Oracle® Java ME Embedded

Application Management System API Guide

Release 3.3

E35109-02

April 2013

The Oracle Java ME Embedded Application Management System API Guide describes the APIs for the Application Management System (AMS) of the Oracle Java ME Embedded software. The AMS APIs contain low-level management functionalities, including application and library installation and storage, certificate maintenance, and task management.

Oracle Java ME Embedded Application Management System API Guide, Release 3.3

E35109-02

Contents

6 Tasks

7 The Certificate Info Manager

8 The Locale Change Notifier

Index

List of Tables

Preface

This document describes the Application Management System (AMS) APIs of the Oracle Java ME Embedded. The AMS APIs contain lower-level management functionality, including application and library installation and storage, certificate maintenance, and task management. You can use these APIs to create a new AMS UI front-end for the Oracle Java ME Embedded, which can be substituted for the legacy AMS UI.

Audience

This document focuses on providing information and guidelines for ISV engineers who want to create their own user interface for the AMS. Together with this document, ISV Java ME engineers should have access to the Oracle Java ME Embedded SDK, a compatible IDE, and a Win32 Runtime Environment.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc`.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` or visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

Related Documentation

For a complete list of documents included with the Oracle Java ME Embedded software, see the Release Notes.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.

Convention	Meaning
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1 AMS Introduction

This chapter introduces the "headless" Application Management System (AMS) APIs that are used to interface with the Oracle Java ME Embedded device. The AMS APIs contain management functionality, including application and library installation and storage, certificate maintenance, and task management. The AMS is typically accessed through a command-line interface, or via a remote protocol and a remote tool with a user interface.

Connecting to the Headless AMS CLI

With the Java ME Embedded distribution running, start a terminal emulator (such as PuTTY) and create a raw socket connection to the device address shown on the board. The default ports that are used are shown in Table 1–1.

WARNING: The command-line interface (CLI) feature in this Oracle Java ME Embedded software release is provided only as a concept for your reference. It uses insecure connections with no encryption, authentication, or authorization.

Table 1–1 Ports Used by the Embedded Board

Port	Description
65000	Logging / Java VM System Output
65002	Command-line interface

Once you have a successful connection to the AMS CLI, you can use it to install Java embedded programs with the AMS commands shown in Table 1–2. Note that the AMS syntax may change in future releases; entering `help [command]` is the best way to obtain the latest CLI syntax.

Table 1–2 AMS CLI Commands

Syntax	Description
`ams-list [INDEX or NAME|VENDOR]`	List all installed IMlet suites and their statuses or show the detail of a single suite
`ams-install <URL> [username:password]`	Install an IMlet using the specified JAR or JAD file, specified as a URL. An optional username and password can be supplied for login information as well.

Table 1–2 (Cont.) AMS CLI Commands

Syntax	Description
`ams-update <INDEX or NAME|VENDOR>`	Update the installed IMlet
`ams-remove <INDEX or NAME|VENDOR>`	Remove an installed IMlet
`ams-run <INDEX or NAME|VENDOR> [IMLET_ID] [-debug]`	Execute the specified IMlet or the default if none is specified. An optional debug parameter can be specified to run the IMlet in debug mode.
`ams-stop <INDEX or NAME|VENDOR> [IMLET_ID]`	Stop the specified IMlet or the default if none is specified
`ams-suspend <INDEX or NAME|VENDOR> [IMLET_ID]`	Suspend (pause) the specified IMlet or the default if none is specified
`ams-resume <INDEX or NAME|VENDOR> [IMLET_ID]`	Resume the specified IMlet or the default if none is specified
`ams-setup <INDEX or NAME|VENDOR>`	Display the setup menu of the IMlet
`ams-info <INDEX or NAME|VENDOR>`	Show information about the installed IMlet

`ams-log <command> [param1, param2, ..., paramN]`
`ams-log wdog`
	Display the IMlet log or watchdog log if recorded by the watchdog handler in the platform

`ams-logger-list [INDEX or NAME|VENDOR]`	Retrieve the logger list for the IMlet or all the tasks if one is not specified
`ams-logger-info <INDEX or NAME|VENDOR> [LOGGER_NAME]`	Retrieve logger info for the specified IMlet and logger or all the loggers if is one is not specified
`ams-logger-level-set <INDEX or NAME|VENDOR> [LOGGER_NAME] <LOGGER_LEVEL>`	Set the logger level for specified IMlet or all loggers if one is not specified
`help [command name]`	List the available commands or detailed usages for a single command
`sysmenu <on PASSWORD|off>`	Enable hidden system menu commands. Currently, the password is `12345`.
`exit`	Terminates the current session.

When the `sysmenu` command is entered with the on option, additional system menu commands are available with the AMS CLI, as shown in Table 1–3.

Table 1–3 Additional System Commands Available in the AMS CLI

Syntax	Description
`setprop <KEY> <VALUE>`	Sets a property identified by `<KEY>` with the value `<VALUE>`
`getprop <KEY>`	Returns a property identified by `<KEY>`
`odd [on|off]`	Explicitly sets the on-device debugging (ODD) property to on or off. If no parameters are passed, returns the current ODD value.
`shutdown [-r]`	Perform either a shutdown of the board, or a reboot if the `-r` parameter has been passed. Note that the `watchdog` property should be set to `true` for some platforms (see appropriate reference documentation) to successfully reboot the board.

Installed IMlets can be in one of three states: stopped, suspended, or running. Here is a typical example of using the AMS to install, list, run, and remove an application.

```
oracle>> ams-install file:///helloworld.jad
<<ams-install,start install,file:///helloworld.jad
<<ams-install,install status: stage 0, 5%
<<ams-install,install status: stage 3, 100%
<<ams-install,install status: stage 4, 100%
<<ams-install,OK,Install success

oracle>> ams-list
<<ams-list,0.helloworld|Oracle,STOPPED
<<ams-list,1.netdemo|Oracle,STOPPED
<<ams-list,2.rs232demo|Oracle,RUNNING
<<ams-list,OK,3 suites are installed

oracle>> ams-remove 0
<<ams-remove,OK,helloworld removed

oracle>> ams-list
<<ams-list,1.netdemo|Oracle,STOPPED
<<ams-list,2.rs232demo|Oracle,RUNNING
<<ams-list,OK,2 suites are installed

oracle>> ams-run 1
<<ams-run,OK,started

oracle>> ams-list
<<ams-list,1.netdemo|Oracle,RUNNING
<<ams-list,2.rs232demo|Oracle,RUNNING
<<ams-list,OK,2 suites are installed
```

AmsFactory Class

Certain Java ME Embedded applications may interface with the low-level API classes for the AMS. The entry point for all AMS functionality is provided by the `AmsFactory` class. The `AmsFactory` must not be instantiated. Instead, the `AmsFactory` class provides seven static methods which return objects that provide AMS operations, including application and library installation and storage, certificate maintenance, and task management.

The seven static methods are:

- `static AMSRequestManager getAMSRequestManager() throws SecurityException, UnsupportedServiceException`

  This static method returns the `AMSRequestManager`, which is responsible for handling special user requests, such as pressing a "Home" button or switching to another running application. This method can throw a `SecurityException` if the user does not have the required permission (for example, if the calling MIDlet is untrusted or unsigned as an AMS UI MIDlet), or an `UnsupportedServiceException` if the service is not implemented.

- `static AppInstaller getAppInstaller()` throws SecurityException

  This static method returns the `AppInstaller`, which is responsible for downloading and installing applications and subordinate libraries. This method can throw a `SecurityException` if the user does not have the required permissions.

- `static CertificateInfoManager getCertificateInfoManager()` throws SecurityException

  This static method returns the `CertificateInfoManager`, which manages the certificates installed on the system. Certificates are used to authenticate apps that are downloaded to the system. This method can throw a `SecurityException` if the user does not have the required permissions.

- `static LinkInstaller getLinkInstaller()` throws SecurityException, UnsupportedServiceException

  This static method returns the `LinkInstaller`, which is responsible for downloading and installing links to applications. This method can throw a `SecurityException` if the user does not have the required permissions, or an `UnsupportedServiceException` if the service is not supported.

- `static LocaleChangeNotifier getLocaleChangeNotifier()` throws SecurityException

  This static method returns a `LocaleChangeNotifier`, which alerts applications when a change in locale occurs. This method can throw a `SecurityException` if the user does not have the required permissions.

- `static SuiteStoreManager getStoreManager()` throws SecurityException

  This static method returns the `SuiteStoreManager`, which is responsible for providing a data store for installed suites. This object does not provide access to any purchasing mechanism, but instead assists in storing and managing applications on the local device. This method can throw a `SecurityException` if the user does not have the required permissions.

- `static TaskManager getTaskManager()` throws SecurityException, UnsupportedServiceException

  This static method returns the `TaskManager`, which is responsible for organizing, identifying, and starting applications. This method can throw a `SecurityException` if the user does not have the required permissions, or an `UnsupportedServiceException` if the service is not supported.

The seven static methods of the `AmsFactory` class provide a complete view of the AMS itself. The AMS consists of two installers:

- App Installer
- Link Installer

In this context, an *app* is synonymous with an application: a MIDlet that is running in the Oracle Java ME Embedded. A *link* is a reference to an application or library that is yet to be downloaded. Apps, libraries, and links are the three unique entities that are installed and managed in the AMS.

In addition, two managers assist with installing and monitoring application suites:

- Suite Store Manager
- Task Manager

The AMS also contains the AMS Request Manager, which notifies the UI of requests such as pressing a "Home" button or switching between applications. In addition, the Certificate Info Manager is responsible for managing certificates. Finally, the AMS contains a notification mechanism, the Locale Change Notifier, that alerts interested listeners if the current locale changes.

The `AmsFactory` is the only class in the AMS APIs. The rest of the objects that are presented to the programmer are implementations of interfaces defined throughout the remainder of this document. The use of encapsulated classes is a common software design pattern that decouples the AMS implementation from the means of accessing it, allowing further development and improvement of the AMS without the risk of breaking the API.

2

Application and Library Suites

This chapter introduces the basic data interfaces used throughout the AMS APIs.

The SuiteInfo Interface

All apps, libraries, and links maintain a basic set of identification and state information that acts as a *descriptor*. This descriptor is represented by an implementation of the `SuiteInfo` interface.

Suites can be one of four types, as shown in Table 2–1:

Table 2–1 AMS Suite Types

Suite Type	Description
`ST_APPLICATION`	The suite contains one or more MIDlets with an entry point that can be executed by the AMS.
`ST_LIBRARY`	The suite is a library that can be used by one or more applications.
`ST_LINK`	The suite is a link, which references another application that has yet to be downloaded.
`ST_INVALID`	The suite is invalid and cannot be found or executed.

In addition, suites contain five binary flags that describe their state, as shown in Table 2–2:

Table 2–2 AMS Suite States

State	Description
`STATE_AVAILABLE`	The suite is available for use.
`STATE_ENABLED`	The suite is enabled. When a suite is disabled, any attempt to run application or use a library from this suite should fail.
`STATE_HIDDEN`	The suite is hidden, and should not be visible to the user.
`STATE_REMOVE_DENIED`	The suite should not be removed.
`STATE_UPDATE_DENIED`	The suite should not be updated.

The suite state flags are not enforced by the AMS APIs. In other words, even though the `STATE_REMOVE_DENIED` or `STATE_UPDATE_DENIED` flags may be set to true, the AMS APIs do not prevent a removal or update if the appropriate method is invoked. It is up to the UI that implements the AMS APIs to enforce this behavior.

Programmers can use the getState() method to obtain the state information for the suites, then use the logical AND operator (&) to test if a given state is true. For example, to test if a suite is disabled:

```
if ((appSuite.getState() & SuiteInfo.STATE_DISABLED) != 0) {
        //  The app is disabled
}
```

The SuiteInfo interface contains the following methods to access basic information about a suite. Many methods throw a SuiteNotFoundException if the AMS can no longer locate the suite described by the SuiteInfo:

- java.lang.String[] getAvailableProperties() throws SuiteNotFoundException

  This method returns a String array that provides the names of the available properties. The properties returned are those from the JAD file and the manifest combined into a single array.

- java.lang.String getDownloadUrl()

  This method returns the URL that the JAD or JAR was downloaded from.

- byte[] getIcon()

  This method returns the icon representing the suite as a byte array. The AMS does not perform any decoding of the image, as this is the job of the AMS UI, so the image format is undefined.

- java.lang.String getName()

  This method returns the name for given suite.

- java.lang.String getProperty(java.lang.String name) throws SuiteNotFoundException

  This method returns the value of the property with the given name.

- SuiteSettings getSettings() throws SuiteNotFoundException

  This method returns the settings of the suite encapsulated in a SuiteSettings class. See "SuiteSetting Interface" on page 2-4 for more details on suite settings.

- int getState() throws SuiteNotFoundException

  This method returns the current state of the suite as a combination of flags: STATE_DISABLED, STATE_HIDDEN, STATE_AVAILABLE, STATE_REMOVE_DENIED, or STATE_UPDATE_DENIED. See Table 2–2 for more information.

- int getSuiteType() throws SuiteNotFoundException

  This method returns the suite type as one of the predefined constants shown in Table 2–1.

- java.lang.String getVendor()

  This method retrieves the vendor name for given suite

- void remove() throws SuiteLockedException, SuiteNotFoundException, SecurityException

  This method is used to remove the suite from the AMS. The method throws a SuiteLockedException if the suite is currently locked by the AMS. A suite is locked if the STATE_REMOVE_DENIED boolean is set to true.

- void remove(boolean ignoreRemoveLock) throws SuiteLockedException, SuiteNotFoundException, SecurityException

This method is used to remove the suite from the AMS, ignoring the STATE_REMOVE_DENIED lock if the boolean parameter is set to true.

- `boolean setState(int state, boolean value) throws SuiteLockedException, SuiteNotFoundException, ConcurrentModificationException`

  This method modifies the state of the suite, as per the constants shown in Table 2–2. The method returns the previous value of the state. If the suite is locked, the method throws a `SuiteLockedException`. A suite is locked if the `STATE_REMOVE_DENIED` boolean is set to true.

  Alternatively, if two threads attempt to modify the state of the suite at the same time, the method can throw a `ConcurrentModificationException`.

AppSuite Interface

The `AppSuite` interface extends the `SuiteInfo` interface, and is used to describe executable apps that are installed in the Oracle Java ME Embedded.

Apps can be one of three types: regular apps, system apps, or preinstalled apps, as shown in Table 2–3:

Table 2–3 Application Suite Types

Type	Description
`AT_PREINSTALLED`	The application suite is preinstalled.
`AT_REGULAR`	A normal application suite.
`AT_SYSTEM`	A system application suite.

To determine the type that this suite belongs to, use the `getType()` method, int `getType()` throws `SuiteNotFoundException`. This method returns the type of the suite as one of the predefined constants shown in Table 2–3. If the suite can no longer locate information about the app referenced by this descriptor, this method throws a `SuiteNotFoundException`.

The programmer can also obtain more detailed information about the suite with the following methods:

- `java.lang.String getDefaultApp() throws SuiteNotFoundException`

  This method returns the name of the default MIDlet from the suite. If the suite can no longer locate information about the app referenced by this descriptor, this method throws a `SuiteNotFoundException`.

- `int getType() throws SuiteNotFoundException`

  This method returns the type of the suite. If the suite cannot be found, this method throws a `SuiteNotFoundException`.

- `java.util.Enumeration getDependencies()`

  This method returns the dynamic components that this MIDlet suite depends on as an `Enumeration` of `LibSuite` object instances. Library suites are only installed when an application that has a dependency on them specifies them using this method.

- `boolean isTrusted()`

  This method returns a boolean indicating whether the AMS considers this application trusted via its signature and certificate authorities.

- `String getSecurityDomain() throws SuiteNotFoundException`

  This method returns the security domain that the suite is bound to.

In addition, you can use the AMS to start the app as a running task with either of the following methods:

- `TaskInfo startTask(java.lang.String className)`

  This method starts the application as a task from this suite, returning information about the executing task in a `TaskInfo` class. `TaskInfo` is covered in more detail in "TaskManager Interface" on page 6-1.

- `TaskInfo debugTask(java.lang.String className)`

  This method starts the application as a task from this suite in debug mode, returning information about the executing task in a `TaskInfo` class. `TaskInfo` is covered in more detail in"TaskManager Interface" on page 6-1.

LibSuite Interface

The `LibSuite` interface is used to provide descriptive information about a library suite that has been installed on the system. A library suite can have one of two types: regular and system, as shown in Table 2–4. Library suites can only be installed on a system if there is an application that has a dependency on them. See `AppSuite.getDependencies()` for more information.

Table 2–4 Library Suite Types

Type	Description
`LT_REGULAR`	A regular application library.
`LT_SYSTEM`	A system library.

To determine the type that this suite belongs to, use the `getType()` method, `int getType()`. This method returns the type of the suite as one of the predefined constants shown in Table 2–4 above.

SuiteSetting Interface

The `SuiteSetting` interface provides the data for a single suite setting. Each setting has an optional title to be displayed to user, an optional description, and number of choices. For example, the following represents a possible suite setting:

```
Title:
               Check for New Mail
Description:
               How often should the application check for new mail?
Choices:
               -Every 5 Minutes
               -Every 10 Minutes
               -Every 30 Minutes
               -Every Hour
               -Only When Requested
```

The following methods are provided by the `SuiteSetting` interface:

- `int getIdx()`

This method returns the integer index of the setting in the suite settings group. See below for more information on the SuiteSettingsGroup interface.

- `java.lang.String getTitle()`

  This method returns the title of the setting.

- `java.lang.String getDescription()`

  This method returns the description of the setting.

- `int getChoicesCount()`

  This method returns an integer indicating the number of choices for this setting.

- `int getSelectedChoice()`

  This method returns the index of the currently selected choice.

- `void setSelectedChoice(int newSelection) throws java.lang.IndexOutOfBoundsException`

  This method sets the current choice. This method throws a `java.lang.IndexOutOfBoundsException` if the selection index is not valid

- `java.lang.String getChoiceTitle(int idx)`

  This method returns the title of choice with specified index.

SuiteSettingsGroup Interface

`SuiteSettingsGroup` is an interface for a logical group of settings. Each group has an optional title, an optional description, and contains several individual settings defined using the `SuiteSetting` interface in "SuiteSetting Interface" on page 2-4. Each `SuiteSettingsGroup` can be part of a larger `SuiteSettings` object, defined in "SuiteSettings Interfaces" on page 2-5.

- `int getIdx()`

  This method returns the index of this settings group in a SuiteSettings object, defined in "SuiteSettings Interfaces" on page 2-5.

- `java.lang.String getTitle()`

  This method returns the settings group title.

- `java.lang.String getDescription()`

  This method returns the settings group description.

- `int getSettingsCount()`

  This method returns the number of individual settings in the group.

- `SuiteSetting getSetting(int idx)`

  This method returns the `SuiteSetting` with specified index.

SuiteSettings Interfaces

The `SuiteSettings` interface provides access to several `SuiteSettingsGroup` objects. Do not confuse the `SuiteSettings` (note the plural) object with the `SuiteSetting` object defined in "SuiteSetting Interface" on page 2-4.

The `SuiteSettings` interface provides the ability to save the settings to persistent storage using the `save()` method. Before doing so, however, the programmer must call the `checkForError()` method to ensure that no settings, especially those from other MIDlet suites, contain a mutually exclusive combination with settings in this object.

- `int getGroupsCount()`

  This method returns the number of suite settings groups contained in this object.

- `SuiteSettingsGroup getGroup(int idx)`

  This method return the suite settings group with specified index.

- `java.lang.String checkForError()`

  This method checks if any settings contain a mutually exclusive combination of setting values, including those from other MIDlet suites. If so, the method returns an error message; otherwise, it returns `null`. Only the first error is reported. Settings containing mutually exclusive combinations cannot be saved using the `save()` method of this interface.

- `java.lang.String checkForWarning()`

  This method checks if a given settings contain a potentially dangerous combination of setting values. If so, the method returns a warning message; otherwise the method returns `null`.

- `void save() throws java.lang.IllegalArgumentException`

  This method saves the suite settings. Before saving the settings, the programmer must check if these are valid using the `checkForError()` method. Settings are not be saved if there are errors. Settings should also be checked for warnings. If there are warnings, those must be shown to the user as per the MIDP 2.0 specification. This method throws a `java.lang.IllegalArgumentException` if the settings contain errors.

3 Installing Suites

This chapter discusses how to install suites using the AMS APIs.

First, any MIDlet that requires permission to install or remove other MIDlets must declare the respective permissions in its JAD descriptor:

```
MIDlet-Permissions: com.sun.ams.SuiteInstaller.start,
com.sun.ams.SuiteInfo.remove
```

The AMS APIs contain two installer interfaces: `AppInstaller` and `LinkInstaller`, both of which extend the base `SuiteInstaller` interface. Likewise, each installer provides for a listener, `AppInstallerListener` or `LinkInstallerListener`, which both extend from the `SuiteInstallerListener` interface, used to monitor an installer's progress.

SuiteInstaller Interface

The `SuiteInstaller` interface is a sub-interface that consists of only two methods: one that starts the installation and one that cancels the installation.

- `void start() throws SecurityException`

  This method begins the installation of a suite. This method returns immediately; the caller can observe the progress of the installation using a listener. A `SecurityException` can be thrown if installation of the MIDlet is prohibited.

- `void cancel()`

  This method cancels an installation that is in progress.

AppInstaller Interface

The `AppInstaller` interface is obtained from the `AmsFactory` class and extends the `SuiteInstaller` interface, and consists of five methods to initialize an app installation. With each installation, the programmer must provide the location of the app using either a URL that points to a JAR/JAD or a `SuiteInfo`, and an `AppInstallerProgressListener` that is called while the app is being installed. In addition, the programmer can provide an optional series of bytes that represents the app icon.

Each `initialize()` method returns a class that implements the `SuiteInfo` interface, which is a descriptor of the application that is being downloaded. Note that if the descriptor is not provided (for example, the `locationUrl` in the `initialize()` method points to a JAR file instead of a JAD), then the returned `SuiteInfo` object will return `null` for all methods except `getDownloadUrl()` and `getSuiteType()`.

The `SuiteInfo` object returned by each of the `initialize()` methods has no suite management methods implemented, so any calls to the following methods results in a `RuntimeException`:

- `SuiteInfo.getIcon()`
- `SuiteInfo.getDownloadUrl()`
- `SuiteInfo.remove()`
- `SuiteInfo.getState()`
- `SuiteInfo.setState(int state, boolean value)`
- `SuiteInfo.getSettings()`

Once the `AppInstaller` method is initialized, the programmer can call the `start()` method to begin the download and installation, monitoring the results with an `AppInstallerProgressListener`.

Here are the `initialize()` methods provided by the `AppInstaller` interface.

- `SuiteInfo initialize(java.lang.String locationUrl, AppInstallerProgressListener listener)`

  This method initializes an installer with the URL address of an app's JAD or JAR file and provides an installation progress listener. The function can result in network access. The installation progress listener must be present and ready to handle callback requests, such as querying the user for a valid login and password.

- `SuiteInfo initialize(java.lang.String locationUrl, AppInstallerProgressListener listener, boolean ignoreUpdateLock)`

  This method initializes an installer with the URL address of an app's JAD or JAR file and provides an installation progress listener. The function can result in network access. The installation progress listener must be present and ready to handle callback requests, such as querying the user for a valid login and password. This method contains a boolean parameter that, if set to true, ignores an update lock for an app if it is present.

- `SuiteInfo initialize(java.lang.String locationUrl, byte[] iconBytes, AppInstallerProgressListener listener)`

  This method initializes an installer with the URL address of an app's JAD or JAR file, a byte array that represents the icon of the app, and an installation progress listener. The function can result in network access. The installation progress listener must be present and ready to handle callback requests, such as querying the user for a valid login and password.

- `SuiteInfo initialize(java.lang.String jadUrl, java.lang.String jarUrl, AppInstallerProgressListener listener)`

  This method initializes an installer with the URL address of an app's JAD and JAR file and provides an installation progress listener. The function can result in network access. The installation progress listener must be present and ready to handle callback requests, such as querying the user for a valid login and password.

- `SuiteInfo initialize(java.lang.String jadUrl, java.lang.String jarUrl, byte[] iconBytes, AppInstallerProgressListener listener)`

  This method initializes an installer with the URL address of the app's JAD and the URL address of the app's JAR file, a byte array that represents the icon of the app, and an installation progress listener. The function can result in network access. The installation progress listener must be present and ready to handle callback requests, such as querying the user for a valid login and password.

- `SuiteInfo initialize(SuiteInfo suiteInfo, AppInstallerProgressListener listener) throws UnsupportedServiceException`

  This method initializes an installer the specified `SuiteInfo` descriptor, and an installation progress listener. The function is intended to use for installation from local storage but is not limited by such use case. The installation progress listener must be present and ready to handle callback requests, such as querying the user for a valid login and password. As this method is not yet implemented, it persistently throws an `UnsupportedServiceException`.

If the program must cancel the installation of the app, use the `cancel()` method.

LinkInstaller Interface

The `LinkInstaller` is obtained from the `AmsFactory` class and is used to download a link that references another application. It consists of only one `initialize()` method. Once the program has initialized a `LinkInstaller`, call the `start()` method to begin the download, monitoring the results with an `LinkInstallerProgressListener`.

```
SuiteInfo initialize(java.lang.String jadUrl, java.lang.String iconUrl,
LinkInstallerProgressListener listener)
```

This method initializes a link installer with the URL address of a JAD and icon file and provides an installation progress listener. The function can result in network access. The installation progress listener must be present and ready to handle callback requests, such as querying the user for a valid login and password.

If the program must cancel the installation of the link, use the `cancel()` method.

SuiteInstallerProgressListener Interface

`SuiteInstallerProgressListener` is a sub-interface that provides progress data for an installer that is downloading an app or a link.

The interface consists of two methods, both of which are called at certain times during installation. One is the `done()` method, which provides only a single code, the definitions of which can be found in the `InstallerErrorCode` interface. The other is the `updateStatus()` method, which identifies the current task as one of the five constants that are shown in Table 3–1, and provides an integer percentage of completeness.

Table 3–1 Progress Constants While Installing a Suite

Name	Description
`DOWNLOADING_BODY`	Install stage: downloading application body.
`DOWNLOADING_DATA`	Install stage: downloading additional application data.
`DOWNLOADING_DESCRIPTOR`	Install stage: downloading application descriptor.
`STORING`	Install stage: storing application.
`VERIFYING`	Install stage: verifying downloaded content.

Here are the two method defined in the `SuiteInstallerProgressListener` interface:

- `void done(int errorCode)`

  This method is called by the installer to report that the installation has completed. The resulting integer code is contained in the `InstallerErrorCode` class. See "InstallerErrorCode" on page 3-7 for more information on installation error codes.

- `void updateStatus(int stage, int percent)`

  This method is called by the installer to inform the listener of the current status of the install. The stage is given by an integer constant as shown in Table 3–1. The percent is an integer between 0 and 100.

AppInstallerProgressListener Interface

The `AppInstallerProgressListener` interface extends `SuiteInstallerProgressListener` and contains methods that the `AppInstaller` calls as it is downloading and installing an app. Any of the methods are called to request additional information from the user.

- `java.lang.String[] getNetworkAccessCredentials()`

  This method is called to ask user for login and password for network access. Typically the function is used for proxy authorization. This method should return a `String` array where first element is the login ID and the second element is the password. If the user wants to cancel the installation, the method should return `NULL`; doing so results in a call to the `done()` method with the `InstallerErrorCodes.CANCEL` constant. See "InstallerErrorCode" on page 3-7 for more information on installation error codes. The credentials provided are stored and reused, unless the credentials are invalid, at which point the user will be repeatedly asked for a proper login ID and password combination.

- `java.lang.String[] getResourceAccessCredentials()`

  This method is called to ask user for login and password for network resource access. This method should return a `String` array where first element is the login ID and the second element is the password. If the user wants to cancel the installation, the method should return `NULL`; doing so results in a call to the `done()` method with the `InstallerErrorCodes.CANCEL` constant. See "InstallerErrorCode" on page 3-7 for more information on installation error codes. The credentials provided are stored and reused, unless the credentials are invalid, at which point the user will be repeatedly asked for a proper login ID and password combination.

- `boolean confirmUpdate(int status)`

  This method is called to ask the user to confirm an update of an installed application. The integer status parameter can be one of `InstallerErrorCode.OLD_VERSION`, `InstallerErrorCode.ALREADY_INSTALLED`, or `InstallerErrorCode.NEW_VERSION`. This method should return true if the user wants to continue, or false if the user wants to cancel. Cancelling results in a call to the `done()` method with the `InstallerErrorCodes.CANCEL` constant. See "InstallerErrorCode" on page 3-7 for more information on installation error codes.

- `boolean confirmJarDownload(int totalSize)`

  This method is called to confirm an application download with the specified size in bytes. Dynamic components and RMS data are included as well. This method should return true if the user wants to continue, or false if the user wants to cancel. Cancelling results in a call to the `done()` method with the

`InstallerErrorCodes.CANCEL` constant. See "InstallerErrorCode" on page 3-7 for more information on installation error codes.

- `boolean keepRMS()`

  This method is called to ask the user to confirm if the Record Management Store (RMS) data should be kept for new version of an updated suite. This method should return true if the user wants to keep the RMS data for the suite, false otherwise.

- `boolean confirmAuthPath(java.lang.String[] authPath)`

  This method is called to ask the user to confirm the authentication path, presented as a String array. The *authentication path* is a list of certificate authorities. Here, descriptions of all the certificates in the chain should be provided for the user to review and authorize. The method should return true if the user wants to continue, or false if the user wants to cancel. Cancelling results in a call to the `done()` method with the `InstallerErrorCodes.CANCEL` constant. See "InstallerErrorCode" on page 3-7 for more information on installation error codes.

- `boolean confirmRedirect(java.lang.String newLocation)`

  This method is called with the URL when a request to be redirected to a new location is made. The method should return true if user wants to install the application suite from the new location, or false if the user wants to cancel. Cancelling results in a call to the `done()` method with the `InstallerErrorCodes.CANCEL` constant. See "InstallerErrorCode" on page 3-7 for more information on installation error codes.

- `boolean confirmUnsignedFxInstall()`

  This method is called to confirm to the user that they indeed with to install an unsigned JavaFX application. This method should return true if the user wants to continue, or false if the user wants to cancel. Cancelling results in a call to the `done()` method with the `InstallerErrorCodes.CANCEL` constant. See "InstallerErrorCode" on page 3-7 for more information on installation error codes.

- `boolean confirmGrantMaximumPermissions(Vector groupNames, boolean hasRisks)`

  This method is called during an installation or update to ask if the user wants to grant the maximum permissions allowed by MIDP specification to the MIDlet suite. The `groupNames` parameter is a `Vector` containing the names of permission groups that match permissions requested by this suite in its JAD or JAR. The `hasRisks` parameter can be set to true if `groupNames` contains high risk combinations. The method should return true if the user wants to grant permissions, false otherwise.

- `boolean confirmCurrentScreenSaverUpdate(java.lang.String name)`

  This method is called when the current screen saver MIDlet is being updated by a new screen saver MIDlet. The name of the MIDlet is provided. The method should return `true` if the user agrees with the update, or `false` if the user wants to stop the installation. Returning `false` results in a `SuiteInstallerProgressListener.done(InstallerErrorCodes.CANCELED)` progress update.

- `boolean confirmCurrentScreenSaverUnset(java.lang.String name)`

  This method is called when the current screen saver MIDlet is no longer set as the system screen saver. The name of the MIDlet is provided. The method should return `true` if the user agrees with the unsetting, or `false` if the user wants to stop the unsetting. Returning `false` results in a `SuiteInstallerProgressListener.done(InstallerErrorCodes.CANCELED)` progress update.

- `boolean confirmPersistentSuiteInstallation()`

  This method is called during an installation to ask if the user wants to install a permanent MIDlet suite. The method should return true if the user wants to continue, false otherwise.

- `java.lang.String getRmsEncryptionPassword()`

  This method is called to request the RMS encryption password from the user.

- `java.lang.String getRmsDecryptionPassword()`

  This method is called to request the RMS decryption password from the user.

- `boolean confirmInstallUnverified()`

  This method is called to ask the user to confirm an untrusted installation even though the MIDlet suite does not pass verification, likely due to an unknown certificate authority. This functionality is optional and may absent in some configurations. The method should return `true` if the user agrees with the install, or `false` if the user wants to stop the install. Returning `false` results in a `SuiteInstallerProgressListener.done(InstallerErrorCodes.CANCELED)` progress update.

- `boolean confirmRebindingServiceProviders(String[] serviceNames)`

  This method is called if new service providers are installed. The user can then be asked to confirm if he wants to perform rebinding existing applications with these new service providers. The method should return true if the user wants to perform the rebinding, false otherwise.

- `boolean confirmCertificateImport(Certificate cert)`

  This method is called to ask the user to confirm that a certificate that the MIDlet suite is signed with may be imported into the internal keystore. This functionality is optional and may absent in some configurations. The method should return true if the user wants to import the certificate, false otherwise.

LinkInstallerProgressListener Interface

The `LinkInstallerProgressListener` interface extends `SuiteInstallerProgressListener` and is used for processing link installer notifications, including asking for user credentials, and confirming if the user wants to perform an update. The interface consists of only two methods:

- `boolean confirmUpdate()`

  This method is called to ask the user to confirm an update of an installed link. This method should return true if the user wants to continue, or false if the user wants to cancel. If the user cancels, the `SuiteInstallerProgressListener.done()` method is called with `InstallerErrorCodes.CANCELED` constant.

- `java.lang.String[] getNetworkAccessCredentials()`

  This method is called to ask user for login and password for network access. Typically the function is used for proxy authorization. This method should return a `String` array where first element is the login ID and the second element is the password. The credentials provided are stored and reused, unless the credentials are invalid, at which point the user will be repeatedly asked for a proper login ID and password combination.

InstallerErrorCode

The `InstallerErrorCode` provides several constants used by the installation routines. These constants are shown in Table 3–2.

Table 3–2 Installer Error Codes

Constant	Error Code	Description
`ALAA_ALIAS_NOT_FOUND`	78	Application Level Access Authorization: The alias definition is missing.
`ALAA_ALIAS_WRONG`	80	Application Level Access Authorization: The alias definition is wrong.
`ALAA_MULTIPLE_ALIAS`	79	Application Level Access Authorization: An alias has multiple entries that match.
`ALAA_TYPE_WRONG`	77	Application Level Access Authorization: The `MIDlet-Access-Auth-Type` has missing parameters.
`ALREADY_INSTALLED`	39	The JAD matches a version of a suite already installed.
`APP_INTEGRITY_FAILURE_DEPENDENCY_CONFLICT`	69	Application Integrity Failure: two or more dependencies exist on the component with the same name and vendor, but have different versions or hashs.
`APP_INTEGRITY_FAILURE_DEPENDENCY_MISMATCH`	70	Application Integrity Failure: there is a component name or vendor mismatch between the component JAD and IMlet or component JAD that depends on it.
`APP_INTEGRITY_FAILURE_HASH_MISMATCH`	68	Application Integrity Failure: hash mismatch.
`ATTRIBUTE_MISMATCH`	50	A attribute in both the JAD and JAR manifest does not match.
`AUTHORIZATION_FAILURE`	49	Application authorization failure, possibly indicating that the application was not digitally signed.
`CA_DISABLED`	60	Indicates that the trusted certificate authority (CA) for this suite has been disabled for software authorization.
`CANCELED`	101	Canceled by user.
`CANNOT_AUTH`	35	The server does not support basic authentication.
`CIRCULAR_COMPONENT_DEPENDENCY`	64	Circular dynamic component dependency.
`COMPONENT_DEPS_LIMIT_EXCEEDED`	65	Dynamic component dependencies limit exceeded.
`COMPONENT_NAMESPACE_COLLISION`	72	The namespace used by a component is the same as another.

Table 3–2 (Cont.) Installer Error Codes

Constant	Error Code	Description
CONTENT_HANDLER_CONFLICT	55	The installation of a content handler would conflict with an already installed handler.
CORRUPT_DEPENDENCY_HASH	71	A dependency has a corrupt hash code.
CORRUPT_JAR	36	An entry could not be read from the JAR.
CORRUPT_PROVIDER_CERT	5	The content provider certificate cannot be decoded.
CORRUPT_SIGNATURE	8	The JAR signature cannot be decoded.
DEVICE_INCOMPATIBLE	40	The device does not support either the configuration or profile in the JAD.
DUPLICATED_KEY	88	Duplicated JAD/manifest key attribute
EXPIRED_CA_KEY	12	The certificate authority's public key has expired.
EXPIRED_PROVIDER_CERT	11	The content provider certificate has expired.
INCORRECT_FONT_LOADING	82	A font that is contained with the JAR cannot be loaded.
INSUFFICIENT_STORAGE	30	Not enough storage for this suite to be installed.
INVALID_CONTENT_HANDLER	54	The `MicroEdition-Handler-<`*n*`>` JAD attribute has invalid values.
INVALID_JAD_TYPE	37	The server did not have a resource with the correct type or the JAD downloaded has the wrong media type.
INVALID_JAD_URL	43	The JAD URL is invalid.
INVALID_JAR_TYPE	38	The server did not have a resource with the correct type or the JAR downloaded has the wrong media type.
INVALID_JAR_URL	44	The JAR URL is invalid.
INVALID_KEY	28	A key for an attribute is not formatted correctly.
INVALID_NATIVE_LIBRARY	85	A native library contained within the JAR cannot be loaded.
INVALID_PACKAGING	87	A dependency cannot be satisfied.
INVALID_PAYMENT_INFO	58	Indicates that the payment information provided with the IMlet suite is incomplete or incorrect.
INVALID_PROVIDER_CERT	7	The signature of the content provider certificate is invalid.
INVALID_RMS_DATA_TYPE	76	The server did not have a resource with the correct type or the JAD downloaded has the wrong media type.
INVALID_RMS_DATA_URL	73	The RMS data file URL is invalid.

Table 3–2 (Cont.) Installer Error Codes

Constant	Error Code	Description
`INVALID_SERVICE_EXPORT`	86	A LIBlet that exports a service with a LIBlet Services attribute does not contain the matching service provider configuration information.
`INVALID_SIGNATURE`	9	The signature of the JAR is invalid.
`INVALID_VALUE`	29	A value for an attribute is not formatted correctly.
`INVALID_VERSION`	16	The format of the version is invalid.
`IO_ERROR`	102	A low-level hardware error has occurred.
`JAD_MOVED`	34	The JAD URL for an installed suite is different than the original JAD URL.
`JAD_NOT_FOUND`	2	The JAD was not found.
`JAD_SERVER_NOT_FOUND`	1	The server for the JAD was not found.
`JAR_CLASSES_VERIFICATION_FAILED`	56	Not all classes within JAR package can be successfully verified with class verifier.
`JAR_IS_LOCKED`	100	Component or MIDlet or IMlet suite is locked by the system.
`JAR_NOT_FOUND`	20	The JAR was not found at the URL given in the JAD.
`JAR_SERVER_NOT_FOUND`	19	The server for the JAR was not found at the URL given in the JAD.
`JAR_SIZE_MISMATCH`	31	The JAR downloaded was not the same size as given in the JAD.
`MISSING_CONFIGURATION`	41	The configuration is missing from the manifest.
`MISSING_DEPENDENCY_HASH`	67	A dependency hash code is missing.
`MISSING_DEPENDENCY_JAD_URL`	66	A dependency JAD URL is missing.
`MISSING_JAR_SIZE`	21	The JAR size is missing.
`MISSING_JAR_URL`	18	The URL for the JAR is missing.
`MISSING_PROFILE`	42	The profile is missing from the manifest.
`MISSING_PROVIDER_CERT`	4	The content provider certificate is missing.
`MISSING_SUITE_NAME`	13	The name of MIDlet or IMlet suite is missing.
`MISSING_VENDOR`	14	The vendor is missing.
`MISSING_VERSION`	15	The version is missing.
`NEW_VERSION`	32	This suite is newer that the one currently installed.
`NO_ERROR`	0	No error.
`NOT_YET_VALID_PROVIDER_CERT`	89	A certificate is not yet valid.
`NOT_YET_VALID_CA_KEY`	90	A CA's public key is not yet valid.

Table 3–2 (Cont.) Installer Error Codes

Constant	Error Code	Description
`OLD_VERSION`	17	This suite is older that the one currently installed.
`OTHER_ERROR`	103	Other errors.
`PROXY_AUTH`	51	Indicates that the user must first authenticate with the proxy.
`PUSH_CLASS_FAILURE`	48	The class in a push attribute is not in `MIDlet-<`*n*`>` attribute.
`PUSH_DUP_FAILURE`	45	The connection in a push entry is already taken.
`PUSH_FORMAT_FAILURE`	46	The format of a push attribute has an invalid format.
`PUSH_PROTO_FAILURE`	47	The connection in a push attribute is not supported.
`REVOKED_CERT`	62	The certificate has been revoked.
`RMS_DATA_DECRYPT_PASSWORD`	83	Indicates that a password is required to decrypt RMS data.
`RMS_DATA_ENCRYPT_PASSWORD`	84	Indicates that a password is required to encrypt RMS data.
`RMS_DATA_NOT_FOUND`	75	The RMS data file was not found at the specified URL.
`RMS_DATA_SERVER_NOT_FOUND`	74	The server for the RMS data file was not found at the specified URL.
`RMS_INITIALIZATION_FAILURE`	81	Failure to import RMS data.
`SUITE_NAME_MISMATCH`	25	The MIDlet or IMlet suite name does not match the one in the JAR manifest.
`TOO_MANY_PROPS`	53	Indicates that either the JAD or manifest has too many properties to fit into memory.
`TRUSTED_OVERWRITE_FAILURE`	52	Indicates that the user tried to overwrite a trusted suite with an untrusted suite during an update.
`UNAUTHORIZED`	33	Web server authentication failed or is required.
`UNKNOWN_CA`	6	The certificate authority (CA) that issued the content provider certificate is unknown.
`UNKNOWN_CERT_STATUS`	63	The certificate is unknown to OCSP server.
`UNSUPPORTED_CERT`	10	The content provider certificate has an unsupported version.
`UNSUPPORTED_CHAR_ENCODING`	61	Indicates that the character encoding specified in the MIME type is not supported.
`UNSUPPORTED_PAYMENT_INFO`	57	Indicates that the payment information provided with the MIDlet or IMlet suite is incompatible with the current implementation.

Table 3–2 (Cont.) Installer Error Codes

Constant	Error Code	Description
`UNTRUSTED_PAYMENT_SUITE`	59	Indicates that the MIDlet or IMlet suite has payment provisioning information but it is not trusted.
`VENDOR_MISMATCH`	27	The vendor does not match the one in the JAR manifest.
`VERSION_MISMATCH`	26	The version does not match the one in the JAR manifest.

4

Suite Storage Manager

This chapter introduces the Suite Storage Manager. The Suite Storage Manager and its associated classes provide the primary interface for accessing all application, library, and link suites that are stored on the system.

SuiteStoreManager Interface

The `SuiteStoreManager` interface is obtained from the `AmsFactory` class and provides the main access to the applications, libraries, and links that have been installed on the AMS. Using the methods in this interface, the programmer can query against a suite for a specific name, vendor, or suite type. In addition, the programmer can install a listener that listens for changes in the suite storage.

The `SuiteStoreManager` interface has the following methods:

- `SuiteInfo getSuiteInfo(java.lang.String vendor, java.lang.String name)`

  This method returns a `SuiteInfo` descriptor of installed suite, given the name of the vendor and the suite. See "SuiteInstaller Interface" on page 3-1 for more information on the `SuiteInfo` interface.

- `SuiteInfo[] getSuites(int types)`

  This method returns list of installed suites of specified types, where the suite type is a constant in the `SuiteInfo` interface (`SuiteInfo.ST_APPLICATION`, `SuiteInfo.ST_LIBRARY`, or `SuiteInfo.ST_LINK`)

- `AppSuite[] getAppSuites()`

  This method returns a list of the currently installed app suites of type `SuiteInfo.ST_APPLICATION`.

- `LibSuite[] getLibSuites()`

  This method returns a list of the currently installed library suites of type `SuiteInfo.ST_LIBRARY`.

- `SuiteInfo[] getLinkSuites()`

  This method returns a list of the currently installed link suites of type `SuiteInfo.ST_LINK`.

- `void setStatusListener(SuiteStoreListener theListener)`

  This method assigns a `SuiteStoreListener` implementation to listen for changes to application suites. The programmer can also pass in `null` to remove the current listener.

SuiteStoreListener Interface

The `SuiteStoreListener` is an interface that is used to monitor changes to the suite storage. There are five methods that are called by the `SuiteStoreManager` to indicate that the state of a suite is changing. Each method passes in the `SuiteInfo` descriptor of the app, library, or link in question.

- `void notifySuiteInstalled(SuiteInfo suite)`

  This method is called to notify a listener that a suite has been installed.

- `void notifySuiteRemoved(SuiteInfo suite)`

  This method is called to notify a listener that a suite has been removed.

- `SuiteInstallerProgressListener notifySuiteInstalling(SuiteInfo suite)`

  This method is called to notify a listener that a suite is installing. The method must return an instance of `SuiteInstallerProgressListener` to be notified about installation process, or `null` if no notifications are required.

- `void notifySuiteSettingsChanged(SuiteInfo suite)`

  This method is called to notify a listener that the suite settings have been changed.

- `void notifySuiteStateChanged(SuiteInfo suite)`

  This method is called to notify a listener that the state of a suite has changed.

5

AMS Request Manager

This chapter discusses how to use the AMS Request Manager. The `AMSRequestManager` interface is obtained from the `AmsFactory` class and is used to set an `AMSRequestListener` to listen for any special requests for the AMS UI MIDlet. The interface consists of only one method, `void setEventListener(AMSRequestListener theListener)`. This method assigns the request manager listener.

AMSRequestListener Interface

The `AMSRequestListener` interface is used for processing system requests to the AMS. It consists of two methods:

- `void selectForegroundRequest()`

  This method notifies the listener that it has received a request to select one of several running applications. The AMS UI MIDlet should present the user with a list of possibilities, and allow he or she to select one application from the list. After the particular application is selected, the AMS UI is responsible for requesting foreground for it. This notification is delivered only to the MIDlet which is registered to be the main AMS MIDlet.

- `void switchToAMSRequest()`

  This method notifies the AMS UI that it has received a request to switch the main AMS screen, possibly as the result of the user pressing the "Home" button on the device. At this point, the AMS UI should set its main screen as the current displayable. This notification is delivered only to the MIDlet which is registered to be the main AMS MIDlet.

6

Tasks

This chapter discusses tasks in the AMS. A *task* is an application that is running, usually started using information that the programmer provides. There can be several tasks running at the same time. Use the `TaskManager`, which can be obtained from the `AmsFactory` class, to manage classes.

TaskManager Interface

The `TaskManager` interface provides several helpful methods. You can obtain a list of all tasks with the `getTaskList()` method. To obtain the `TaskInfo` for the application that is currently making the call, use the `getCurrentTask()` method. To obtain the `TaskInfo` of the task that the user is interacting with in the foreground, use the `getForegroundTask()` method.

To initiate a task, call the `startTask()` method. The following information must be provided:

- `suiteName` – Name of the suite where the task is launched from
- `vendorName` – Vendor of the suite
- `className` – Startup class

The programmer can also do this from the `AppSuite` class that represents the app.

The AMS allows the programmer to assign a `TaskManagerListener` that listens to any activity made by the `TaskManager` using the `setStatusListener()` method.

The `TaskManager` interface consists of the following methods:

- `TaskInfo startTask(java.lang.String suiteName, java.lang.String vendorName, java.lang.String className)`

  This method starts a task with given parameters. It returns a task descriptor as a `TaskInfo` class, which can control the task further, or `null` if the task cannot start.

- `TaskInfo startTaskWithOptions(java.lang.String suiteName, java.lang.String vendorName, java.lang.String className, int options)`

  This method starts a task with given parameters and options. It returns a task descriptor as a `TaskInfo` class, which can control the task further, or `null` if the task cannot start. See the Javadocs documentation for more information on possible options that can be passed into this method.

- `TaskInfo getForegroundTask()`

  This method returns the task descriptor of the current foreground task, or null if there is no current foreground class.

- `TaskInfo[] getTaskList(boolean includeSystem)`

  This method returns a list of running tasks, presented as an array of `TaskInfo` classes. The boolean `includeSystem` parameter indicates whether system tasks should be included in the list.

- `TaskInfo getCurrentTask()`

  This method returns the `TaskInfo` descriptor of the task the caller belongs to.

- `void setStatusListener(TaskManagerListener theListener)`

  This method assigns the task status update listener, using the `TaskManagerListener` interface defined in "TaskManagerListener Interface" on page 6-3.

TaskInfo

The `TaskInfo` interface describes information about any task this is currently running. A task is always in one of four states, and has a priority from 1 to 10, as shown in Table 6–1. You can also use this class to obtain the entry class of the task. Finally, the class has three constants used to report how a task has exited.

Table 6–1 Task Status Constants in the TaskInfo Interface

Name	Description
`DESTROYED`	The task was destroyed and the `TaskInfo` descriptor is invalid.
`PAUSED`	The task has been paused.
`RUNNING`	The task is currently running.
`STARTING`	The task is starting up.
`MIN_PRIORITY`	The minimum priority for a task. Equivalent to 1.
`NORM_PRIORITY`	The normal priority for a task. Equivalent to 5.
`MAX_PRIORITY`	The maximum priority for a task. Equivalent to 10.
`EXIT_REGULAR`	The task has finished its execution without any errors.
`EXIT_TERMINATED`	The task has been terminated.
`EXIT_FATAL_ERROR`	The task has finished its execution with a fatal error.

The `TaskInfo` interface contains the following methods:

- `java.lang.String getClassName()`

  This method returns a name of the entry class.

- `int getHeapUse()`

  This method returns the current heap usage of the task, in bytes.

- `LoggerInfo getLogger(String name)`

  This method returns the logger associated with the specified name.

- `Enumeration getLoggerNames()`

  This method returns an enumeration of all the logger names currently in use.

- `int getPriority()`

This method returns the priority of given task, expressed as an integer between 1 and 10. A higher number indicates a higher priority. The programmer can also use the three constants shown in Table 6–1.

- `int getStatus()`

  This method returns the current status of the task, expressed as an integer constant shown in Table 6–1 above.

- `SuiteInfo getSuiteInfo()`

  This method returns suite descriptor representing the task is executing.

- `boolean pauseTask()`

  This method pauses the task. The method returns true if successful, false otherwise.

- `boolean resumeTask()`

  This method resumes the task. The method returns true if successful, false otherwise.

- `boolean setForegroundTask()`

  This method sets the current task as the foreground task. The method returns true if successful, false otherwise.

- `boolean setPriority(int priority)`

  This method changes priority for the current task. A higher priority indicates a higher priority. The method returns true if successful, false otherwise. A higher number indicates a higher priority. The programmer can also use the three constants shown in Table 6–1.

- `boolean stopTask()`

  This method stops the current task. The method returns true if successful, false otherwise.

TaskManagerListener Interface

The `TaskManagerListener` interface is used to receive update information about a change in status of a particular task. It consists of two methods.

- `void notifyStatusUpdate(TaskInfo task, int newStatus)`

  This method notifies a listener about a task's new status, defined by the constants shown in Table 6–1.

- `void notifyTaskStopped(TaskInfo task, int exitCode)`

  This method notifies a listener when a task finishes its execution, providing an integer exit code as defined in the `TaskInfo` class.

LoggerInfo Interface

The `LoggerInfo` interface is a named descriptor that is used to log output. It consists of three methods.

- `com.oracle.util.logging.Level getLevel()`

  This method returns the log level that has been assigned to this logger.

- `void setLevel(com.oracle.logging.Level level)`

This method sets the log level for this logger. Only logging messages of this level or higher are recorded to the log.

- `String getName()`

  This method returns the string-based name of the `LoggerInfo` descriptor.

7

The Certificate Info Manager

This chapter introduces the Certificate Info Manager. The `CertificateInfoManager` interface, obtained from the `AmsFactory` class, is a starting point to begin working with installed certificates. Certificates are used to verify the signature of MIDlet suites that are installed by the AMS. The interface consists of only four methods:

- `CertificateInfo[] getCertificates()`

  This method returns an array containing all root certificates available in the system.

- `CertificateInfo[] getCertificates(String domain)`

  This method fetches all installed certificates for a specific domain, presented as one of three constants in Table 7–1.

- `void setStatusListener(CertificateManagerListener theListener)`

  This method assigns the certificate manager status listener.

CertificateInfo Interface

The `CertificateInfo` interface represents a Java ME certificate. The certificate can exist in one of three domains, as shown inTable 7–1.

Table 7–1 Certificate Domains

Name	Description
`DOMAIN_IDENTIFIED`	This constant indicates an identified third party security domain.
`DOMAIN_MANUFACTURER`	This constant indicates a manufacturer security domain.
`DOMAIN_OPERATOR`	This constant indicates an operator security domain.

The `CertificateInfo` interface has the following methods:

- `String getDomain()`

  This method returns the domain the certificate is bound to as a constant shown in Table 7–1.

- `long getNotAfter()`

  This method returns the end of the key's validity period in milliseconds since January 1, 1970.

- `long getNotBefore()`

This method returns the start of the key's validity period in milliseconds since January 1, 1970.

- `java.lang.String getOwner()`

  This method returns the distinguished name of the key's owner.

- `boolean isEnabled()`

  This method returns a boolean indicating if the certificate is enabled.

- `void setEnabled(boolean enabled)`

  This method sets the enabled status for this certificate.

CertificateManagerListener Interface

The `CertificateManagerListener` is an interface for processing certificate updates. It consists of four methods. Each method in the listener interface passes in a `CertificateInfo` that describes the certificate in question.

- `void notifyCertificateInstalled(CertificateInfo cert)`

  This method notifies a listener the certificate has been installed.

- `void notifyCertificateRemoved(CertificateInfo cert)`

  This method notifies a listener the certificate has been removed.

- `void notifyCertificateEnabled(CertificateInfo cert)`

  This method notifies a listener the certificate has been enabled.

- `void notifyCertificateDisabled(CertificateInfo cert)`

  This method notifies a listener the certificate has been disabled.

8

The Locale Change Notifier

This chapter discusses the Locale Change Notifier in the AMS. The `LocateChangeNotifier`, which is obtained from the `AmsFactory` class, is an interface for managing locale change notification subscriptions. It consists of two methods, which allow the programmer to add or remove listeners from the AMS:

- `void addLocaleChangeListener(LocaleChangeListener listener)`

  This method registers the specified locale change listener with the AMS.

- `void removeLocaleChangeListener(LocaleChangeListener listener)`

  This method removes the specified locale change listener with the AMS.

LocaleChangeListener Interface

This `LocaleChangeListener` interface is used to receive locale change notifications. It consists of only one method, `void localeChanged()`. This method is called to notify about a locale change. The new locale can be retrieved by querying the "`microedition.locale`" system property.

Glossary

Access Point

A network-connectivity configuration that is predefined on a device. An access point can represent different network profiles for the same bearer type, or for different bearer types that may be available on a device, such as WiFi or bluetooth.

ADC

Analog-to-Digital Converter. A hardware device that converts analog signals (time and amplitude) into a stream of binary numbers that can be processed by a digital device.

AMS

Application Management System. The system functionality that completes tasks such as installing applications, updating applications, and managing applications between foreground and background.

APDU

Application Protocol Data Unit. A communication mechanism used by SIM Cards and smart cards to communicate with card reader software or a card reader device.

API

Application Programming Interface. A set of classes used by programmers to write applications that provide standard methods and interfaces and eliminate the need for programmers to reinvent commonly used code.

ARM

Advanced RISC Machine. A family of computer processors using reduced instruction set (RISC) CPU technology, developed by ARM Holdings. ARM is a licensable instruction set architecture (ISA) and is used in the majority of embedded platforms.

AT commands

A set of commands developed to facilitate modem communications, such as dialing, hanging up, and changing the parameters of a connection. Also known as the Hayes command set, AT means *attention.*

AXF

ARM Executable Format. An ARM executable image generated by ARM tools.

BIP

Bearer Independent Protocol. Allows an application on a SIM Card to establish a data channel with a terminal, and through the terminal, to a remote server on the network.

CDMA

Code Division Multiple Access. A mobile telephone network standard used primarily in the United States and Canada as an alternative to GSM.

CLDC

Connected Limited Device Configuration. A Java ME platform configuration for devices with limited memory and network connectivity. It uses a low-footprint Java virtual machine such as the CLDC HotSpot Implementation, and several minimalist Java platform APIs for application services.

Configuration

Defines the minimum Java runtime environment (for example, the combination of a Java virtual machine and a core set of Java platform APIs) for a family of Java ME platform devices.

DAC

Digital-to-Analog Converter. A hardware device that converts a stream of binary numbers into an analog signal (time and amplitude), such as audio playback.

ETSI

European Telecommunications Standards Institute. An independent, non-profit group responsible for the standardization of information and communication technologies within Europe. Although based in Europe, it carries worldwide influence in the telecommunications industry.

Foreground switching

Changing which application is in the foreground by shifting the focus from one application to another.

GCF

Generic Connection Framework. A part of CLDC, it is a Java ME API consisting of a hierarchy of interfaces and classes to create connections (such as HTTP, datagram, or streams) and perform I/O.

GPIO

General Purpose Input/Output. Unassigned pins on an embedded platform that can be assigned or configured as needed by a developer.

GPIO Port

A group of GPIO pins (typically 8 pins) arranged in a group and treated as a single port.

GSM

Global System for Mobile Communications. A 3G mobile telephone network standard used widely in Europe, Asia, and other parts of the world.

HTTP

HyperText Transfer Protocol. The most commonly used Internet protocol, based on TCP/IP that is used to fetch documents and other hypertext objects from remote hosts.

HTTPS

Secure HyperText Transfer Protocol. A protocol for transferring encrypted hypertext data using Secure Socket Layer (SSL) technology.

ICCID

Integrated Circuit Card Identification. The unique serial number assigned to an individual SIM Card.

IMP-NG

Information Module Profile Next Generation. A profile for embedded "headless" devices, the IMP-NG specification (JSR 228) is a subset of MIDP 2.0 that leverages many of the APIs of MIDP 2.0, including the latest security and networking+, but does not include graphics and user interface APIs.

IMEI

International Mobile Equipment Identifier. A number unique to every mobile phone. It is used by a GSM or UMTS network to identify valid devices and can be used to stop a stolen or blocked phone from accessing the network. It is usually printed inside the battery compartment of the phone.

IMlet

An application written for IMP-NG. An IMlet does not differ from MIDP 2.0 MIDlet, except by the fact that an IMlet can not refer to MIDP classes that are not part of IMP-NG. An IMlet can only use the APIs defined by the IMP-NG and CLDC specifications.

IMlet Suite

A way of packaging one or more IMlets for easy distribution and use. Similar to a MIDlet suite, but for smaller applications running in an embedded environment.

IMSI

International Mobile Subscriber Identity. A unique number associated with all GSM and UMTS network mobile phone users. It is stored on the SIM Card inside a phone and is used to identify itself to the network.

I2C

Inter-Integrated Circuit. A multi-master, serial computer bus used to attach low-speed peripherals to an embedded platform

ISA

Instruction Set Architecture. The part of a computer’s architecture related to programming, including data type, addressing modes, interrupt and exception handling, I/O, and memory architecture, and native commands. Reduced instruction set computing (RISC) is one kind of instruction set architecture.

JAD file

Java Application Descriptor file. A file provided in a MIDlet or IMlet suite that contains attributes used by application management software (AMS) to manage the MIDlet or IMlet life cycle, and other application-specific attributes used by the MIDlet or IMlet suite itself.

JAR file

Java Archive file. A platform-independent file format that aggregates many files into one. Multiple applications written in the Java programming language and their required components (class files, images, sounds, and other resource files) can be bundled in a JAR file and provided as part of a MIDlet or IMlet suite.

JCP

Java Community Process. The global standards body guiding the development of the Java programming language.

JDTS

Java Device Test Suite. A set of Java programming language tests developed specifically for the wireless marketplace, providing targeted, standardized testing for CLDC and MIDP on small and handheld devices.

Java ME platform

Java Platform, Micro Edition. A group of specifications and technologies that pertain to running the Java platform on small devices, such as cell phones, pagers, set-top boxes, and embedded devices. More specifically, the Java ME platform consists of a configuration (such as CLDC) and a profile (such as MIDP or IMP-NG) tailored to a specific class of device.

JSR

Java Specification Request. A proposal for developing new Java platform technology, which is reviewed, developed, and finalized into a formal specification by the JCP program.

Java Virtual Machine

A software “execution engine” that safely and compatibly executes the byte codes in Java class files on a microprocessor.

KVM

A Java virtual machine designed to run in a small, limited memory device. The CLDC configuration was initially designed to run in a KVM.

LCDUI

Liquid Crystal Display User Interface. A user interface toolkit for interacting with Liquid Crystal Display (LCD) screens in small devices. More generally, a shorthand way of referring to the MIDP user interface APIs.

MIDlet

An application written for MIDP.

MIDlet suite

A way of packaging one or more MIDlets for easy distribution and use. Each MIDlet suite contains a Java application descriptor file (`.jad`), which lists the class names and files names for each MIDlet, and a Java Archive file (`.jar`), which contains the class files and resource files for each MIDlet.

MIDP

Mobile Information Device Profile. A specification for a Java ME platform profile, running on top of a CLDC configuration that provides APIs for application life cycle, user interface, networking, and persistent storage in small devices.

MSISDN

Mobile Station Integrated Services Digital Network. A number uniquely identifying a subscription in a GSM or UMTS mobile network. It is the telephone number to the SIM Card in a mobile phone and used for voice, FAX, SMS, and data services.

MVM

Multiple Virtual Machines. A software mode that can run more than one MIDlet or IMlet at a time.

Obfuscation

A technique used to complicate code by making it harder to understand when it is decompiled. Obfuscation makes it harder to reverse-engineer applications and therefore, steal them.

Optional Package

A set of Java ME platform APIs that provides additional functionality by extending the runtime capabilities of an existing configuration and profile.

Preemption

Taking a resource, such as the foreground, from another application.

Preverification

Due to limited memory and processing power on small devices, the process of verifying Java technology classes is split into two parts. The first part is preverification which is done off-device using the preverify tool. The second part, which is verification, occurs on the device at runtime.

Profile

A set of APIs added to a configuration to support specific uses of an embedded or mobile device. Along with its underlying configuration, a profile defines a complete and self-contained application environment.

Provisioning

A mechanism for providing services, data, or both to an embedded or mobile device over a network.

Pulse Counter

A hardware or software component that counts electronic pulses, or events, on a digital input line, for example, a GPIO pin.

Push Registry

The list of inbound connections, across which entities can push data. Each item in the list contains the URL (protocol, host, and port) for the connection, the entity permitted to push data through the connection, and the application that receives the connection.

RISC

Reduced Instruction Set Computing. A CPU design based on simplified instruction sets that provide higher performance and faster execution of individual instructions. The ARM architecture is based on RISC design principles.

RL-ARM

Real-Time Library. A group of tightly coupled libraries designed to solve the real-time and communication challenges of embedded systems based on ARM processor-based microcontroller devices.

RMI

Remote Method Invocation. A feature of Java SE technology that enables Java technology objects running in one virtual machine to seamlessly invoke objects running in another virtual machine.

RMS

Record Management System. A simple record-oriented database that enables an IMlet or MIDlet to persistently store information and retrieve it later. MIDlets can also use the RMS to share data.

RTOS

Real-Time Operating System. An operating system designed to serve real-time application requests. It uses multi-tasking, an advanced scheduling algorithm, and minimal latency to prioritize and process data.

RTSP

Real Time Streaming Protocol. A network control protocol designed to control streaming media servers and media sessions.

RTX

The real-time operating system used on the Keil MCBSTM32F200 embedded platform. The Oracle Java ME Embedded software runs on the Keil platform.

SCWS

Smart Card Web Server. A web server embedded in a smart card (such as a SIM Card) that allows HTTP transactions with the card.

SD card

Secure Digital cards. A non-volatile memory card format for use in portable devices, such as mobile phones and digital cameras, and embedded systems. SD cards come in three different sizes, with several storage capacities and speeds.

SIM

Subscriber Identity Module. An integrated circuit embedded into a removable SIM card that securely stores the International Mobile Subscriber Identity (IMSI) and the related key used to identify and authenticate subscribers on mobile and embedded devices.

Slave Mode

Describes the relationship between a master and one or more devices in a Serial Peripheral Interface (SPI) bus arrangement. Data transmission in an SPI bus is initiated by the master device and received by one or more slave devices, which cannot initiate data transmissions on their own.

Smart Card

A card that stores and processes information through the electronic circuits embedded in silicon in the substrate of its body. Smart cards carry both processing power and information. A SIM Card is a special kind of smart card for use in a mobile device.

SMS

Short Message Service. A protocol allowing transmission of short text-based messages over a wireless network. SMS messaging is the most widely-used data application in the world.

SMSC

Short Message Service Center. The SMSC routes messages and regulates **SMS** traffic. When an SMS message is sent, it goes to an SMS center first, then gets forwarded to the destination. If the destination is unavailable (for example, the recipient embedded board is powered down), the message is stored in the SMSC until the recipient becomes available.

SOAP

Simple Object Access Protocol. An XML-based protocol that enables objects of any type to communicate in a distributed environment. It is most commonly used to develop web services.

SPI

Serial Peripheral Interface. A synchronous bus commonly used in embedded systems that allows full-duplex communication between a master device and one or more slave devices.

SSL

Secure Sockets Layer. A protocol for transmitting data over the Internet using encryption and authentication, including the use of digital certificates and both public and private keys.

SVM

Single Virtual Machine. A software mode that can run only one MIDlet or IMlet at a time.

Task

At the platform level, each separate application that runs within a single Java virtual machine is called a task. The API used to instantiate each task is a stripped-down version of the Isolate API defined in JSR 121.

TCP/IP

Transmission Control Protocol/Internet Protocol. A fundamental Internet protocol that provides for reliable delivery of streams of data from one host to another.

Terminal Profile

Device characteristics of a terminal (mobile or embedded device) passed to the SIM Card along with the IMEI at SIM Card initialization. The terminal profile tells the SIM Card what values are supported by the device.

UART

Universal Asynchronous Receiver/Transmitter. A piece of computer hardware that translates data between serial and parallel formats. It is used to facilitate communication between different kinds of peripheral devices, input/output streams, and embedded systems, to ensure universal communication between devices.

UICC

Universal Integrated Circuit Card. The smart card used in mobile terminals in GSM and UMTS networks. The UICC ensures the integrity and security of personal data on the card.

UMTS

Universal Mobile Telecommunications System. A third-generation (3G) mobile communications technology. It utilizes the radio spectrum in a fundamentally different way than GSM.

URI

Uniform Resource Identifier. A compact string of characters used to identify or name an abstract or physical resource. A URI can be further classified as a uniform resource locator (URL), a uniform resource name (URN), or both.

USAT

Universal SIM Application Toolkit. A software development kit intended for 3G networks. It enables USIM to initiate actions that can be used for various value-added services, such as those required for banking and other privacy related applications.

USB

Universal Serial Bus. An industry standard that defines the cables, connectors, and protocols used in a bus for connection, communication, and power supply between computers and electronic devices, such as embedded platforms and mobile phones.

USIM

Universal Subscriber Identity Module. An updated version of a SIM designed for use over 3G networks. USIM is able to process small applications securely using better cryptographic authentication and stronger keys. Larger memory on USIM enables the addition of thousands of contact details including subscriber information, contact details, and other custom settings.

WAE

Wireless Application Environment. An application framework for small devices, which leverages other technologies, such as Wireless Application Protocol (WAP).

WAP

Wireless Application Protocol. A protocol for transmitting data between a server and a client (such as a cell phone or embedded device) over a wireless network. WAP in the wireless world is analogous to HTTP in the World Wide Web.

Watchdog Timer

A dedicated piece of hardware or software that "watches" an embedded system for a fault condition by continually polling for a response. If the system goes offline and no response is received, the watchdog timer initiates a reboot procedure or takes other steps to return the system to a running state.

WCDMA

Wideband Code Division Multiple Access. A detailed protocol that defines how a mobile phone communicates with the tower, how its signals are modulated, how datagrams are structured, and how system interfaces are specified.

WMA

Wireless Messaging API. A set of classes for sending and receiving Short Message Service (SMS) messages.

XML Schema

A set of rules to which an XML document must conform to be considered valid.

Index