulator

Add the jar file that is specified by the option-jar to the classpath of the emulator in the same way as in STEP-1.3.

Note: The option-jar tag of the DALP file is not supported by the emulator.

STEP-3 Adding the jar file of the CardServiceBundle to the classpath of the emulator

ServerType applications are not supported by the emulator.

When CardService services are installed by the CardServiceBundle, add the jar file to the classpath of the emulator in the same way as in STEP-1.3.

STEP-4 Registering CardService services

When the application does not dynamically register CardService services, or the application uses the CardServiceBundle for registration of CardService services, they need to be registered from the SmartCard.properties file*.

Describe the CardService services the application uses in the SmartCard.properties file and locate the file under <emulator_installpath>\cdc-dsdk4\cdc-toolkit\CDCTK10\lib directory.

STEP-5 Installing the applicaiton

Start the emulator by using <emulator_installpath>\startemulator-jvm.bat.

Install the application in the same way that a normal SDK/J application is installed.

* Registration of CardService service from the SmartCard.properties file

The SCF checks if the SmartCard.properties file is present under the [java.home]/lib/ directory, when the CardManager class is first loaded. If the SmartCard.properties file is present on the [java.home]/lib/ directory, the CardServices that are specified to the property "jp.co.ricoh.dsdk.scard.service" will be registered with the CardServiceRegistry.

The following is an example of describing the SmartCard.properties file.

To separate CardServices, use a semicolon.

The SmartCard.properties file must be placed under the [java.home]/lib/ directory.

```
jp.co.ricoh.dsdk.scard.service=my.package.CardServiceA;my.package.CardServiceB
```

Change History

Ver. 1.0	SDK/J v4 or later version based on SDK/J AP v1.0 option package for SDK/J v2. Update software requirements Implementation-Version:1.0-1.0
Ver. 1.0.4	Update “5.1. How to check the operation of a SmartCard Framework application by using an emulator” Implementation-Version:1.0-1.0