Technical Manual

Motorola C975

J2ME™ Developer Guide

Version 1.0

Table of Contents

TABLE OF CONTENTS **2**

1 INTRODUCTION **6**

PURPOSE 6
AUDIENCE 6
DISCLAIMER 6
REFERENCES 7
REVISION HISTORY 7
DEFINITIONS, ABBREVIATIONS, ACRONYMS 8
DOCUMENT OVERVIEW 8

2 J2ME INTRODUCTION **10**

THE JAVA 2 PLATFORM, MICRO EDITION (J2ME) 10
THE MOTOROLA J2ME PLATFORM 11
RESOURCES AND API'S AVAILABLE 11

3 DEVELOPING AND PACKAGING J2ME APPLICATIONS **12**

GUIDE TO DEVELOPMENT IN J2ME 12

4 DOWNLOADING APPLICATIONS **14**

METHOD OF DOWNLOADING 14

5 APPLICATION MANAGEMENT **15**

DOWNLOADING A JAR WITHOUT A JAD 15
INSTALLATION AND DELETION STATUS REPORTS 15
DRM CONTENT PROTECTION IN JAVA 16

6 SHARED JAD URLS **17**

INTRODUCTION 17
Tell-A-Friend Option *17*

7 JAD ATTRIBUTES **18**

JAD / MANIFEST ATTRIBUTE IMPLEMENTATIONS 18

8 MIDLET STORAGE IN REMOVABLE MEMORY **21**

INTRODUCTION 21
Installing downloaded applications into removable memory *21*
Listing and Launch J2ME Applications from removable memory *22*

9 ITAP **24**

INTELLIGENT KEYPAD TEXT ENTRY API 24

10 NETWORK APIS **25**

NETWORK CONNECTIONS 25
USER PERMISSION 27
INDICATING A CONNECTION TO THE USER 27
HTTPS CONNECTION 28
DNS IP 29
PUSH REGISTRY 30
MECHANISMS FOR PUSH 30
PUSH REGISTRY DECLARATION 30
DELIVERY OF A PUSH MESSAGE 38
DELETING AN APPLICATION REGISTERED FOR PUSH 39
SECURITY FOR PUSH REGISTRY 39
NETWORK ACCESS 39

11 INTERFACE COMMCONNECTION **41**

COMMCONNECTION 41
ACCESSING 41
PARAMETERS 41
BNF FORMAT FOR CONNECTOR.OPEN () STRING 42
COMM SECURITY 43
PORT NAMING CONVENTION 44
METHOD SUMMARY 44

12 JSR120 – WIRELESS MESSAGING API **45**

WIRELESS MESSAGING API (WMA) 45
SMS CLIENT MODE AND SERVER MODE CONNECTION 45
SMS PORT NUMBERS 46
SMS STORING AND DELETING RECEIVED MESSAGES 47
SMS MESSAGE TYPES 47
SMS MESSAGE STRUCTURE 47
SMS NOTIFICATION 47
APP INBOX CLEAN-UP 53

13 JSR 135 – MOBILE MEDIA API **54**

JSR 135 MOBILE MEDIA API 54
TONECONTROL 56
VOLUMECONTROL 56
STOPTIMECONTROL 56
MANAGER CLASS 57
AUDIO MEDIA 57
MOBILE MEDIA FEATURE SETS 59
Supported Multimedia File Types *62*

14 JSR 139 – CLDC 1.1 .... **66**

JSR 139 66

15 JSR 184 – 3D API **71**

OVERVIEW 71
MOBILE 3D API 71
Mobile 3D API File Format Support *72*
Mobile 3D Graphics – M3G API *72*

16 PHONEBOOK ACCESS API **80**
Phonebook Access API 80
Phonebook Access API Permissions 81
17 TELEPHONY API **90**
Dialer Class 90
Class DialerEvent 90
Class Dialer 92
getDefaultDialer *93*
setDialerListener *93*
startCall *93*
startCall *94*
sendExtNo *94*
endCall *95*
Interface DialerListener 95
Sample DialerListener Implementation 95
notifyDialerEvent *97*
Class Hierarchy 97
Interface Hierarchy 97
18 JSR 185 - JTWI **98**
Overview 98
CLDC related content for JTWI 99
MIDP 2.0 specific information for JTWI 100
Wireless Messaging API 1.1 (JSR 120) specific content for JTWI 101
Mobile Media API 1.1 (JSR 135) specific content for JTWI 102
MIDP 2.0 Security specific content for JTWI 102
19 MIDP 2.0 SECURITY MODEL **103**
Untrusted MIDlet Suites 104
Untrusted Domain 104
Trusted MIDlet Suites 105
Permission Types concerning the Handset 105
User Permission Interaction Mode 105
Implementation based on Recommended Security Policy 106
Trusted 3rd Party Domain 106
Security Policy for Protection Domains 107
Displaying of Permissions to the User 110
Trusted MIDlet Suites Using x.509 PKI 110
Signing a MIDlet Suite 111
Signer of MIDlet Suites 111
MIDlet Attributes Used in Signing MIDlet Suites 111
Creating the Signing Certificate 112
Inserting Certificates into JAD 112
Creating the RSA SHA-1 signature of the JAR 112
Authenticating a MIDlet Suite 113
Verifying the Signer Certificate 113
Verifying the MIDlet Suite JAR 114
Carrier Specific Security Model 115
Bound Certificates 115
APPENDIX A: KEY MAPPING **117**

Key Mapping for the Motorola C975 117

APPENDIX B: MEMORY MANAGEMENT CALCULATION 119

Available Memory 119
Memory Calculation for MIDlets 119

APPENDIX C: FAQ 120

Online FAQ 120

APPENDIX F: SPEC SHEET 121

Motorola C975 Spec Sheet 121

1
Introduction

Purpose

This document describes the application program interfaces used to develop Motorola compliant Java™ 2 Platform, Micro Edition (J2ME™) applications for the Motorola C975.

Audience

This document is intended for developers involved with the development of J2ME applications for the Motorola C975.

Disclaimer

Motorola reserves the right to make changes without notice to any products or services described herein. "Typical" parameters, which may be provided in Motorola Data sheets and/or specifications can and do vary in different applications and actual performance may vary. Customer's technical experts will validate all "Typicals" for each customer application.

Motorola makes no warranty with regard to the products or services contained herein. Implied warranties, including without limitation, the implied warranties of merchantability and fitness for a particular purpose, are given only if specifically required by applicable law. Otherwise, they are specifically excluded.

No warranty is made as to coverage, availability, or grade of service provided by the products or services, whether through a service provider or otherwise.

No warranty is made that the software will meet your requirements or will work in combination with any hardware or applications software products provided by third parties, that the operation of the software products will be uninterrupted or error free, or that all defects in the software products will be corrected.

In no event shall Motorola be liable, whether in contract or tort (including negligence), for any damages resulting form use of a product or service described herein, or for any indirect, incidental, special or consequential damages of any kind, or loss of revenue or profits, loss of business, loss of information or data, or other financial loss arising out of or in connection with the ability or inability to use the Products, to the full extent these damages may be disclaimed by law.

Some states and other jurisdictions do not allow the exclusion or limitation of incidental or consequential damages, or limitation on the length of an implied warranty, so the above limitations or exclusions may not apply to you.

This warranty gives you specific legal rights, and you may also have other rights, which vary from jurisdiction to jurisdiction.

Motorola products or services are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Motorola product or service could create a situation where personal injury or death may occur.

Should the buyer purchase or use Motorola products or services for any such unintended or unauthorized application, buyer shall release, indemnify and hold Motorola and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Motorola was negligent regarding the designing or manufacture of the product or service.

References

Reference	Link
MIDP 2.0	http://java.sun.com/products/midp/
JSR 118	http://www.jcp.org
JSR 120	http://www.jcp.org
JSR 135	http://www.jcp.org
Sun J2ME	http://java.sun.com/j2me/
SAR	http://www.wapforum.org

Revision History

Version	Date	Name	Reason
00.01	July 08, 2004	C.E.S.A.R.	Initial Draft

00.02	August 30, 2004	C.E.S.A.R.	Updates after Motorola's review
00.03	September 08, 2004	C.E.S.A.R.	Updates after review

Definitions, Abbreviations, Acronyms

Acronym	Description
AMS	Application Management Software
API	Application Program Interface
CLDC	Connected, Limited Device Configuration
DRM	Digital Rights Management
IDE	Integrated Development Environments
JAD	Java Application Descriptor
JAM	Java Application Manager
JAR	Java Archive
J2ME	Java 2 Micro Edition
JSR 120	Java Specification Request 120 defines a set of optional APIs that provides standard access to wireless communication resources.
MIDP	Mobile Information Device Profile
MMA	Multimedia API
MMS	Multimedia Messaging Service
OTA	Over The Air
SAR	Segmentation & Reassembly
SDK	Software Development Kit
WMA	Wireless Messaging API

Document Overview

This developer's guide is organized into the following chapters and appendixes:

Chapter 1 – Introduction: this chapter has general information about this document, including purpose, scope, references, and definitions.

Chapter 2 – J2ME Introduction: this chapter describes the J2ME platform and the available resources on the Motorola C975.

Chapter 3 – Developing and Packaging J2ME Applications: this chapter describes some details about J2ME, development tools, specifications and some general concepts.

Chapter 4 – Downloading Applications: this chapter describes the method for downloading applications on devices and some details about available options for Motorola C975.

Chapter 5 – Application Management: this chapter describes the application management scheme for the Motorola C975, including DRM Content Protection in Java.

Chapter 6 – Shared JAD URLs: this chapter describes briefly a new feature that allows users to share their downloaded J2ME application URLs with others.

Chapter 7 – JAD Attributes: this chapter describes what attributes are supported.

Chapter 8 – MIDlet Storage in Removable Memory: this chapter describes how to install, list and launch downloaded applications on removable memory.

Chapter 9 – iTAP: this chapter describes iTAP support.

Chapter 10 – Network API: this chapter describes the Java Networking API and network access.

Chapter 11 – CommConnection Interface: this chapter describes the CommConnection API.

Chapter 12 – JSR 120 – Wireless Messaging API: this chapter describes JSR 120 implementation.

Chapter 13 – JSR 135 – Mobile Media API: this chapter describes image types and supported formats.

Chapter 14 – JSR 139 – CLDC 1.1: this chapter describes briefly some characteristics of CLDC 1.1 and presents additional classes, fields, and methods supported for CLDC 1.1.

Chapter 15 – JSR 184 – 3D API: this chapter describes JSR 184.

Chapter 16 – Phonebook Access API: this chapter describes the Phonebook Access API.

Chapter 17 – Telephony API – this chapter describes the Telephony API.

Chapter 18 – JSR 185 – JTWI – this chapter describes JTWI functionality.

Chapter 19 – MIDP 2.0 Security Model: this chapter describes the MIDP 2.0 default security model.

Appendix A – Key Mapping: this appendix describes the key mapping for the Motorola C975, including the key name, key code and game action of all Motorola keys.

Appendix B – Memory Management Calculation: this appendix describes the memory management calculations.

Appendix C – FAQ: this appendix provides a link to the dynamic online FAQ.

Appendix F – Spec Sheet: this appendix provides spec sheets for the Motorola C975 handset.

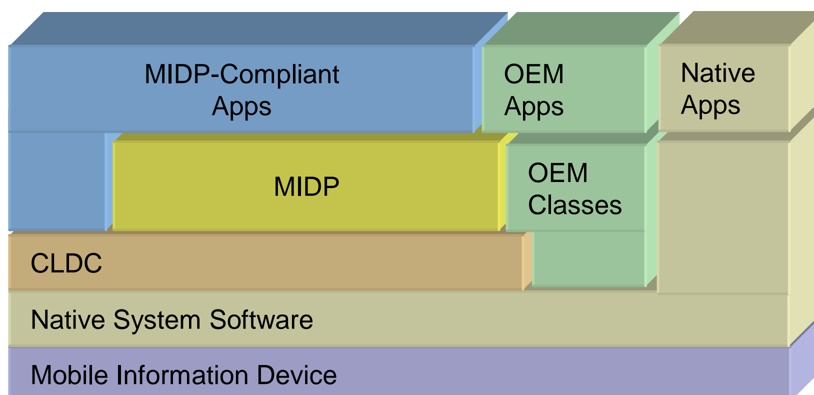
2
J2ME Introduction

The Motorola C975 includes the Java™ 2 Platform, Micro Edition, also known as the J2ME platform. The J2ME platform enables developers to easily create a variety of Java applications ranging from business applications to games. Prior to its inclusion, services or applications residing on small consumer devices like cell phones could not be upgraded or added to without significant effort. By implementing the J2ME platform on devices like the Motorola C975, service providers, as well as customers, can easily add and remove applications allowing for quick and easy personalization of each device. This chapter of the guide presents a quick overview of the J2ME environment and the tools that can be used to develop applications for the Motorola C975.

The Java 2 Platform, Micro Edition (J2ME)

The J2ME platform is a new, very small application environment. It is a framework for the deployment and use of Java technology in small devices such as cell phones and pagers. It includes a set of APIs and a virtual machine that is designed in a modular fashion allowing for scalability among a wide range of devices.

The J2ME architecture contains three layers consisting of the Java Virtual Machine, a Configuration Layer, and a Profile Layer. The Virtual Machine (VM) supports the Configuration Layer by providing an interface to the host operating system. Above the VM is the Configuration Layer, which can be thought of as the lowest common denominator of the Java Platform available across devices of the same "horizontal market." Built upon this Configuration Layer is the Profile Layer, typically encompassing the presentation layer of the Java Platform.



The Configuration Layer used in the Motorola C975 is the Connected Limited Device Configuration 1.0 (CLDC 1.0) and the Profile Layer used is the Mobile Information Device Profile 2.0 (MIDP 2.0). Together, the CLDC and MIDP provide common APIs for I/O, simple math functionality, UI, and more.

For more information on J2ME, see the Sun™ J2ME documentation (http://java.sun.com/j2me/).

The Motorola J2ME Platform

Functionality not covered by the CLDC and MIDP APIs is left for individual OEMs to implement and support. By adding to the standard APIs, manufacturers can allow developers to access and take advantage of the unique functionality of their handsets.

The Motorola C975 contains OEM APIs for extended functionality ranging from enhanced UI to advanced data security. While the Motorola C975 can run any application written in standard MIDP, it can also run applications that take advantage of the unique functionality provided by these APIs. These OEM APIs are described in this guide.

Resources and API's Available

MIDP 2.0 will provide support to the following functional areas on the Motorola C975:

MIDP 2.0

- Application Management Software
- Digital Rights Management for MIDlets
- Alpha Blending for Image
- File Image Support (.JPEG / .JFIF, .JPEG / .EXIF, .GIF[87a, 89a], BMP, WBMP, EMS BMP)
- Networking Interface
- Gaming Interface
- Sounds
- Timers
- User Interface
- Multimedia

Additional Functionality

- Wireless Messaging API (JSR 120)
- Mobile Media API (JSR 135)
- MIDlet storage in removable memory
- Increase RAM for Java Apps
- CLDC Next Generation (JSR 139)
- Phone Book

3
Developing and Packaging J2ME Applications

Guide to Development in J2ME

Introduction

In this chapter, previous experience in J2ME development is required for the reader. This chapter will also provide some information about Connected, Limited Device Configuration (CLDC) and Mobile Information Device Profile (MIDP).

More information and materials can be found on websites maintained by Motorola, Sun Microsystems and others. Consult the following URLs for this information and materials:

- http://www.motocoder.com
- http://www.java.sun.com/j2me
- http://www.corej2me.com

The following line briefly describes some details about J2ME and some general concepts.

A MIDlet is an abstract class that is subclassed to form the basis of an application. It is the heart of a MIDP application and allows the device to start, pause and destroy an application.

The MIDlet will consist of two core specifications, namely Connected, Limited Device Configuration (CLDC), where the specification can be located at the http://www.jcp.org/ website, and Mobile Information Device Profile (MIDP), located at same address. Both specifications can be found in Java Specification Requests.

The specifications mentioned are:

- MIDP 1.0 (JSR 37)
- MIDP 2.0 (JSR 118)
- CLDC 1.0.4 (JSR 30)
- CLDC 1.1 (139)

There are key points to consider for an initial development, they are: processing power, memory size, screen capabilities and wireless networking characteristics. They are very important for the development of a MIDlet.

These characteristics are applied to devices specifics. For example, considering networking conditions for an application, it would only apply for networked applications such as email clients, among others.

About tools, there is an array of tools available to assist in the development. The Companies' Software Development Kits (SDK), such as Sun, that can run inside of an Integrated Development Environments (IDEs) can be found free or purchased. These kits present several options to facilitate the development of an application.

In addition to the IDEs and Sun SDK for development, Motorola offers access to their own SDK that contains Motorola device emulators. From here, a MIDlet can be built and then deployed onto an emulated target handset. This will enable debugging and validation of the MIDlet before deployment to a real, physical handset. The latest Motorola SDK can be downloaded from the MOTOCODER website.

4
Downloading Applications

Method of Downloading

The load of applications (MIDlets) in Motorola devices that consist of the transmission of an application from PC to device can be carried through the direct cable USB, via CE Bus.

The direct cable approach can be performed using a tool available from MOTOCODER called MIDway. The version available of writing is 2.8, which supports USB cable download.
It is important to note that the MIDway tool will only work with a device that has been enabled to support direct cable Java download. This feature is not available by purchasing a device through a standard consumer outlet.
The easiest method of confirming support for this is by looking at the "Java Tool" menu on the phone in question and seeing if a "Java app loader" option is available on that menu. If it is not, then contact MOTOCODER support for advice on how to receive an enabled handset.
For more information about MIDway tool can be obtained through the MOTOCODER website (http://www.motocoder.com).

5
Application Management

The following sections describe the application management scheme for the Motorola C975. This chapter will discuss the following:

- Downloading a JAR without a JAD
- Installation and Deletion Status Reports
- DRM Content Protection in Java

Downloading a JAR without a JAD

In Motorola's MIDP 2.0 implementation, a JAR file can be downloaded without a JAD. In this case, the user clicks on a link for a JAR file, the file is downloaded, and a confirmation will be obtained before the installation begins. The information presented is obtained from the JAR manifest instead of the JAD.

Installation and Deletion Status Reports

The status (success or failure) of an installation, upgrade, or deletion of a MIDlet suite will be sent to the server according to the JSR 118 specification. If the status report cannot be sent, the MIDlet suite will still be enabled and the user will be allowed to use it. In some instances, if the status report cannot be sent, the MIDlet will be deleted by operator's request. Upon successful deletion, the handset will send the status code 912 to the MIDlet-Delete-Notify URL. If this notification fails, the MIDlet suite will still be deleted. If this notification cannot be sent due to lack of network connectivity, the notification will be sent at the next available network connection.
The following table presents the codes that are supported:

Code	Description
900	Success
901	Insufficient Memory
902	User Cancelled
903	Loss Of Service

904	JAR Size Mismatch
905	Attribute Mismatch
906	Invalid Descriptor
907	Invalid JAR
908	Incompatible Configuration or Profile
909	Application Authentication Failure
910	Application Authorization Failure
911	Push Registration Failure
912	Deletion Notification

DRM Content Protection in Java

Digital Rights Management (DRM) is a method to prevent MIDlets from distributing DRM content using any packet data network connection. In others words, DRM is a method of protecting content from illegal distribution by embedding the content into an encrypted package, along with rules dictating its use.

If the user has a set of keys and a valid license, then they are used for a specific file. A DRM application is required to decrypt the content for playback. This method will be transparent for the user, if he has a valid license.

The invalid license might happen because elapsed number of times the content to be executed/played, or elapsed validity for the license, or the content received through separate delivery.

For more information about this method, see at http://www.openmobilealliance.org.

6
Shared JAD URLs

Introduction

Actually, users are able to download J2ME applications. The first step is to download the JAD file and, after a confirmation, the site is launched to download the application. If they want to forward the JAD link to someone else, it's impossible.

The Share JAD URLs is a feature that resolves the prior problem, it allows users to share their downloaded J2ME application URLs with others. When J2ME applications are downloaded, the browser shall provide the Java Application Manager (JAM) with the JAD URL address. When J2ME applications are downloaded via PC or MMS, a new JAD attribute shall specify the JAD URL address.

Tell-A-Friend Option

When entering the J2ME application context-sensitive menu, a Tell-A-Friend option will be provided. Upon selecting this option, the standard SMS messaging form will appear. The link to the URL where the application JAD file can be found and its name will be pre-populated into the message body. This allows the user to send messages to friends, telling them where to download the application.

Upon receipt of a Tell-A-Friend message, a Motorola handset user should be able to use the browser's GOTO functionality. Selecting GOTO will cause the download of JAD to occur. The remaining download steps will occur as normal.

7
JAD Attributes

JAD / Manifest Attribute Implementations

The JAR manifest defines attributes to be used by the application management software (AMS) to identify and install the MIDlet suite. These attributes may or may not be found in the application descriptor.

The application descriptor is used, in conjunction with the JAR manifest, by the application management software to manage the MIDlet. The application descriptor is also used for the following:

- By the MIDlet for configuration specific attributes
- Allows the application management software on the handset to verify the MIDlet is suited to the handset before loading the JAR file
- Allows configuration-specific attributes (parameters) to be supplied to the MIDlet(s) without modifying the JAR file.

Motorola has implemented the following support for the MIDP 2.0 Java Application Descriptor attributes as outlined in the JSR-118. The table below lists all MIDlet attributes, descriptions, and its location in the JAD and/or JAR manifest that are supported in the Motorola implementation. Please note that the MIDlet will not install if the MIDlet-Data-Size is greater than 512k.

Attribute Name	Attribute Description	JAR Manifest	JAD
MIDlet-Name	The name of the MIDlet suite that identifies the MIDlets to the user	Yes	Yes
MIDlet-Version	The version number of the MIDlet suite	Yes	Yes
MIDlet-Vendor	The organization that provides the MIDlet suite.	Yes	Yes
MIDlet-Icon	The case-sensitive absolute name of a PNG file within the JAR used to represent the MIDlet suite.	Yes	Yes

MIDlet-Description	The description of the MIDlet suite.	No	No
MIDlet-Info-URL	A URL for information further describing the MIDlet suite.	Yes	No
MIDlet-<n>	The name, icon, and class of the nth MIDlet in the JAR file. Name is used to identify this MIDlet to the user. Icon is as stated above. Class is the name of the class extending the `javax.microedition.midlet.MIDletclass`.	Yes, or no if included in the JAD.	Yes, or no if included in the JAR Manifest.
MIDlet-Jar-URL	The URL from which the JAR file can be loaded.		Yes
MIDlet-Jar-Size	The number of bytes in the JAR file.		Yes
MIDlet-Data-Size	The minimum number of bytes of persistent data required by the MIDlet.	Yes	Yes
MicroEdition-Profile	The J2ME profiles required. If any of the profiles are not implemented the installation will fail.	Yes, or no if included in the JAD.	Yes, or no if included in the JAR Manifest.
MicroEdition-Configuration	The J2ME Configuration required, i.e. CLDC 1.0	Yes, or no if included in the JAD.	Yes, or no if included in the JAR Manifest.
MIDlet-Permissions	Zero or more permissions that are critical to the function of the MIDlet suite.	Yes	Yes
MIDlet-Permissions-Opt	Zero or more permissions that are non-critical to the function of the MIDlet suite.	Yes	Yes
MIDlet-Push-<n>	Register a MIDlet to handle inbound connections	Yes	Yes
MIDlet-Install-Notify	The URL to which a POST request is sent to report installation status of the MIDlet suite.	Yes	Yes
MIDlet-Delete-Notify	The URL to which a POST request is sent to report deletion of the MIDlet suite.	Yes	Yes
MIDlet-Delete-Confirm	A text message to be provided to the user when prompted to confirm deletion of the MIDlet suite.	Yes	Yes
FlipInsensitive	MIDlets with this Motorola specific attribute will enable the MIDlet to run with the flip closed.	Yes	Yes

Background	MIDlets with this Motorola specific attribute will continue to run when not in focus.	Yes	Yes

8
MIDlet Storage in Removable Memory

Introduction

Motorola J2ME enabled devices with removable memory like SD/MMC cards will be able to store, install and access J2ME applications from removable memory. This feature has some functionality that allows the user to install J2ME Applications on removable memory or phone memory once the application gets downloaded. The user also can to list and launch J2ME Applications stored on removable memory.
Other feature included is a mechanism to DRM protects J2ME Applications installed in secondary memory.

Installing downloaded applications into removable memory

A J2ME application may get downloaded via direct cable (USB). The installation procedure for downloaded applications onto phone/removable memory follows the below steps and rules:

- Initially, the user has a DOWNLOAD option that allows to choice between installing the application on removable memory or phone. The phone option should be the default option.
- There will be a separate directory within removable memory for J2ME Applications. All J2ME Applications stored on removable memory and information associated with them shall reside in this directory.
- An installed application on one device cannot be run on another device by swapping memory card. Separate installation is required for each device.
- Memory full condition handling while installing will be same for phone and removable memory.

- It is to provide push registry support for applications residing on removable memory. If this feature is not implemented, application that declare push registry usage in JAD file will not be allowed to be installed on removable memory.

Listing and Launch J2ME Applications from removable memory

By default, the JAM will list all installed applications on phone. The following rules will guide the user through of the available options:

- There will be a "Switch Storage Device" option under Games & Apps (MyJavaApps) menu allowing user to switch between storage devices (phone/removable memory) while listing applications. If user selects removable memory option, all installed applications on removable memory will get listed.
- Delete All Apps option under Java Settings menu will be effective for current storage device specified as above only. If phone is current storage device, only J2ME applications installed in phone will be deleted. The delete confirmation notice shall be modified to provide the current storage device information to the user. Delete all operation should only uninstall the application installed on removable memory. Original JAD and JAR files will not be deleted.
- Last menu item in the applications listing from removable memory shall be named [Install New]. If user selects this item, all application files (including already installed applications) from the J2ME directory in removable memory shall be listed. Both JAD and JAR files shall be listed. JAD and JAR file names will be preceded with a distinct icon to distinguish each type of file.
- User can select either JAD or JAR file if both are available. If user selects JAR file, implementation will search for JAD file with the same name in the same directory. If a JAD file is found and it refers to the selected JAR file, it shall be used. Otherwise JAR only installation shall be followed.
- Corresponding JAD and JAR files will be available in removable memory for installable applications requiring JAD file. JAD file should refer to the local JAR file only. Downloading of JAR files over the network for new application installation shall not be supported for this release.
- All externally loaded application JAD and JAR files will be kept in the J2ME directory in removable memory for JAM to list these as new applications for installation.
- Context sensitive menu of new applications should have "Delete" option to delete the application files. This operation should delete both JAD and JAR files (if present) permanently from removable memory.
- JAM will refresh application listing from removable device each time listing is required after a power cycle. This is to ensure that applications removed/added externally can be reflected upon each invocation.
- AMS will do an extra preverification step before launching applications from removable memory. This is to ensure that applications are not corrupted (externally or otherwise). If application is corrupted, a prompt shall be displayed after the user selects application to be launched.
- Delete, Details, Permissions options should be available under Games & Apps Menu for J2ME Applications installed on removable memory. Delete operation

should only uninstall the application installed on removable memory. Original JAD and JAR files will not be deleted from removable memory.

- AMS should support same application being installed on phone and removable memory. However duplicate applications will not be permitted on same storage device.
- If push registration is not supported for applications stored on removable memory, an IOException shall be returned if a J2ME application residing on removable memory tries to use javax.microedition.io.PushRegistry.registerConnection() method. The handling of this exception is left to the application.

9
iTAP

Intelligent Keypad Text Entry API

When users are using features such as SMS (short message service), or "Text Messaging", they can opt for a predictive text entry method from the handset. The J2ME environment has the ability to use SMS in its API listing. The use of a predictive entry method is a compelling feature to the MIDlet.

This API will enable a developer to access iTAP, Numeric, Symbol and Browse text entry methods. With previous J2ME products, the only method available was the standard use of TAP.

Predictive text entry allows a user to simply type in the letters of a word using only one key press per letter, as apposed to the TAP method that can require as many as four or more key presses. The use of the iTAP method can greatly decrease text-entry time. Its use extends beyond SMS text messaging, but into other functions such as phonebook entries.

The following J2ME text input components will support iTAP.

- javax.microedition.lcdui.TextBox

The TextBox class is a Screen that allows the user to edit and enter text.

- javax.microedition.lcdui.TextField

A TextField is an editable text component that will be placed into a Form. It is given a piece of text that is used as the initial value.

Refer to the table below for iTAP feature/class support for MIDP 2.0:

Feature/Class
Predictive text capability will be offered when the constraint is set to ANY User will be able to change the text input method during the input process when the constraint is set to ANY (if predictive text is available) Multi-tap input will be offered when the constraint on the text input is set to EMAILADDR, PASSWORD, or URL

10
Network APIs

Network Connections

The Motorola implementation of Networking APIs will support several network connections. The network connections necessary for Motorola implementation are the following:

- CommConnection for serial interface
- HTTP connection
- HTTPS connection
- Push registry
- SSL (secure socket)
- Datagram (UDP)

Refer to the table below for Network API feature/class support for MIDP 2.0:

Feature/Class	Implementation
All fields, methods, and inherited methods for the Connector class in the javax.microedition.io package	Supported
Mode parameter for the open () method in the Connector class the javax.microedition.io package	READ, WRITE, READ_WRITE
The timeouts parameter for the open () method in the Connector class of the javax.microedition.io package	Supported
HttpConnection interface in the javax.microedition.io package	Supported
HttpsConnection interface in the javax.microedition.io package	Supported
SecureConnection interface in the javax.microedition.io package	Supported
SecurityInfo interface in the javax.microedition.io package	Supported
UDPDDatagramConnection interface in the javax.microedition.io package	Supported
Connector class in the javax.microedition.io.package	Supported
PushRegistry class in the javax.microedition.io package	Supported

CommConnection interface in the javax.microedition.io package	Supported
Dynamic DNS allocation through DHCP	Supported
HttpConnection interface in the javax.microedition.io.package.	Supported
HttpsConnection interface in the javaxmicroedition.io.package	Supported
SecureConnection interface in the javax.microedition.io.package	Supported
SecurityInfo Interface in the javax.microedition.io.package	Supported
UDPDatagramConnection interface in the javax.microedition.io.package	Supported

The following is a code sample to show implementation of Socket Connection:

Socket Connection

```
import javax.microedition.io.*;
import java.io.*;
import javax.microedition.midlet.*;

….

         try {
              //open the connection and io streams
                    sc =
(SocketConnection)Connector.open("socket://www.myserver.com
:8080", Connector.READ_WRITE, true);
                    is = sc[i].openInputStream();
                    os = sc[i].openOutputStream();

            } catch (Exception ex) {
                closeAllStreams();
                System.out.println("Open Failed: " +
ex.getMessage());
             }
        }
        if (os != null && is != null)
        {
            try
            {
                os.write(someString.getBytes()); //write
some data to server

                int bytes_read = 0;
                int offset = 0;
                int bytes_left = BUFFER_SIZE;

          //read data from server until done
                do
                {
                    bytes_read = is.read(buffer, offset,
bytes_left);

                    if (bytes_read > 0)
```

```
                                {
                                    offset += bytes_read;
                                    bytes_left -= bytes_read;
                                 }
                            }
                            while (bytes_read > 0);

                        } catch (Exception ex) {
                                 System.out.println("IO failed: "+
ex.getMessage());
                        }
                        finally {
                            closeAllStreams(i); //clean up
                        }
                }
```

User Permission

The user of the handset will explicitly grant permission to add additional network connections.

Indicating a Connection to the User

When the java implementation makes any of the additional network connections, it will indicate to the user that the handset is actively interacting with the network. To indicate this connection, the network icon will appear on the handset's status bar as shown below.

Conversely, when the network connection is no longer used the network icon will be removed from the status bar.

If the handset supports applications that run when the flip is closed, the network icon on the external display will be activated when the application is in an active network

connection with the flip closed. Please note that this indication is done by the implementation.

HTTPS Connection

Motorola implementation supports a HTTPS connection on the Motorola C975 handset. Additional protocols that will be supported are the following:

- TLS protocol version 1.0 as defined in http://www.ietf.org/rfc/rfc2246.txt
- SSL protocol version 3.0 as defined in http://home.netscape.com/eng/ssl3/draft302.txt
- The following is a code sample to show implementation of HTTPS:

HTTPS

```
import javax.microedition.io.*;
import java.io.*;
import javax.microedition.midlet.*;
…

        try {
            hc[i] =
(HttpConnection)Connector.open("https://" + url[i] + "/");

        } catch (Exception ex) {
            hc[i] = null;
            System.out.println("Open Failed: " +
ex.getMessage());
        }

        if (hc[i] != null)
        {
            try {
                is[i] = hc[i].openInputStream();

                 byteCounts[i] = 0;
                 readLengths[i] = hc[i].getLength();

                 System.out.println("readLengths = " +
readLengths[i]);

                if (readLengths[i] == -1)
                {
                    readLengths[i] = BUFFER_SIZE;
                }

                int bytes_read = 0;
                int offset = 0;
                int bytes_left = (int)readLengths[i];
```

```
                do
                {
                    bytes_read = is[i].read(buffer, offset,
bytes_left);
                    offset += bytes_read;
                    bytes_left -= bytes_read;
                    byteCounts[i] += bytes_read;
                }
                while (bytes_read > 0);

                System.out.println("byte read = " +
byteCounts[i]);

                } catch (Exception ex) {
                  System.out.println("Downloading Failed: "+
ex.getMessage());
                  numPassed = 0;
                }
               finally {
                  try {
                            is[i].close();
                            is[i] = null;
                  } catch (Exception ex) {}
                }
        }
        /**
         *  close http connection
         */
        if (hc[i] != null)
        {
            try {
                hc[i].close();
            } catch (Exception ex) { }
            hc[i] = null;
        }
```

DNS IP

The DNS IP will be flexed on or off (per operator requirement) under Java Settings as read only or as user-editable. In some instances, it will be flexed with an operator-specified IP address.

Push Registry

The push registry mechanism allows an application to register for notification events that are meant for the application. The push registry maintains a list of inbound connections.

Mechanisms for Push

Motorola implementation for push requires the support of certain mechanisms. The mechanisms that will be supported for push are the following:

- SMS push: an SMS with a port number associated with an application used to deliver the push notification
-

The formats for registering any of the above mechanisms will follow those detailed in JSR 118 specification.

Push Registry Declaration

The application descriptor file will include information about static connections that are needed by the MIDlet suite. If all static push declarations in the application descriptor cannot be fulfilled during the installation, the MIDlet suite will not be installed. The user will be notified of any push registration conflicts despite the mechanism. This notification will accurately reflect the error that has occurred.

Push registration can fail as a result of an Invalid Descriptor. Syntax errors in the push attributes can cause a declaration error resulting in the MIDlet suite installation being cancelled. A declaration referencing a MIDlet class not listed in the MIDlet-<n> attributes of the same application descriptor will also result in an error and cancellation of the MIDlet installation.

Two types of registration mechanisms will be supported. The registration mechanisms to be supported are the following:

- Registration during installation through the JAD file entry using a fixed port number
- Dynamically register using an assigned port number
-

If the port number is not available on the handset, an installation failure notification will be displayed to the user while the error code 911 push is sent to the server. This error will cease the download of the application.

Applications that wish to register with a fixed port number will use the JAD file to identify the push parameters. The fixed port implementation will process the MIDlet-Push-n parameter through the JAD file.

The following is a code sample to show implementation of Push Registry:

<u>Push Registry Declaration</u>

```
import javax.microedition.lcdui.*;
import javax.microedition.midlet.*;
import javax.microedition.io.PushRegistry;

public class PushTest_1 extends MIDlet implements
CommandListener{

        public Display display;

        public static Form regForm;
        public static Form unregForm;
        public static Form mainForm;
        public static Form messageForm;

        public static Command exitCommand;
        public static Command backCommand;
        public static Command unregCommand;
        public static Command regCommand;

        public static TextField regConnection;
        public static TextField regFilter;
        public static ChoiceGroup registeredConnsCG;
        public static String[] registeredConns;

        public static Command mc;
        public static Displayable ms;

        public PushTest_1(){
            regConnection = new TextField("Connection
port:", "1000", 32, TextField.PHONENUMBER);
            regFilter = new TextField("Filter:", "*", 32,
TextField.ANY);

            display = Display.getDisplay(this);

            regForm = new Form("Register");
            unregForm = new Form("Unregister");
            mainForm = new Form("PushTest_1");
            messageForm = new Form("PushTest_1");

            exitCommand = new Command("Exit", Command.EXIT,
0);
            backCommand = new Command("Back", Command.BACK,
0);
            unregCommand = new Command("Unreg",
Command.ITEM, 1);
            regCommand = new Command("Reg", Command.ITEM,
1);

            mainForm.append("Press \"Reg\" softkey to
register a new connection.\n" +
                            "Press \"Unreg\" softkey to
unregister a connection.");
            mainForm.addCommand(exitCommand);
            mainForm.addCommand(unregCommand);
            mainForm.addCommand(regCommand);
```

```
                mainForm.setCommandListener(this);


                regForm.append(regConnection);
                regForm.append(regFilter);
                regForm.addCommand(regCommand);
                regForm.addCommand(backCommand);
                regForm.setCommandListener(this);

                unregForm.addCommand(backCommand);
                unregForm.addCommand(unregCommand);
                unregForm.setCommandListener(this);

                messageForm.addCommand(backCommand);
                messageForm.setCommandListener(this);


            }
        public void pauseApp(){}


        protected void startApp() {
            display.setCurrent(mainForm);
        }

        public void destroyApp(boolean unconditional) {
                notifyDestroyed();
        }

          public void showMessage(String s) {
                if(messageForm.size() != 0 )
messageForm.delete(0);
                messageForm.append(s);
                display.setCurrent(messageForm);
          }

        public void commandAction(Command c, Displayable s) {

                if((c == unregCommand) && (s == mainForm)){
                    mc = c;
                    ms = s;
                    new runThread().start();
                }

                if((c == regCommand) && (s == mainForm)){
                    display.setCurrent(regForm);
                }

                if((c == regCommand) && (s == regForm)){
                    mc = c;
                    ms = s;
                    new runThread().start();
                }

                if((c == unregCommand) && (s == unregForm)){
                    mc = c;
                    ms = s;
```

```
                    new runThread().start();
                }

                if((c == backCommand) && (s == unregForm )){
                    display.setCurrent(mainForm);
                }
                if((c == backCommand) && (s == regForm )){
                    display.setCurrent(mainForm);
                }

                if((c == backCommand) && (s == messageForm)){
                    display.setCurrent(mainForm);
                }

                if((c == exitCommand) && (s == mainForm)){
                    destroyApp(false);
                }

        }

        public class runThread extends Thread{
                public void run(){
                    if((mc == unregCommand) && (ms ==
mainForm)){
                        try{
                            registeredConns =
PushRegistry.listConnections(false);
                            if(unregForm.size() > 0)
unregForm.delete(0);
                            registeredConnsCG = new
ChoiceGroup("Connections", ChoiceGroup.MULTIPLE,
registeredConns, null);
                            if(registeredConnsCG.size() > 0)
unregForm.append(registeredConnsCG);
                            else unregForm.append("No
registered connections found.");
                            display.setCurrent(unregForm);
                        } catch (Exception e) {
                            showMessage("Unexpected " +
e.toString() + ": " + e.getMessage());
                        }

                    }

                    if((mc == regCommand) && (ms == regForm)){
                        try{

PushRegistry.registerConnection("sms://:" +
regConnection.getString(), "Receive",
regFilter.getString());
                            showMessage("Connection
successfully registered");
                        } catch (Exception e){
                            showMessage("Unexpected " +
e.toString() + ": " + e.getMessage());
                        }
                    }
```

```
                if((mc == unregCommand) && (ms ==
unregForm)){
                    try{
                        if(registeredConnsCG.size() > 0){
                            for(int i=0;
i<registeredConnsCG.size(); i++){

if(registeredConnsCG.isSelected(i)){

PushRegistry.unregisterConnection(registeredConnsCG.getStri
ng(i));

registeredConnsCG.delete(i);

if(registeredConnsCG.size() == 0){

unregForm.delete(0);

unregForm.append("No registered connections found.");
                                    }
                                }
                            }
                        }
                    } catch (Exception e) {
                        showMessage("Unexpected " +
e.toString() + ": " + e.getMessage());
                    }
                }
            }
        }
}
```

WakeUp.java

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import javax.microedition.io.PushRegistry;
import javax.microedition.rms.*;
import java.util.*;
import javax.microedition.io.*;

public class WakeUp extends MIDlet implements
CommandListener{

    public static Display display;
    public static Form mainForm;
    public static Command exitCommand;
    public static TextField tf;
    public static Command registerCommand;


    public void startApp() {

        display = Display.getDisplay(this);

        mainForm = new Form("WakeUp");
```

```
        exitCommand = new Command("Exit", Command.EXIT, 0);
        registerCommand = new Command("Register",
Command.SCREEN, 0);
        tf = new TextField("Delay in seconds", "10", 10,
TextField.NUMERIC);
        mainForm.addCommand(exitCommand);
        mainForm.addCommand(registerCommand);
        mainForm.append(tf);
        mainForm.setCommandListener(this);



        display.setCurrent(mainForm);

    }

    public void pauseApp() {
    }

    public void destroyApp(boolean unconditional) {
        notifyDestroyed();
    }

    public void commandAction(Command c, Displayable s) {
        if((c == exitCommand) && (s == mainForm)){
            destroyApp(false);
        }
        if(c == registerCommand){

            new regThread().start();

        }
    }

    public class regThread extends Thread{


        public void run(){

            try {
                long delay =
Integer.parseInt(tf.getString()) * 1000;

                long curTime = (new Date()).getTime();

                System.out.println(curTime + delay);

                PushRegistry.registerAlarm("WakeUp",
curTime + delay);
                mainForm.append("Alarm registered
successfully");

            } catch (NumberFormatException nfe) {
                mainForm.append("FAILED\nCan not decode
delay " + nfe);
            } catch (ClassNotFoundException cnfe) {
                mainForm.append("FAILED\nregisterAlarm
```

```
thrown " + cnfe);
            } catch (ConnectionNotFoundException cnfe) {
                mainForm.append("FAILED\nregisterAlarm
thrown " + cnfe);
            }
        }
    }
}
```

<u>SMS_send.java</u>

```
import javax.microedition.lcdui.*;
import javax.microedition.midlet.*;
import javax.microedition.io.PushRegistry;
import javax.wireless.messaging.*;
import javax.microedition.io.*;

public class SMS_send extends MIDlet implements
CommandListener{

        public Display display;

        public static Form messageForm;
        public static Form mainForm;

        public static Command exitCommand;
        public static Command backCommand;
        public static Command sendCommand;

        public static TextField address_tf;
        public static TextField port_tf;
        public static TextField message_text_tf;

        String[] binary_str = {"Send BINARY message"};
        public static ChoiceGroup binary_cg;

        byte[] binary_data = {1, 2, 3, 4, 5, 6, 7, 8, 9};
        String address;
        String text;

        MessageConnection conn = null;
        TextMessage txt_message = null;
        BinaryMessage bin_message = null;

        public SMS_send(){
            address_tf = new TextField("Address:", "", 32,
TextField.PHONENUMBER);
            port_tf = new TextField("Port:", "1000", 32,
TextField.PHONENUMBER);

            message_text_tf = new TextField("Message
text:", "test message", 160, TextField.ANY);
            binary_cg = new ChoiceGroup(null,
Choice.MULTIPLE, binary_str, null);
```

```
            display = Display.getDisplay(this);

            messageForm = new Form("SMS_send");
            mainForm = new Form("SMS_send");

            exitCommand = new Command("Exit", Command.EXIT,
0);
            backCommand = new Command("Back", Command.BACK,
0);
            sendCommand = new Command("Send", Command.ITEM,
1);

            mainForm.append(address_tf);
            mainForm.append(port_tf);
            mainForm.append(message_text_tf);
            mainForm.append(binary_cg);

            mainForm.addCommand(exitCommand);
            mainForm.addCommand(sendCommand);
            mainForm.setCommandListener(this);

            messageForm.addCommand(backCommand);
            messageForm.setCommandListener(this);

        }

      public void pauseApp(){
      }

      protected void startApp() {
          display.setCurrent(mainForm);
      }

      public void destroyApp(boolean unconditional) {
            notifyDestroyed();
      }

        public void showMessage(String s) {
            if(messageForm.size() != 0 )
messageForm.delete(0);
            messageForm.append(s);
            display.setCurrent(messageForm);
        }

      public void commandAction(Command c, Displayable s) {
            if((c == backCommand) && (s == messageForm)){
                display.setCurrent(mainForm);
            }
            if((c == exitCommand) && (s == mainForm)){
                destroyApp(false);
            }
            if((c == sendCommand) && (s == mainForm)){
                address = "sms://" +
address_tf.getString();
                if(port_tf.size() != 0) address += ":" +
port_tf.getString();
                text = message_text_tf.getString();
```

```
                new send_thread().start();
            }
        }

        public class send_thread extends Thread{
            public void run(){
                try{
                    conn = (MessageConnection)
Connector.open(address);
                    if(!binary_cg.isSelected(0)){
                        txt_message = (TextMessage)
conn.newMessage(MessageConnection.TEXT_MESSAGE);
                        txt_message.setPayloadText(text);
                        conn.send(txt_message);
                    } else {
                        bin_message = (BinaryMessage)
conn.newMessage(MessageConnection.BINARY_MESSAGE);

bin_message.setPayloadData(binary_data);
                        conn.send(bin_message);
                    }
                    conn.close();
                    showMessage("Message sent");
                } catch (Throwable t) {
                    showMessage("Unexpected " +
t.toString() + ": " + t.getMessage());
                }
            }
        }
}
```

Delivery of a Push Message

A push message intended for a MIDlet on the Motorola C975 handset will handle the following interactions:

- MIDlet running while receiving a push message – if the application receiving the push message is currently running, the application will consume the push message without user notification.
- No MIDlet suites running – if no MIDlets are running, the user will be notified of the incoming push message and will be given the option to run the intended application as shown below.

- Push registry with Alarm/Wake-up time for application – push registry supports one outstanding wake-up time per MIDlet in the current suite. An application will use the TimerTask notification of time-based events while the application is running.
- Another MIDlet suite is running during an incoming push – if another MIDlet is running, the user will be presented an option to launch the application that had registered for the push message. If the user selects the launch, the current MIDlet is terminated.
- Stacked push messages – it is possible for the handset to receive multiple push messages at one time while the user is running a MIDlet. The user will be given the option to allow the MIDlets to end and new MIDlets to begin. The user will be given the ability to read the messages in a stacked manner (stack of 5 supported), and if not read, the messages should be discarded.
- No applications registered for push – if there are no applications registered to handle this event, the incoming push message will be ignored.

Deleting an Application Registered for Push

If an application registered in the Push Registry is deleted, the corresponding push entry will be deleted, making the PORT number available for future Push Registrations.

Security for Push Registry

Push Registry is protected by the security framework. The MIDlet registered for the push should have the necessary permissions. Details on permissions are outlined in the Security chapter.

Network Access

Untrusted applications will use the normal HttpConnection and HttpsConnection APIs to access web and secure web services. There are no restrictions on web server port

numbers through these interfaces. The implementations augment the protocol so that web servers can identify untrusted applications. The following will be implemented:

- The implementation of HttpConnection and HttpsConnection will include a separate User-Agent header with the Product-Token "UNTRUSTED/1.0".User-Agent headers supplied by the application will not be deleted.
- The implementation of SocketConnection using TCP sockets will throw java.lang.SecurityException when an untrusted MIDlet suite attempts to connect on ports 80 and 8080 (http) and 443 (https).
- The implementation of SecureConnection using TCP sockets will throw java.lang.SecurityException when an untrusted MIDlet suites attempts to connect on port 443 (https).
- The implementation of the method DatagramConnection.send will throw java.lang.SecurityException when an untrusted MIDlet suite attempts to send datagrams to any of the ports 9200-9203 (WAP Gateway).
- The above requirements should be applied regardless of the API used to access the network. For example, the javax.microedition.io.Connector.open and javax.microedition.media.Manager.createPlayer methods should throw java.lang.SecurityException if access is attempted to these port numbers through a means other than the normal HttpConnection and HttpsConnection APIs.

11 Interface CommConnection

CommConnection

The CommConnection interface defines a logical serial port connection. A logical serial port connection is a logical connection through which bytes are transferred serially. This serial port is defined within the underlying operating system and may not correspond to a physical RS-232 serial port. For example, IrDA IRCOMM ports can be configured as a logical serial port within the operating system so it can act as a logical serial port.

Accessing

The comm port is accessed using a Generic Connection Framework string with an explicit port identifier and embedded configuration parameters, each separated with a semi-colon (;). Only one application may be connected to a particular serial port at a given time. A `java.io.IOException` is thrown if an attempt is made to open the serial port with `Connector.open()` if the connection is already open.

A URI with the type and parameters is used to open the connection. The scheme, as defined in RFC 2396, will be the following:

- `Comm.: <port identifier> [<optional parameters>]`

Parameters

The first parameter will be a port identifier, which is a logical device name. These port identifiers are device specific and should be used with care.

The valid identifiers for a particular device and OS can be queried through the `System.getproperty()` method using the key `microedition.commports`. A list of ports, separated by commas, is returned which can be combined with a `comm:` prefix as the URL string to open a serial port connection.

Any additional parameters will be separated by a semi-colon (;) without spaces. If a particular parameter is not applicable to a particular port, the parameter will be ignored. The port identifier cannot contain a semi-colon (;).

Legal parameters are defined by the definition of the parameters below. Illegal or unrecognized parameters cause an `IllegalArgumentException`. If the value of a parameter is supported by the device, it will be honored. If the value of a parameter is not supported, a `java.io.IOException` is thrown. If a `baudrate` parameter is requested, it is treated the same way that a `setBaudRate` method handles baud rates. For example, if the baudrate requested is not supported, the system will substitute a valid baudrate which can be discovered using the `getBaudRate` method.

The table below describes optional parameters.

Parameter	Default	Description
`baudrate`	platform dependent	The speed of the port.
`bitsperchar`	8	The number bits per character(7 or 8).
`stopbits`	1	The number of stop bits per char(1 or 2)
`parity`	none	The parity can be odd, even, or none.
`blocking`	on	If on, wait for a full buffer when reading.
`autocts`	on	If on, wait for the CTS line to be on before writing.
`autorts`	on	If on, turn on the RTS line when the input buffer is not full. If off, the RTS line is always on.

BNF Format for Connector.open () string

The URI will conform to the BNF syntax specified below. If the URI does not conform to this syntax, an `IllegalArgumentException` is thrown.

<comm_connection_string>	::= "**comm:**"<port_id>[<options_list>] ;
<port_id>	::= *string of alphanumeric characters*
<options_list>	::= *(<baud_rate_string>| <bitsperchar>| <stopbits>| <parity>| <blocking>| <autocts>| <autorts>) ; ; if an option duplicates a previous option in the ; option list, that option overrides the previous ; option
<baud_rate_string>	::= "**;baudrate=**"<baud_rate>
<baud_rate>	::= *string of digits*
<bitsperchar>	::= "**;bitsperchar=**"<bit_value>
<bit_value>	::= "7" | "8"
<stopbits>	::= "**;stopbits=**"<stop_value>
<stop_value>	::= "1" | "2"
<parity>	::= "**;parity=**"<parity_value>
<parity_value>	::= "even" | "odd" | "none"

<blocking>	::= ";**blocking**="<on_off>
<autocts>	::= ";**autocts**="<on_off>
<autorts>	::= ";**autorts**="<on_off>
<on_off>	::= "on" | "off"

Comm Security

Access to serial ports is restricted to prevent unauthorized transmission or reception of data. The security model applied to the serial port connection is defined in the implementing profile. The security model will be applied on the invocation of the `Connector.open ()` method with a valid serial port connection string. Should the application not be granted access to the serial port through the profile authorization scheme, a `java.lang.SecurityException` will be thrown from the `Connector.open ()` method. The security model will be applied during execution, specifically when the methods `openInputStream()`, `openDataInputStream()`, `openOutputStream()`, and `openDataOutputStream()` are invoked.

The following are code samples to implementation of CommConnection:

Sample of a CommConnection accessing a simple loopback program

```
CommConnection cc = (CommConnection)
            Connector.open("comm:com0;baudrate=19200");
 int baudrate = cc.getBaudRate();
 InputStream is  = cc.openInputStream();
 OutputStream os = cc.openOutputStream();
 int ch = 0;
 while(ch != 'Z') {
     os.write(ch);
     ch = is.read();
     ch++;
 }
 is.close();
 os.close();
 cc.close();
```

Sample of a CommConnection discovering available comm Ports

```
String port1;
 String ports =
System.getProperty("microedition.commports");
 int comma = ports.indexOf(',');
 if (comma > 0) {
     // Parse the first port from the available ports list.
     port1 = ports.substring(0, comma);
 } else {
     // Only one serial port available.
     port1 =ports;
```

```
                    }
```

Port Naming Convention

Logical port names can be defined to match platform naming conventions using any combination of alphanumeric characters. Ports will be named consistently among the implementations of this class according to a proposed convention. VM implementations will follow the following convention:

- Port names contain a text abbreviation indicating port capabilities followed by a sequential number for the port. The following device name types will be used:
    - COM# - COM is for RS-232 ports and # is a number assigned to the port
    - IR# - IR is for IrDA IRCOMM ports and # is a number assigned to the port

The naming scheme allows API users to determine the type of port to use. For example, if an application "beams" a piece of data, the application will look for IR# ports for opening the connection.

Method Summary

The tables below describe the CommConnection method summary for MIDP 2.0.

Method Summary	
Int	`getBaudRate()` Gets the baudrate for the serial port connection
Int	`setBaudRate (int baudrate)` Sets the baudrate for the serial port connection

12
JSR120 – Wireless Messaging API

Wireless Messaging API (WMA)

Motorola has implemented certain features that are defined in the Wireless Messaging API (WMA) 1.0 and 1.3 versions. The complete specification document is defined in JSR 120.

The JSR 120 specification states that developers can be provided access to send (MO – mobile originated) and receive (MT – mobile terminated) SMS (Short Message Service) on the target device.

A simple example of the WMA is the ability of two J2ME applications using SMS to communicate game moves running on the handsets. This can take the form of chess moves being passed between two players via the WMA.

Motorola in this implementation of the specification supports the following features.

- Creating an SMS
- Sending an SMS
- Receiving an SMS
- Viewing an SMS
- Deleting an SMS

SMS Client Mode and Server Mode Connection

The Wireless Messaging API is based on the Generic Connection Framework (GCF), which is defined in the CLDC specification 1.0. The use of the "Connection" framework, in Motorola's case is "`MessageConnection`".

The `MessageConnection` can be opened in either server or client mode. A server connection is opened by providing a URL that specifies an identifier (port number) for an application on the local device for incoming messages.

```
(MessageConnection)Connector.open("sms://:6000");
```

Messages received with this identifier will then be delivered to the application by this connection. A server mode connection can be used for both sending and receiving messages. A client mode connection is opened by providing a URL which points to another device. A client mode connection can only be used for sending messages.

```
(MessageConnection)Connector.open("sms://+441234567890
:6000");
```

SMS Port Numbers

When a port number is present in the address, the TP-User-Data of the SMS will contain a User-Data-Header with the application port addressing scheme information element. When the recipient address does not contain a port number, the TP-User-Data will not contain the application port addressing header. The J2ME MIDlet cannot receive this kind of message, but the SMS will be handled in the usual manner for a standard SMS to the device.

When a message identifying a port number is sent from a server type `MessageConnection`, the originating port number in the message is set to the port number of the `MessageConnection`. This allows the recipient to send a response to the message that will be received by this `MessageConnection`.

However, when a client type `MessageConnection` is used for sending a message with a port number, the originating port number is set to an implementation specific value and any possible messages received to this port number are not delivered to the `MessageConnection`. Please refer to the section A.4.0 and A.6.0 of the JSR 120.

When a MIDlet in server mode requests a port number (identifier) to use and it is the first MIDlet to request this identifier it will be allocated. If other applications apply for the same identifier then an `IOException` will be thrown when an attempt to open `MessageConnection` is made. If a system application is using this identifier, the MIDlet will not be allocated the identifier. The port numbers allowed for this request are restricted to SMS messages. In addition, a MIDlet is not allowed to send messages to certain restricted ports a `SecurityException` will be thrown if this is attempted.

JSR 120 Section A.6.0 Restricted Ports:
2805, 2923, 2948, 2949, 5502, 5503, 5508, 5511, 5512, 9200, 9201, 9203, 9207, 49996, 49999.

If you intend to use SMSC numbers then please review A.3.0 in the JSR 120 specification. The use of an SMSC would be used if the MIDlet had to determine what recipient number to use.

SMS Storing and Deleting Received Messages

When SMS messages are received by the MIDlet, they are removed from the SIM card memory where they were stored. The storage location (inbox) for the SMS messages has a capacity of up to thirty messages. If any messages are older than five days then this will be removed, from the inbox by way of a FIFO stack.

SMS Message Types

The types of messages that can be sent are TEXT or BINARY, the method of encoding the messages are defined in GSM 03.38 standard (Part 4 SMS Data Coding Scheme). Refer to section A.5.0 of JSR 120 for more information.

SMS Message Structure

The message structure of SMS will comply with GSM 03.40 v7.4.0 Digital cellular telecommunications system (Phase 2+); Technical realization of the Short Message Service (SMS) ETSI 2000.

Motorola's implementation uses the concatenation feature specified in sections 9.2.3.24.1 and 9.2.3.24.8 of the GSM 03.40 standard for messages that the Java application sends that are too long to fit in a single SMS protocol message.

This implementation automatically concatenates the received SMS protocol messages and passes the fully reassembled message to the application via the API. The implementation will support at least three SMS messages to be received and concatenated together. Also, for sending, support for a minimum of three messages is supported. Motorola advises that developers will not send messages that will take up more than three SMS protocol messages unless the recipient's device is known to support more.

SMS Notification

Examples of SMS interaction with a MIDlet would be the following:

- A MIDlet will handle an incoming SMS message if the MIDlet is registered to receive messages on the port (identifier) and is running.

- When a MIDlet is paused and is registered to receive messages on the port number of the incoming message, then the user will be queried to launch the MIDlet.
- If the MIDlet is not running and the Java Virtual Machine is not initialized, then a Push Registry will be used to initialize the Virtual Machine and launch the J2ME MIDlet. This only applies to trusted, signed MIDlets.
- If a message is received and the untrusted unsigned application and the KVM are not running then the message will be discarded.
- There is a SMS Access setting in the Java Settings menu option on the handset that allows the user to specify when and how often to ask for authorization. Before the connection is made from the MIDlet, the options available are:
    - Always ask for user authorization
    - Ask once per application
    - Never Ask

The following is a list of Messaging features/classes supported in the device.

Feature/Class	Implementation
JSR-120 API. Specifically, APIs defined in the javax.wireless.messaging package will be implemented with regards to the GSM SMS Adaptor	Supported
Removal of SMS messages	Supported
Terminated SMS removal – any user prompts handled by MIDlet	Supported
Originated SMS removal – any user prompts handled by MIDlet	Supported
All fields, methods, and inherited methods for the Connector Class in the javax.microedition.io package	Supported
All methods for the BinaryMessage interface in the javax.wireless.messaging package	Supported
All methods for the Message interface in the javax.wireless.messaging package	Supported
All fields, methods, and inherited methods for the MessageConnection interface in the javax.wireless.messaging package	Supported
Number of MessageConnection instances in the javax.wireless.messaging package	32 maximum
Number of MessageConnection instances in the javax.wireless.messaging package	16
All methods for the MessageListener interface in the javax.wireless.messaging package	Supported
All methods and inherited methods for the TextMessage interface in the javax.wireless.messaging package	Supported

16 bit reference number in concatenated messages	Supported
Number of concatenated messages.	30 messages in inbox, each can be concatenated from 3 parts. No limitation on outbox (immediately transmitted)
Allow MIDlets to obtain the SMSC address with the wireless.messaging.sms.smsc system property	Supported

The following are code samples to show implementation of the JSR 120 Wireless Messaging API:

<u>**Creation of client connection and for calling of method 'numberOfSegments' for Binary message:**</u>

```
BinaryMessage binMsg;
MessageConnection connClient;
int MsgLength = 140;

   /* Create connection for client mode */
   connClient = (MessageConnection) Connector.open("sms://"
+ outAddr);

   /* Create BinaryMessage for client mode */
   binMsg =
(BinaryMessage)connClient.newMessage(MessageConnection.BINAR
Y_MESSAGE);

   /* Create BINARY of 'size' bytes for BinaryMsg */
   public byte[] createBinary(int size) {
      int nextByte = 0;
byte[] newBin = new byte[size];

      for (int i = 0; i < size; i++) {
         nextByte = (rand.nextInt());
         newBin[i] = (byte)nextByte;
         if ((size > 4) && (i == size / 2)) {
            newBin[i-1] = 0x1b;
            newBin[i] = 0x7f;
         }
      }
      return newBin;
   }

byte[] newBin = createBinary(msgLength);
   binMsg.setPayloadData(newBin);

int num = connClient.numberOfSegments(binMsg);
```

Creation of server connection:

```
MessageConnection messageConnection =
(MessageConnection)Connector.open("sms://:9532");
```

Creation of client connection with port number:

```
MessageConnection messageConnection =
(MessageConnection)Connector.open("sms://+18473297274:9532")
;
```

Creation of client connection without port number:

```
MessageConnection messageConnection =
(MessageConnection)Connector.open("sms://+18473297274");
```

Closing of connection:

```
MessageConnection messageConnection.close();
```

Creation of SMS message:

```
Message textMessage =
messageConnection.newMessage(MessageConnection.TEXT_MESSAGE)
;
```

Setting of payload text for text message:

```
((TextMessage)message).setPayloadText("Text Message");
```

Getting of payload text of received text message:

```
receivedText =
((TextMessage)receivedMessage).getPayloadText();
```

Getting of payload data of received binary message:

```
BinaryMessage binMsg;
byte[] payloadData = binMsg.getPayloadData();
```

Setting of address with port number:

```
message.setAddress("sms://+18473297274:9532");
```

Setting of address without port number:

```
message.setAddress("sms://+18473297274");
```

Sending of message:

```
messageConnection.send(message);
```

Receiving of message:

```
Message receivedMessage = messageConnection.receive();
```

Getting of address:

```
String address = ((TextMessage)message).getAddress();
```

Getting of SMS service center address via calling of System.getProperty():

```
String addrSMSC =
System.getProperty("wireless.messaging.sms.smsc");
```

Getting of timestamp for the message:

```
Message message;
System.out.println("Timestamp: " +
message.getTimestamp().getTime());
```

Creation of client connection, creation of binary message, setting of payload for binary message and calling of method 'numberOfSegments(Message)' for Binary message:

```
BinaryMessage binMsg;
MessageConnection connClient;
int MsgLength = 140;

   /* Create connection for client mode */
   connClient = (MessageConnection) Connector.open("sms://"
+ outAddr);

   /* Create BinaryMessage for client mode */
   binMsg =
(BinaryMessage)connClient.newMessage(MessageConnection.BINAR
Y_MESSAGE);

   /* Create BINARY of 'size' bytes for BinaryMsg */
   public byte[] createBinary(int size) {
      int nextByte = 0;
byte[] newBin = new byte[size];

      for (int i = 0; i < size; i++) {
         nextByte = (rand.nextInt());
         newBin[i] = (byte)nextByte;
         if ((size > 4) && (i == size / 2)) {
            newBin[i-1] = 0x1b;
            newBin[i] = 0x7f;
         }
      }
      return newBin;
   }

byte[] newBin = createBinary(msgLength);
```

```
   binMsg.setPayloadData(newBin);

int num = connClient.numberOfSegments(binMsg);
```

<u>Setting of MessageListener and receiving of notifications about incoming messages:</u>

```
public class JSR120Sample1 extends MIDlet implements
CommandListener {
…
JSR120Sample1Listener listener = new
JSR120Sample1Listener();
…
// open connection
messageConnection =
(MessageConnection)Connector.open("sms://:9532");
…
// create message to send
…
listener.run();
…
// set payload for the message to send
…
// set address for the message to send
messageToSend.setAddress("sms://+18473297274:9532");
…
// send message (via invocation of 'send' method)
…
// set address for the message to receive
receivedMessage.setAddress("sms://:9532");
…
// receive message (via invocation of 'receive' method)
…

class JSR120Sample1Listener implements MessageListener,
Runnable {
private int messages = 0;

public void notifyIncomingMessage(MessageConnection
connection) {
System.out.println("Notification about incoming message
arrived");
   messages++;
}

public void run() {
      try {
      messageConnection.setMessageListener(listener);
      } catch (IOException e) {
         result = FAIL;
```

```
System.out.println("FAILED: exception while setting
listener: " + e.toString());
   }
}
}
```

App Inbox Clean-up

Actually, messages for MIDlets are stored in a separate App Inbox. This App Inbox is cleaned up automatically.

The App Inbox capacity is 26 messages or 26 segments and when a new message is received for a certain port number, and the App Inbox capacity has reached its limit of 26 messages, then the messages in the App Inbox will be deleted in the following order:

- If a certain port number has any unread messages in the App Inbox, then the oldest unread message in the buffer relative to that port number WILL be deleted next.
- If a certain port number currently has no messages in the App Inbox, then the oldest unread message in the buffer relative to all port numbers will be deleted next.

When the maximum number of messages is reached and the phone has reached memory full condition, no new messages can be received by the applications. A blinking messaging icon is used to inform the user that the messaging folder is full. At this stage the user has to manually delete some messages to clear some memory to allow the reception of incoming messages.

13
JSR 135 – Mobile Media API

JSR 135 Mobile Media API

The JSR 135 Mobile Media APIs feature sets are defined for five different types of media. The media defined is as follows:

- Tone Sequence
- Sampled Audio
- MIDI

The new implementation of JSR 135 supports playback of more audio formats and recording of time-based media – audio and video as well as still-image capture.

When a player is created for a particular type, it will follow the guidelines and control types listed in the sections outlined below.

The following is a code sample to show implementation of the JSR 135 Mobile Media API:

JSR 135

```
Player player;

// Create a media player, associate it with a stream
containing media data
try
{
    player =
Manager.createPlayer(getClass().getResourceAsStream("MP3.mp3
"), "audio/mp3");
}
catch (Exception e)
{
    System.out.println("FAILED: exception for createPlayer:
" + e.toString());
}

// Obtain the information required to acquire the media
resources
```

```
try
{
    player.realize();
}
catch (MediaException e)
{
    System.out.println("FAILED: exception for realize: " +
e.toString());
}

// Acquire exclusive resources, fill buffers with media data
try
{
    player.prefetch();
}
catch (MediaException e)
{
    System.out.println("FAILED: exception for prefetch: " +
e.toString());
}

// Start the media playback
try
{
    player.start();
}
catch (MediaException e)
{
    System.out.println("FAILED: exception for start: " +
e.toString());
}

// Pause the media playback
try
{
    player.stop();
}
catch (MediaException e)
{
    System.out.println("FAILED: exception for stop: " +
e.toString());
}

// Release the resources
player.close();
```

ToneControl

ToneControl is the interface to enable playback of a user-defined monotonic tone sequence. The JSR 135 Mobile Media API will implement public interface ToneControl.

A tone sequence is specified as a list of non-tone duration pairs and user-defined sequence blocks and is packaged as an array of bytes. The `setSequence()` method is used to input the sequence to the ToneControl.

The following is the available method for ToneControl:

`-setSequence (byte[] sequence)`: Sets the tone sequence

VolumeControl

VolumeControl is an interface for manipulating the audio volume of a Player. The JSR 135 Mobile Media API will implement public interface VolumeControl.

The following describes the different volume settings found within VolumeControl:

- Volume Settings - allows the output volume to be specified using an integer value that varies between 0 and 100. Depending on the application, this will need to be mapped to the volume level on the phone (0-7).
- Specifying Volume in the Level Scale - specifies volume in a linear scale. It ranges from 0 – 100 where 0 represents silence and 100 represents the highest volume available.
- Mute – setting mute on or off does not change the volume level returned by the getLevel. If mute is on, no audio signal is produced by the Player. If mute is off, an audio signal is produced and the volume is restored.

The following is a list of available methods with regards to VoumeControl:

`-getLevel`: Get the current volume setting.

`-isMuted`: Get the mute state of the signal associated with this VolumeControl.

`-setLevel (int level)`: Set the volume using a linear point scale with values between 0 and 100.

`-setMute (Boolean mute)`: Mute or unmute the Player associated with this VolumeControl.

StopTimeControl

StopTimeControl allows a specific preset sleep timer for a player. The JSR 135 Mobile Media API will implement public interface StopTimeControl.

The following is a list of available methods with regards to StopTimeControl:

`-getStopTime`: Gets the last value successfully by setStopTime.

`-setStopTime (long stopTime)`: Sets the media time at which you want the Player to stop.

Manager Class

Manager Class is the access point for obtaining system dependant resources such as players for multimedia processing. A Player is an object used to control and render media that is specific to the content type of the data. Manager provides access to an implementation specific mechanism for constructing Players. For convenience, Manager also provides a simplified method to generate simple tones. Primarily, the Multimedia API will provide a way to check available/supported content types.

Audio Media

The following multimedia file formats are supported:

File Type	CODEC
WAV	PCM
WAV	ADPCM
SP MIDI	General MIDI
MIDI Type 1	General MIDI
iMelody	IMelody
CTG	CTG
MP3	MPEG-1 layer III
AMR	AMR
BAS	General MIDI

The following is a list of audio MIME types supported:

Category	Description	MIME Type
Audio	iMelody	`audio/imelody x-imelody imy x-imy`
	MIDI	`audio/midi x-midi mid x-mid sp-midi`
	WAV	`audio/wav x-wav`
	MP3	`audio/mp3 x-mp3 mpeg3 x-mpeg3 mpeg x-mpeg`
	AMR/MP4	`audio/amr x-amr mp4 x-mp4`

Refer to the table below for multimedia feature/class support for JSR 135:

Feature/Class	Implementation
Media package found	Supported
Media control package	Supported
Media Protocol package	Streaming not supported
Control interface in javax.microedition.media	Supported
All methods for the Controllable interface in javax.microedition.media.control	Supported
All fields, methods, and inherited methods for the Player interface in javax.microedition.media	Supported
All fields and methods for the PlayerListener interface in javax.microedition.media	Supported
PlayerListener OEM event types for the PlayerListener interface	Standard types only
All fields, methods, and inherited methods for the Manager Class in javax.microedition.media	Supported
TONE_DEVICE_LOCATOR support in the Manager class of javax.microedition.media	Supported
TONE_DEVICE_LOCATOR content type will be audio/x-tone-seq	Supported
TONE_DEVICE_LOCATOR media locator will be device://tone	Supported
All constructors and inherited methods in javax.microedition.medi.MediaException	Supported
All fields and methods in the StopTimeControl interface in javax.microedition.media.control	Supported
All fields and methods in the ToneControl interface in javax.microedition.media.control	Supported
All methods in the VolumeControl interface in javax.microedition.media.control	Supported
Max volume of a MIDlet will not exceed the maximum speaker setting on the handset	Supported
Multiple SourceStreams for a DataSource	2

Note: The multimedia engine only supports prefetching 1 sound at a time, but 2 exceptions exist where 2 sounds can be prefetched at once. These exceptions are listed below:

1. Motorola provides the ability to play MIDI and WAV files simultaneously, but the MIDI track will be started first. The WAV file should have the following format: PCM 8,000 Khz; 8 Bit; Mono

2. When midi, iMelody, mix, and basetracks are involved, two instances of midi, iMelody, mix, or basetrack sessions can be prefetched at a time, although one of these instances has to be stopped. This is a strict requirement as (for example) two midi sounds cannot be played simultaneously.

Mobile Media Feature Sets

The following table lists the new packages, classes, fields and methods implemented for JSR 135. Appropriate exception shall be generated if the called method is not supported by the implementation. If a method is accessed without proper security permissions, security exception shall be thrown.

Packages	Classes	Methods	Comments & Requirements
javax.microedition.media.control	TempoControl (Applicable to MIDI/iMelody audio formats. Implementation guidance - SHOULD.)	setTempo()	Sets the current playback tempo. WILL implement a tempo range of 10 to 300 beats per minute.
		getTempo()	Gets the current playback tempo.
	PitchControl (Applicable to MIDI /iMelody audio formats. Implementation guidance - SHOULD)	getMaxPitch()	Gets the maximum playback pitch raise supported by the player. SHOULD implement a maximum playback pitch raise of 12,000 milli-semitones.
		getMinPitch()	Gets the minimum playback pitch raise supported by the player. SHOULD implement a minimum playback pitch raise of 12,000 millisemitones.
		getPitch()	Gets the current playback pitch raise.

		setPitch()	Sets the relative pitch raise.
	FramePositioningControl (Implementation guidance - SHOULD)	mapFrameToTime()	Converts the given frame number to the corresponding media time.
		mapTimeToFrame()	Converts the given media time to the corresponding frame number.
		seek()	Seeks to a given video frame.
		skip()	Skips a given number of frames from the current position.
	MIDIControl (Implementation guidance - SHOULD)	All fields & methods	
	RecordControl	All fields & methods	RecordControl controls the recording of media from a Player. Supports all methods. Required for audio capture functionality. Video capture support is optional. RecordControl is a protected API as specified in the Security section.

	VideoControl (Implementation guidance - SHOULD)	All fields & methods. getSnapshot() method **WILL** be supported if the VideoControl is implemented by an instance of camera device. If VideoControl is implemented by video player for video file/ stream playback, it is not mandatory to support get Snap Shot() method.	VideoControl controls the display of video. A Player which supports the playback of video WILL provide a VideoControl via its getControl and getControls methods.
	MetaDataControl	Implement all fields and methods. Support title, copyright, data, author keys for CODECs supporting these keys.	
javax.microedition.media	Player	All fields and methods	SHOULD allow a Player to use a different TimeBase other than its own. This is required for synchronization between multiple Media Players.
	PlayerListener	All fields and methods	SHOULD let applications register PlayerListener for receiving Player events.
	Manager	All fields and methods	WILL support file locator for local playback. For streaming, RTP locator needs to be supported. For camera, new device locator, “camera” has to be supported.
	TimeBase	getTime()	Gets the current time of this TimeBase.

javax.microedition.media.protocol	ContentDescriptor	getContentType()	Obtains a string that represents the content type for this descriptor.

There are others features that can be considered, such as:

- Support synchronous mixing of two or more sound channels. MIDI+WAV is supported, but MIDI+MP3 is highly desirable.
- The classes Manager, DataSource and RecordControl interface accepts media locators. In addition to normal playback locators specified by JSR – 135, the following special locators are supported:
  - **RTP Locators** are supported for streaming media on devices supporting real time streaming using RTSP. This support will be available for audio and video streaming through Manager (for playback media stream).
  - **HTTP Locators** are supported for playing back media over network connections. This support should be available through Manager implementation.
    e.g.: Manager.createPlayer("http://webserver/tune.mid")
  - **File locators** are supported for playback and capture of media. This is specific to Motorola J2ME implementations supporting file system API and not as per JSR-135. The support should be available through Manager and RecordControl implementations.
    e.g.: Manager.createPlayer("file://motorola/audio/sample.mid")
  - Capture Locator is supported for audio and video devices. A new device "camera" **will** be defined and supported for camera device. Manager.createPlayer() call shall return camera player as a special type of video player. Camera player should implement VideoControl and should support taking snapShots using VideoControl.getSnapshot() method.
    e.g.: Manager.createPlayer("capture://camera")

Supported Multimedia File Types

The following table lists multimedia file types (with corresponding CODECs) that are supported in products that are JSR-135 compliant. The common guideline being all codecs and file types supported by native side should be accessible through JSR-135 implementation. The implementation of JSR-135 (and these tables) is updated every time a new file type and/or CODEC is released.

Image Media

File Type	CODEC	Functionality
JPEG/JFIF	JPEG	Capture
JPEG/EXIF	JPEG	Capture

Audio Media

File Type	CODEC	Functionality
AAC	AAC	Playback
WMA	Proprietary (Microsoft)	Playback
AU	PCM Mu-law	Playback
AIFF	PCM	Playback
XMF	General MIDI	Playback
AMR	AMR NB	Playback and Capture

Video Media

File Type	CODEC	Functionality
MP4	H.263 (profile 0) or MPEG 4 with or without AMR/AAC audio.	Playback and Capture
3GP	H.263 (profile 0) or MPEG 4 with or without AMR/AAC audio.	Playback and Capture

RTF Streaming	RTP/RTSP/RTCP Streaming Engine	Playback
ASF	Proprietary (Microsoft)	Playback
WMV	Proprietary (Microsoft)	Playback
Windows Streaming	Proprietary (Microsoft)	Playback

Canned Sounds

The implementation supports predefined sounds which can be used by applications like games and alerts. Each predefined sound has an associated locator, which are used by the application.
The sounds and the specific locators are to be defined by CxD.

e.g.: Manager.createPlayer (CANNED_SOUND_ALARM)

Security

Mobile Media API follows MIDP 2.0 security model. Recording functionality APIs need to be protected. Trusted third party and untrusted applications will utilize user permissions. Specific permission settings are detailed below.

Policy

The following security policy will be flexed in per operator requirements at ship time of the handset.

Function Group	Trusted Third Party	Untrusted	Manufacturer	Operator
Multimedia Record	**Ask once Per App**, Always Ask, Never Ask, No Access	**Always Ask**, Ask once Per App, Never Ask, No Access	Full Access	Full Access

Permissions

The following table lists individual permissions within Multimedia Record function group.

Permission	Protocol	Function Group

javax.microedition. media.control. RecordControl.record	RecordControl.startRecord()	MultimediaRecord

14
JSR 139 – CLDC 1.1

JSR 139

CLDC 1.1 is an incremental release of CLDC version 1.0. CLDC 1.1 is fully backwards compatible with CLDC 1.0. Implementation of CLDC 1.1 supports the following:

- Floating Point
  - Data Types float and double
  - All floating point byte codes
  - New Data Type classes Float and Double
  - Library classes to handle floating point values
- Weak reference
- Classes Calender, Date and TimeZone are J2SE compliant
- Thread objects to be compliant with J2SE.

The support of thread objects to be compliant with J2SE requires the addition of `Thread.getName` and a few new contructors. The following table lists the additional classes, fields, and methods supported for CLDC 1.1 compliance:

	Classes	Additional Fields/Methods	Comments
System Classes	Java.lang.Thread	Thread (Runnable target, String name)	Allocates a new Thread object with the given target and name.
		Thread (String name)	Allocates a new Thread object with the given name
		String getName ()	Returns this thread's name

		Void interrupt ()	Interrupts this thread
	Java.lang.String	Boolean equalIgnoreCase (String anotherString)	Compares this string to another String, ignoring case considerations
		String intern ()	Returns a canonical representation for the string object
		Static String valueOf (float f)	Returns the string representation of the float argument
		Static String valueOf (double d)	Returns the string representation of the double argument
Data Type Classes	Java.lang.Float		New Class: Refer to CLDC Spec for more details
	Java.lang.Double		New Class: Refer to CLDC Spec for more details
Calender and Time Classes	Java.util.Calender	Protected int [ ] fields	The field values for the currently set time for this calendar
		Protected boolean { } is set	The flags which tell if a specified time field for the calendar is set
		Protected long time	The currently set time for this calendar, expressed in milliseconds after January 1, 1970, 0:00:00 GMT
		Protected abstract void ComputeFields	Converts the current millisecond time value to field values in fields [ ]
		Protected abstract void ComputeTime	Converts the current field values in fields [ ] to the millisecond time value time
	Java.lang.Date	String toString ()	Converts this date object to a String of the form: Dow mon dd hh:mm:ss

			zzz yyyy
Exception and Error Classes	Java.lang.NoClassDefFoundError		New Class: Refer to CLDC Spec for more details
Weak References	Java.lang.ref.Reference		New Class: Refer to CLDC Spec for more details
	Java.lang.ref.WeakReference		New Class: Refer to CLDC Spec for more details
Additional Utility Classes	Java.util.Random	Double nextDouble ()	Returns the nextpseudorandom, uniformly distributed double value between 0.0 and 1.0 from the random number generator's sequence
		Float nextFloat ()	Returns the next pseudorandom, uniformly distributed double value between 0.0 and 1.0 from the random number generator's sequence
		Int nextInt (int n)	Returns a pseudorandom, uniformly distributed int value between 0 (inclusive) and the specified value (exclusive), drawn from this random number generator's sequence
	Java.lang.Math	Static double E	The double value that is closer than any other to e, the base of the natural logarithms
		Static double PI	The double value that is closer than any other to pi, the ratio of the circumference of a circle to its diameter
		Static double abs (double a)	Returns the absolute value of a double value

		Static float abs (float a)	Returns the absolute value of a double value
		Static double ceil (double a)	Returns the smallest (closest to negative infinity) double value that is not less than the argument and is equal to a mathematical integer
		Static double cos (double a)	Returns the trigonometric cosine of an angle
		Static double floor (double a)	Returns the largest (closest to positive infinity) double value that is not greater than the argument and is equal to a mathematical integer.
		Static double max (double a, double b)	Returns the greater of two double values
		Static float max (float a, float b)	Returns the greater of two float values
		Static double min (float a, float b)	Returns the smaller of two double values
		Static float min (float a, float b)	Returns the smaller of two float values
		Static double sin (double a)	Returns the trigonometric sine of an angle
		Static double sqrt (double a)	Returns the correctly rounded positive square root of a double value
		Static double tan (double a)	Returns the trigonometric tangent of angle
		Static double todegrees (double angrad)	Converts an angle measured in radians to the equivalent angle measured in degrees
		Static double toradians (double angdeg)	Converts an angle measured in degrees to the equivalent angle

measured in radians

15
JSR 184 – 3D API

Overview

JSR 184 Mobile 3D API defines an API for rendering three-dimensional (3D) graphics at interactive frame rates, including a scene graph structure and a corresponding file format for efficient management and deployment of 3D content. Typical applications that might make use of JSR 184 Mobile 3D API include games, map visualizations, user interface, animated messages, and screen savers. JSR 184 requires a J2ME device supporting MIDP 2.0 and CLDC 1.1 at a minimum.

Mobile 3D API

The Motorola C975 contains full implementation of JSR 184 Mobile 3D API (http://jcp.org/en/jsr/detail?id=184). The Motorola C975 has also implemented the following:

- Call to `System.getProperty` with key – microedition.m3g.version will return 1.0, otherwise null will be returned.
- Floating point format for input and output is the standard IEEE float having a 8-bit exponent and a 24-bit mantissa normalized to 1.0, 2.0.
- Implementation will ensure the Object3D instances will be kept reference to reduce overhead and possible inconsistency.
- Thread safety
- Necessary pixel format conversions for rendering output onto device
- Support at least 10 animation tracks to be associated with an Object 3D instance (including animation controller) subject to dynamic memory availability.

Mobile 3D API File Format Support

The Motorola C975 supports both M3G and PNG file formats for loading 3D content. The C975 supports the standard .m3g and .png extensions for its file formats. Mime type and not extension will be used for identifying file type. In the case that the Mime type is not available, M3G files will be identified using the file identifier and PNG files using signature.

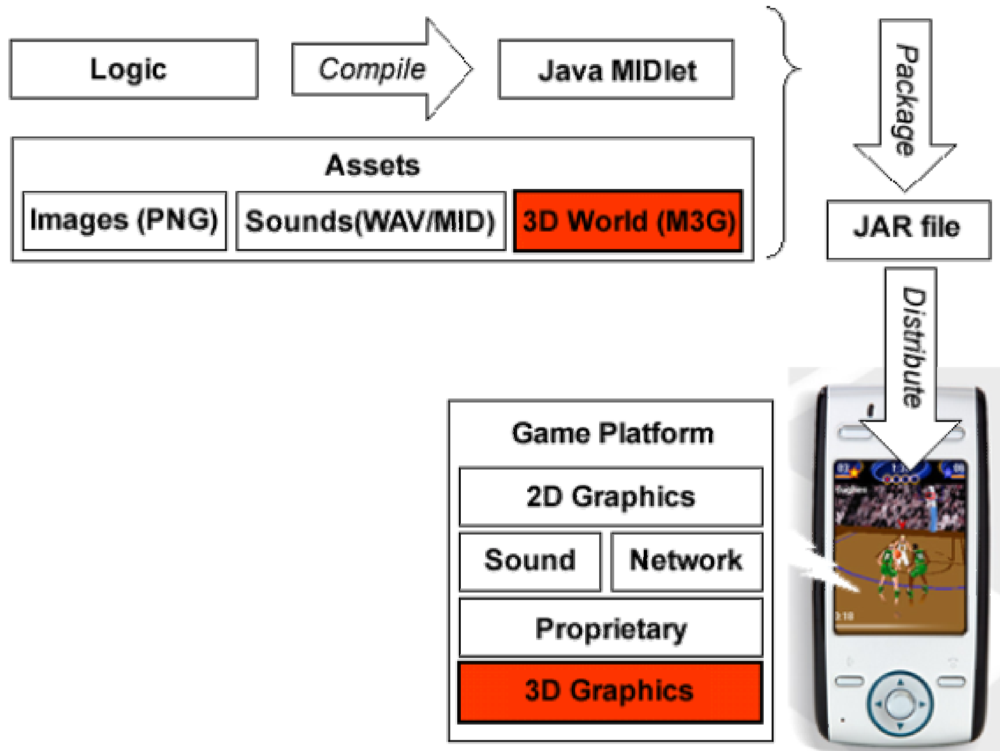
Mobile 3D Graphics – M3G API

The M3G API lets you access the realtime 3D engine embedded on the device, to create console quality 3D applications, such as games and menu systems. The main benefits of the M3G engine are the following:

- the whole 3D scene can be stored in a very small file size (typically 50-150K), allowing you to create games and applications in under 256K;
- the application can change the properties (such as position, rotation, scale, color and textures) of objects in the scene based on user interaction with the device;
- the application can switch between cameras to get different views onto the scene;
- the rendered images have a very high photorealistic quality.

Typical M3G Application

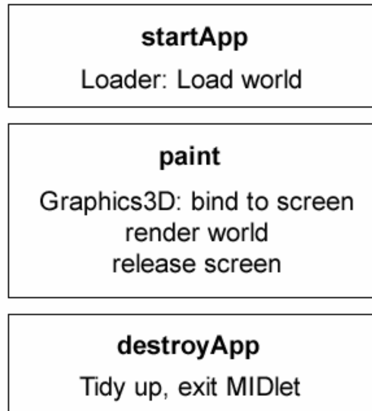
An application consists of logic that uses the M3G, MIDP 2.0 and CDLC 1.1 classes. The application is compiled into a Java MIDlet that can embedded on the target device. The MIDlet can also contain additional assets, such as one or more M3G files that define the 3D scene graph for the objects in the scene, images and sounds.



Most M3G applications use an M3G resource file that contains all the information required to define the 3D resources, such as objects, their appearance, lights, cameras and animations, in a scene graph. The file should be loaded into memory where object properties can be interrogated and altered using the M3G API. Alternatively all objects can be created from code, although this is likely to be slower and limits creativity for designers.

Simple MIDlets

The simplest application consists of an M3G file that is loaded into the application using the M3G Loader class, which is then passed to a Graphics3D object that renders the world to the Display.

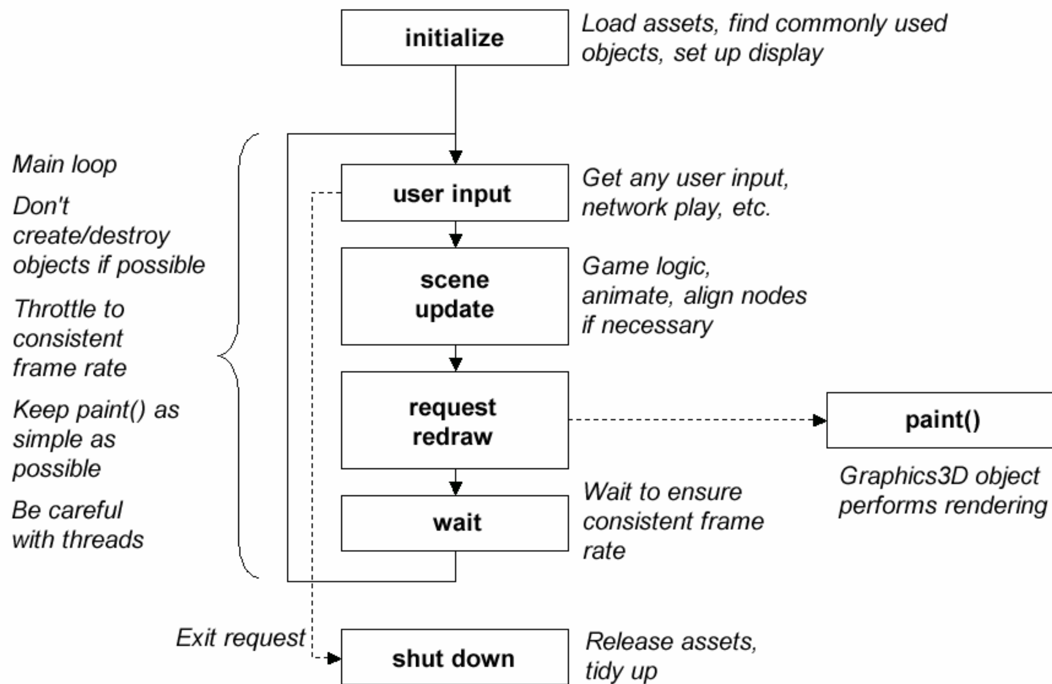


The World object contains the objects that define a complete 3D scene - geometry, textures, lights, cameras, and animations. The World object mediates access to the objects within the world. It can be passed as a block to the renderer, the Graphics3D class.

The Loader object, populates a World by loading an M3G file from a URI or other asset source, such as a buffer of bytes in M3G format. The Loader is not restricted to loading just Worlds, each file can contain as little as a single object and multiple files can be merged together on the device, or you can put everything into a single file.

The rendering class Graphics3D (by analogy to the MIDP Graphics class) takes a whole scene (or part of a scene graph), and renders a view onto that scene using the current camera and lighting setup. This view can be to the screen, to a MIDP image, or to a texture in the scene for special effects. You can pass a whole world in one go (retained mode) or you can pass individual objects (immediate mode). There is only one Graphics3D object present at one time, so that hardware accelerators can be used.

The following graphic shows the structure of a more typical MIDlet.



Initializing the world

The Loader class is used to initialize the world. It has two static methods: one takes in a byte array, while the other takes a named resource, such as a URI or an individual file in the JAR package.

The load methods return an array of Object3Ds that are the root level objects in the file.

The following example calls Loader.load() and passes it an M3G file from the JAR file using a property in the JAD file. Alternatively, you could specify a URI, for example:

```
Object3D[] roots =
Loader.load(http://www.example.com/m3g/simple.m3g)[0];
```

The example assumes that there is only one root node in the scene, which will be the world object. If the M3G file has multiple root nodes the code should be changed to reflect this, but generally most M3G files have a single root node.

```
public void startApp() throws MIDletStateChangeException
{
	myDisplay.setCurrent(myCanvas);

	try
	{
		// Load a file.
		Objects3D[] roots = Loader.load(getAppProperty("Content-
```

```
1"));

            // Assume the world is the first root node loaded.
            myWorld = (World) roots[0];
      }
      catch(Exception e)
      {
            e.printStackTrace();
      }

      // Force a repaint so the update loop is started.
      myCanvas.repaint();
}
```

Using the Graphics3D object

Using the Graphics3D is very straightforward. Get the Graphics3D instance, bind a target to it, render everything, and release the target.

```
public class myCanvas extends Canvas
{
      Graphics3D  myG3D = Graphics3D.getInstance();

      public void paint(Graphics g)
      {
            myG3D.bindTarget(g);
            try
            {
                  myG3D.render(myWorld);
            }
            finally
            {
                  myG3D.releaseTarget();
            }
      }
```

```
}
```

The finally block makes sure that the target is released and the Graphics3D can be reused. The bindTarget call should be outside the try block, as it can throw exceptions that will cause releaseTarget to be called when a target has not been bound, and releaseTarget throwing an exception.

Interrogating and interacting with objects

The World object is a container that sits at the top of the hierarchy of objects that form the scene graph. You can find particular objects within the scene very simply by calling find() with an ID. find() returns a reference to the object which has been assigned that ID in the authoring tool (or manually assigned from code). This is important because it largely makes the application logic independent of the detailed structure of the scene.

```
final int   PERSON_OBJECT_ID = 339929883;
Node  personNode = (Node)theWorld.find(PERSON_OBJECT_ID);
```

If you need to find many objects, or you don't have a fixed ID, then you can follow the hierarchy explicitly using the Object3D.getReferences() or Group.getChild() methods.

```
static void traverseDescendants(Object3D obj)
{
      int numReferences = obj.getReferences(null);

      if (numReferences > 0)
      {
            Object3D[] objArray = new Object3D[numReferences];
            obj.getReferences(objArray);

            for (int i = 0; i < numReferences; i++)
                  traverseDescendants(objArray[i]);
      }
}
```

Once you have an object, most of the properties on it can be modified using the M3G API. For example, you can change the position, size, orientation, color, brightness, or whatever other attribute of the object is important. You can also create and delete objects and insert them into the world, or link parts of other M3G files into the scene graph.

Animations

As well as controlling objects from code, scene designers can specify how objects should move under certain circumstances, and store this movement in "canned" or block animation sequences that can be triggered from code. Many object properties are animatable, including position, scale, orientation, color and textures. Each of these properties can be attached to a sequence of keyframes using an AnimationTrack. The keyframe sequence can be looped, or just played once, and they can be interpolated in several ways (stepwise, linear, spline).

A coherent action typically requires the simultaneous animation of several properties on several objects, the tracks are grouped together using the AnimationController object. This allows the application to control a whole animation from one place.

All the currently active animatable properties can be updated by calling animate() on the World. (You can also call this on individual objects if you need more control). The current time is passed through to animate(), and is used to determine the interpolated value to assign to the properties.

The animate() method returns a validity value that indicates how long the current value of a property is valid. Generally this is 0 which means that the object is still being animated and the property value is no longer valid, or infinity when the object is in a static state and does not need to be updated. If nothing is happening in the scene, you do not have to continually redraw the screen, reducing the processor load and extending battery life. Similarly, simple scenes on powerful hardware may run very fast; by restricting the frame-rate to something reasonable, you can extend battery life and are more friendly to background processes.

The animation subsystem has no memory, so time is completely arbitrary. This means that there are no events reported (for example, animation finished). The application is responsible for specifying when the animation is active and from which position in the keyframe sequence the animated property is played.

Consider a world myWorld that contains an animation of 2000 ms, that you want to cycle. First you need to set up the active interval for the animation, and set the position of the sequence to the start. Then call World.animate() with the current world time:

```
anim.setActiveInterval(worldTime, worldTime+2000);

anim.setPosition(0, worldTime);


int validity = myWorld.animate(worldTime);
```

Authoring M3G files

You can create all your M3G content from code if necessary but this is likely to be very time consuming and does not allow 3D artists and scene designers to easily create and rework visually compelling content with complex animations. You can use professional, visual development tools such as Swerve™ Studio or Swerve™ M3G exporter from Superscape Group plc, which export content from 3ds max, the industry standard 3D animation tool, in fully compliant M3G format. For more information please visit http://www.superscape.com/.

16 Phonebook Access API

Phonebook Access API

Using the Phonebook Access API, an application will be able to locate and update contact information on the handset. This contact information includes phone numbers, email addresses, and any other directory information related to individuals, groups, or organizations. The database used to store phonebook information will be unique and integrated for native phonebook, SIM card, and other applications using Phonebook API.

The primary goal of the Phonebook Access API is to be simple and thin to fit in resource-limited devices like the Motorola C975 handset. This API will specify a base storage class for all types of contacts items presented in the vCard specification (RFC2426 –vCard MIME Directory Profile – vCard 3.0 Specification). In addition, schema strings used in querying and storing contact information are those specified in the RFC2426 specification.

The Phonebook Access API will perform the following functions:

- Support multiple phonebook categories
- Allow multiple phone numbers and email addresses for each contact
- Store new entries
- Retrieve entries
- Edit existing entries
- Delete entries
- Check memory status
- Order and sort contact parameters
- Support standard schema strings
- Support recent calls information

The Motorola C975 also implements com.cmcc.phonebook, PhoneBookEntry and PhoneBook.

For more details about PhoneBookEntry and PhoneBook see CMCC GPRS Mobile Terminal Specification, sections 6.9.4.1 and 6.9.4.2.

Phonebook Access API Permissions

Prior to a MIDlet accessing the Phonebook API for all Phonebook operations, the implementation will check the Phonebook permissions under the Java Settings Menu. The phonebook permissions menu gives the user the following options:

- Always ask the user for authorization on all Phonebook access requests
- Ask the user for authorization once per application (Default setting)
- Never ask the user for authorization

The following are code samples to show implementation of the Phonebook API:

Sample of code to create object of PhoneBookRecord class:

```
PhoneBookRecord phbkRecEmpty = new PhoneBookRecord();

String name = “Name”;
String telNo = “9999999”;
int type = PhoneBookRecord.MAIN;
int categoryId = PhoneBookRecord.CATEGORY_GENERAL;

PhoneBookRecord phbkRec = new PhoneBookRecord(name, telNo,
type, categoryId);
```

Sample of code for calling of ‘add(int sortOrder)’ method:

```
int index = phbkRec.add(PhoneBookRecord.SORT_BY_NAME);
```

Sample of code for calling of ‘update(int index, int sortOrder)’ method:

```
phbkRec.type = PhoneBookRecord.HOME;
int newIndex = phbkRec.update(index,
PhoneBookRecord.SORT_BY_NAME);
```

Sample of code for calling of ‘delete(int index, int sortOrder)’ method:

```
PhoneBookRecord.delete(index, PhoneBookRecord.SORT_BY_NAME);
```

Sample of code for calling of ‘deleteAll()’ method:

```
PhoneBookRecord.deleteAll();
```

Sample of code for calling of ‘getRecord(int index, int sortOrder)’ method:

```
phbkRec.getRecord(index, PhoneBookRecord.SORT_BY_NAME);
```

Sample of code for calling of ‘findRecordByTelNo(String tel, int sortOrder)’ method:

```
index = phbkRec.findRecordByTelNo(telNo,
```

```
PhoneBookRecord.SORT_BY_NAME);
```

<u>Sample of code for calling of 'findRecordByName(char firstChar, int sortOrder)' method:</u>

```
index = PhoneBookRecord.findRecordByName('N',
PhoneBookRecord.SORT_BY_NAME);
```

<u>Sample of code for calling of 'findRecordByEmail(String email, int sortOrder)' method:</u>

```
String email = "email@mail.com";
index = phbkRec.findRecordByEmail(email,
PhoneBookRecord.SORT_BY_NAME);
```

<u>Sample of code for calling of 'getNumberRecords(int device)' method:</u>

```
// get total number of records
int numberRecsInPhone =
PhoneBookRecord.getNumberRecords(PhoneBookRecord.PHONE_MEMOR
Y);
int numberRecsInSim =
PhoneBookRecord.getNumberRecords(PhoneBookRecord
.SIM_MEMORY);
int numberRecsAll =
PhoneBookRecord.getNumberRecords(PhoneBookRecord.ALL_MEMORY)
;
```

<u>Sample of code for calling of 'getAvailableRecords(int device)' method:</u>

```
// get number of available records
int numberRecsAvalPhone =
PhoneBookRecord.getAvailableRecords(PhoneBookRecord.PHONE_ME
MORY);
int numberRecsAvalSim =

PhoneBookRecord.getAvailableRecords(PhoneBookRecord.SIM_MEMO
RY);
int numberRecsAvalAll =
PhoneBookRecord.getAvailableRecords(PhoneBookRecord.ALL_MEMO
RY);
```

<u>Sample of code for calling of 'getUsedRecords(int device, int sortOrder)' method:</u>

```
// get number of used records
int numberRecsUsedPhone =
PhoneBookRecord.getUsedRecords(PhoneBookRecord.PHONE MEMORY,
PhoneBookRecord.SORT_BY_NAME);
int numberRecsUsedSim =
PhoneBookRecord.getUsedRecords(PhoneBookRecord.SIM_MEMORY,
PhoneBookRecord.SORT_BY_NAME);
int numberRecsUsedAll =
PhoneBookRecord.getUsedRecords(PhoneBookRecord.ALL_MEMORY,
PhoneBookRecord.SORT_BY_NAME);
```

Sample of code for calling of 'getNumberRecordsByName(String name)' method:

```
int num = PhoneBookRecord.getNumberRecordsByName(name);
```

Sample of code for calling of 'getMaxNameLength(int device)' method:

```
int maxNameLengthPhone =
PhoneBookRecord.getMaxNameLength(PhoneBookRecord.PHONE_MEMOR
Y);
int maxNameLengthSim =
PhoneBookRecord.getMaxNameLength(PhoneBookRecord.SIM_MEMORY)
;
int maxNameLengthAll =
PhoneBookRecord.getMaxNameLength(PhoneBookRecord.ALL_MEMORY)
;
```

Sample of code for calling of 'getMaxTelNoLength (int device)' method:

```
int maxTelNoLengthPhone =
PhoneBookRecord.getMaxTelNoLength(PhoneBookRecord.PHONE_MEMO
RY);
int maxTelNoLengthSim =
PhoneBookRecord.getMaxTelNoLength(PhoneBookRecord.SIM_MEMORY
);
int maxTelNoLengthAll =
PhoneBookRecord.getMaxTelNoLength(PhoneBookRecord.ALL_MEMORY
);
```

Sample of code for calling of 'getMaxEmailLength ()' method:

```
int maxEmailLength =
PhoneBookRecord.getMaxEmailLength();
```

Sample of code for calling of 'getIndexBySpeedNo(int speedNo, int sortOrder)' method:

```
int speedNo = 1;
index = PhoneBookRecord.getIndexBySpeedNo(speedNo,
PhoneBookRecord.SORT_BY_NAME);
```

Sample of code for calling of 'getNewSpeedNo(int num, int device)' method:

```
int speedNo = 1;
int speedNo_phone =
PhoneBookRecord.getNewSpeedNo(speedNo,
PhoneBookRecord.PHONE_MEMORY);
int speedNo_sim =
PhoneBookRecord.getNewSpeedNo(speedNo,
 PhoneBookRecord.PHONE_MEMORY);
int speedNo_all =
PhoneBookRecord.getNewSpeedNo(speedNo,
PhoneBookRecord.PHONE_MEMORY);
```

Sample of code for calling of ‘getDeviceType(int speedNo)’ method:

```
int speedNo = 1;
int type = PhoneBookRecord.getDeviceType(speedNo);
```

Sample of code for calling of ‘setPrimary(int index, int sortOrder)’ method:

```
int index = 1;
PhoneBookRecord.setPrimary(index,
PhoneBookRecord.SORT_BY_NAME);
```

Sample of code for calling of ‘resetPrimary(int index, int sortOrder)’ method:

```
int index = 1;
PhoneBookRecord.resetPrimary(index,
PhoneBookRecord.SORT_BY_NAME);
```

Sample of code for calling of ‘isPrimary(int speedNo)’ method:

```
int speedNo = 1;
boolean res = PhoneBookRecord.isPrimary(speedNo);
```

Sample of code for calling of ‘fromVFormat(InputStream in, int device)’ method:

```
buffer = new String("BEGIN:VCARD\r\nN:;" + new String(name)
+
"\r\nTEL;TYPE=WORK:1\r\nEND:VCARD\r\n");
int num =
PhoneBookRecord.fromVFormat((InputStream)(new
ByteArrayInputStream(buffer.getBytes())),
PhoneBookRecord.PHONE_MEMORY);

Sample of code for calling of ‘toVFormat(OutputStream out,
int index, int outFormat, int sortOrder)’ method:

int index = 1;
ByteArrayOutputStream outStream = new
ByteArrayOutputStream();
PhoneBookRecord.toVFormat(outStream, index,
PhoneBookRecord.VCARD_3_0,
PhoneBookRecord.SORT_BY_NAME);


System.out.println("***** Contents of the output stream:
*****");
System.out.print(new String(outStream.toByteArray()));
```

Sample of code for calling of ‘createMailingList(int[] members, int sortOrder)’ method:

```
PhoneBookRecord mailingList = new PhoneBookRecord();
int mlSpeedNumbers[] = new int[2];
mlSpeedNumbers[0] = 1;
mlSpeedNumbers[1] = 2;
```

```
mailingList.name = "MList";
mailingList.type = PhoneBookRecord.MAILING_LIST;
mailingList.speedNo =
 PhoneBookRecord.getNewSpeedNo(1,
PhoneBookRecord.PHONE_MEMORY);

index = mailingList.createMailingList(mlSpeedNumbers,
PhoneBookRecord.SORT_BY_NAME);
```

Sample of code for calling of 'addMailingListMember(int mlSpeedNo, int mbSpeedNo)' method:

```
int mlspeedNo = 3, mbspeedNo = 4;
PhoneBookRecord.addMailingListMember(mlspeedNo, mbspeedNo);
```

Sample of code for calling of 'deleteMailingListMember(int mlSpeedNo, int mbSpeedNo)' method:

```
int mlspeedNo = 3, mbspeedNo = 4;
PhoneBookRecord.deleteMailingListMember(mlspeedNo,
mbspeedNo);
```

Sample of code for calling of 'getMailingListMembers(int speedNo)' method:

```
int mlspeedNo = 3;
int[] returnArray =
PhoneBookRecord.getMailingListMembers(mlspeedNo);
```

Sample of code for calling of 'isMailingListMember(int mlSpeedNo, int mbSpeedNo)' method:

```
boolean returnValue = false;
int mlspeedNo = 3, mbspeedNo = 4;
returnValue = PhoneBookRecord.isMailingListMember(mlspeedNo,
mbspeedNo);
```

Sample of code for calling of 'getNumberMailingListMembers(int speedNo)' method:

```
int numberMembers, mlspeedNo = 3;
numberMembers =
PhoneBookRecord.getNumberMailingListMembers(mlspeedNo);
```

Sample of code for calling of 'addCategory(String name)' method:

```
String categoryName = "CatName";
int categoryId = PhoneBookRecord.addCategory(categoryName);
```

Sample of code for calling of 'deleteCategory(int categoryId)' method:

```
PhoneBookRecord.deleteCategory(categoryId);
```

Sample of code for calling of 'getCategoryName(int categoryId)' method:

```
String categoryName =
PhoneBookRecord.getCategoryName(categoryId);
```

Sample of code for calling of 'getCategoryMembers(int categoryId)' method:

```
int SpeedNumbersArray[] = null;
SpeedNumbersArray =
PhoneBookRecord.getCategoryMembers(categoryId);
```

Sample of code for calling of 'getNumberCategoryMembers (int categoryId)' method:

```
int numberMembers =
PhoneBookRecord.getNumberCategoryMembers(categoryId);
```

Sample of code for calling of 'getNumberCategories()' method:

```
int numberCategories =
PhoneBookRecord.getNumberCategories();
```

Sample of code for calling of 'getCategoryIdByIndex(int index)' method:

```
int index = 1;
int categoryId =
PhoneBookRecord.getCategoryIdByIndex(index);
```

Sample of code for calling of 'getMaxCategoryNameLength()' method:

```
int maxCategoryNameLength =
PhoneBookRecord.getMaxCategoryNameLength();
```

Sample of code for calling of 'getCurrentCategoryView()' method:

```
int categoryView = PhoneBookRecord.getCurrentCategoryView();
```

Sample of code for calling of 'setCategoryView()' method:

```
int oldCategoryView =
PhoneBookRecord.setCategoryView(categoryId);
```

Sample of code to create object of RecentCallDialed class:

```
String name = "Name";
String telNo = "9999999";
int type = RecentCallRecord.VOICE;
int attribute = RecentCallRecord.CALL_CONNECTED;
long time = 10000;
int duration = 3000;
boolean show_id = true;

RecentCallDialed dialedRecentCall = new
RecentCallDialed(name, telNo, type, attribute, time,
duration, show_id);
```

Sample of code for calling of 'add()' method:

```
String name = “Name”;
String telNo = “9999999”;
int type = RecentCallRecord.VOICE;
int attribute = RecentCallRecord.CALL_CONNECTED;
long time = 10000;
int duration = 3000;
boolean show_id = true;

RecentCallDialed dialedRecord = new RecentCallDialed(name,
telNo, type, attribute, time, duration, show_id);
dialedRecord.add();
```

Sample of code for calling of ‘delete(int index)’ method:

```
int index = 1;
RecentCallDialed.delete(1);
```

Sample of code for calling of ‘deleteAll()’ method:

```
RecentCallDialed.deleteAll();
```

Sample of code for calling of ‘getRecord(int index)’ method:

```
int index = 1;
dialedRecord.getRecord(1);
```

Sample of code for calling of ‘getUsedRecords()’ method:

```
int usedRecs = RecentCallDialed.getUsedRecords();
```

Sample of code for calling of ‘getNumberRecords()’ method:

```
int numberRecs = RecentCallDialed.getNumberRecords();
```

Sample of code for calling of ‘getMaxNameLength()’ method:

```
int maxNameLength = RecentCallDialed.getMaxNameLength();
```

Sample of code for calling of ‘getMaxTelNoLength()’ method:

```
int maxTelNoLength = RecentCallDialed.getMaxTelNoLength();
```

Sample of code to create object of RecentCallReceived class:

```
String name = “Name”;
String telNo = “9999999”;
int type = RecentCallRecord.VOICE;
int attribute = RecentCallRecord.CALL_CONNECTED;
long time = 10000;
int duration = 3000;
int cli_type  = RecentCallReceived.CALLER_ID_NAME;
```

```
RecentCallReceived receivedRecentCall = new
RecentCallReceived (name, telNo, type, attribute, time,
duration, cli_type);
```

<u>Sample of code for calling of 'add()' method:</u>

```
String name = "Name";
String telNo = "9999999";
int type = RecentCallRecord.VOICE;
int attribute = RecentCallRecord.CALL_CONNECTED;
long time = 10000;
int duration = 3000;
int cli_type  = RecentCallReceived.CALLER_ID_NAME;

RecentCallReceived receivedRecord = new
RecentCallReceived(name, telNo, type, attribute, time,
duration, show_id);
receivedRecord.add();
```

<u>Sample of code for calling of 'delete(int index)' method:</u>

```
int index = 1;
RecentCallReceived.delete(1);
```

<u>Sample of code for calling of 'deleteAll()' method:</u>

```
RecentCallReceived.deleteAll();
```

<u>Sample of code for calling of 'getRecord(int index)' method:</u>

```
int index = 1;
receivedRecord.getRecord(1);
```

<u>Sample of code for calling of 'getUsedRecords()' method:</u>

```
int usedRecs = RecentCallReceived.getUsedRecords();
```

<u>Sample of code for calling of 'getNumberRecords()' method:</u>

```
int numberRecs = RecentCallReceived.getNumberRecords();
```

<u>Sample of code for calling of 'getMaxNameLength()' method:</u>

```
int maxNameLength = RecentCallReceived.getMaxNameLength();
```

<u>Sample of code for calling of 'getMaxTelNoLength()' method:</u>

```
int maxTelNoLength = RecentCallReceived.getMaxTelNoLength();
```

17
Telephony API

The Telephony API allows a MIDlet to make a voice call, however, the user needs to confirm the action before any voice call is made. The reason for the confirmation is to provide a measure of security from rogue applications overtaking the handset.

Unlike standard TAPI, the wireless Telephony API provide simple function and simple even listener: startCall (), send ExtNo(), and endCall (), DialerListener.

The tables below describe the Interface and Class Summary:

Interface Summary	
DialerListener	The DialerListener interface provides a mechanism for the application to be notified of phone call event.

Class Summary	
Dialer	The Dialer class defines the basic functionality for start and end phone call.
DialerEvent	The DialerEvent class defines phone call events.

Dialer Class

The dialer Class can be used to start and end a phone call and user listener to receive an event. The applications use a Dialer to make a phone call and use DialerListener to receive event.

Class DialerEvent

The DialerEvent class defines phone call events. The table below describes the Field Summary:

Summary	
static byte	PHONE_VOICECALL_CONNECT Phone call was connected event
static byte	PHONE_VOICECALL_DISCONNECT

	Phone call was disconnected event
static byte	PHONE_VOICECALL_DTMF_FAILURE Send extension number error event
static byte	PHONE_VOICECALL_FAILURE start phone call error event
static byte	PHONE_VOICECALL_HOLD Current java phone call was held by native phone event
static byte	PHONE_VOICECALL_INIT Phone start dial-up event
static byte	PHONE_VOICECALL_TIMEOUT Phone process timeout event
static byte	PHONE_VOICECALL_UNHOLD Current java phone call was unheld event

The table below describes the Constructor Summary:

Constructor Summary
DialerEvent()

The following methods are inherited from class java.lang.Object:

- equals
- getClass
- hashCode
- notify
- notifyAll
- toString
- wait

The following table describes the Field Details:

Field	Detail	Definition
PHONE_VOICECALL_INIT	public static final byte **PHONE_VOICECALL_INIT**	Phone start dial-up event
PHONE_VOICECALL_FAILURE	public static final byte **PHONE_VOICECALL_FAILURE**	Start phone call error event
PHONE_VOICECALL_CONNECT	public static final byte **PHONE_VOICECALL_CONNECT**	Phone call was connected event
PHONE_VOICECALL_DISCONNECT	public static final byte	Phone call

	PHONE_VOICECALL_DISCONNECT	was disconnected event
PHONE_VOICECALL_TIMEOUT	public static final byte **PHONE_VOICECALL_TIMEOUT**	Phone process timeout event
PHONE_VOICECALL_HOLD	public static final byte **PHONE_VOICECALL_HOLD**	Current java phone call was held by native phone event
PHONE_VOICECALL_UNHOLD	public static final byte **PHONE_VOICECALL_UNHOLD**	Current java phone call was unheld event
PHONE_VOICECALL_DTMF_FAILURE	public static final byte **PHONE_VOICECALL_DTMF_FAILURE**	Send extension number error event

Class Dialer

The Dialer class defines the basic functionality for starting and ending a phone call. The table below describes the Method Summary:

Method Summary	
void	**endCall**() end or cancel a phone call
static Dialer	**getDefaultDialer**()
void	**sendExtNo**(String extNumber) send extension number.
void	**setDialerListener**(DialerListener listener) Registers a DialerListener object.
void	**startCall**(String telenumber) start a phone call using given telephone number.
void	**startCall**(String teleNumber, String extNo)

	start a phone call using given telephone number and extension number.

The following methods are inherited from java.lang.object:

- equals
- getClass
- hashCode
- notify
- notifyAll
- toString
- wait

getDefaultDialer

public static Dialer **getDefaultDialer**()

Get a `Dialer` instance.

setDialerListener

public void **setDialerListener**(`DialerListener` listener)

Registers a `DialerListener` object.

The platform will notify this listener object when a phone event has been received to this `Dialer object`.

There can be at most one listener object registered for a `Dialer` object at any given point in time. Setting a new listener will implicitly de-register the possibly previously set listener.

Passing `null` as the parameter de-registers the currently registered listener, if any.

Parameters:

`listener` - `DialerListener` object to be registered. If `null`, the possibly currently registered listener will be de-registered and will not receive phone call event.

startCall

public void **startCall**(`String` telenumber)

throws `IOException`

start a phone call using given telephone number.

Parameters:

`telenumber` - the telephone number to be call.

`extNo` - the extension number to be send.

Throws:

`IOException` - if the call could not be created or because of network failure

`NullPointerException` - if the parameter is null

`SecurityException` - if the application does not have permission to start the call

startCall

public void **startCall**(`String` teleNumber,
        `String` extNo)
    throws `IOException`

start a phone call using given telephone number and extension number.

Parameters:

`telenumber` - the telephone number to be call.

`extNo` - the extension number to be send.

Throws:

`IOException` - if the call could not be created or because of network failure

`NullPointerException` - if the parameter is null

`SecurityException` - if the application does not have permission to start the call

sendExtNo

public void **sendExtNo**(`String` extNumber)
    throws `IOException`

send extension number.

Parameters:

`sendExtNo` - the extension number to be send.

Throws:

`IOException` - if the extension could not be send or because of network failure

endCall

public void **endCall**()

throws `IOException`

end or cancel a phone call

Throws:

`IOException` - if the call could not be end or cancel.

Interface DialerListener

public interface **DialerListener**

The `DialerListener` interface provides a mechanism for the application to be notified of phone call event.

When an event arrives, the `notifyDialerEvent()` method is called

The listener mechanism allows applications to receive TAPI voice call event without needing to have a listener thread

If multiple event arrive very closely together in time, the implementation has calling this listener in serial.

Sample DialerListener Implementation

Dialer listener program

```
import java.io.IOException;
 import javax.microedition.midlet.*;
 import javax.microedition.io.*;
 import com.motorola.*;

 public class Example extends MIDlet implements DialerListener {
     Dialer dialer;

     // Initial tests setup and execution.

     public void startApp() {
       try {
         dialer = Dialer.getDefaultDialer();

         // Register a listener for inbound TAPI voice call events.
         dialer.setDialerListener(this);
         dialer.startCall("01065642288");

       } catch (IOException e) {
```

```
      // Handle startup errors
    }
  }
```

<u>Asynchronous callback for receive phone call event</u>

```
  public void notifyDialerEvent(Dialer dialer, byte event) {
    switch (event) {
      case DialerEvent.PHONE_VOICECALL_INIT:
        // your process
          break;
      case DialerEvent.PHONE_VOICECALL_FAILURE:
        // your process
          break;
      case DialerEvent.PHONE_VOICECALL_CONNECT:
        // your process
          break;
      case DialerEvent.PHONE_VOICECALL_DISCONNECT:
        // your process
          break;
      case DialerEvent.PHONE_VOICECALL_TIMEOUT:
        // your process
          break;
      case DialerEvent.PHONE_VOICECALL_HOLD:
        // your process
          break;
      case DialerEvent.PHONE_VOICECALL_UNHOLD:
        // your process
          break;
      case DialerEvent.PHONE_VOICECALL_DTMF_FAILURE:
        // your process
          break;
    }
  }

  // Required MIDlet method - release the connection and
  // signal the reader thread to terminate.

  public void pauseApp() {
    try {
      dialer.endCall();
    } catch (IOException e) {
      // Handle errors
    }
  }


  // Required MIDlet method - shutdown.
  // @param unconditional forced shutdown flag

  public void destroyApp(boolean unconditional) {
    try {
      dialer.setDialerListener(null);
      dialer.endCall();
    } catch (IOException e) {
      // Handle shutdown errors.
```

```
        }
      }
  }
```

notifyDialerEvent

public void **notifyDialerEvent**(`Dialer` dialer,

byte event)

Called by the platform when a phone call event was received by a `Dialer` object where the application has registered this listener object.

This method is called once for each TAPI voice call event to the `Dialer` object.

NOTE: The implementation of this method MUST return quickly and MUST NOT perform any extensive operations. The application SHOULD NOT receive and handle the message during this method call. Instead, it should act only as a trigger to start the activity in the application's own thread.

Parameters:

`dialer` - the `Dialer` where the TAPI voice call event has arrived

`event` - the TAPI voice call event type.

Class Hierarchy

- class java.lang.**Object**
  - class com.motorola.phone.**Dialer**
  - class com.motorola.phone.**DialerEvent**

Interface Hierarchy

- interface com.motorola.phone.**DialerListener**

18
JSR 185 - JTWI

JTWI specifies a set of services to develop highly portable, interoperable Java applications. JTWI reduces API fragmentation and broadens the number of applications for mobile phones.

Overview

Any Motorola device implementing JTWI will support the following minimum hardware requirements in addition to the minimum requirements specified in MIDP 2.0:

- At least a 125 x 125 pixels screen size as returned by full screen mode Canvas.getHeight () and Canvas.getWidth ()
- At least a color depth of 4096 colors (12-bit) as returned by Display.numColors ()
- Pixel shape of 1:1 ratio
- At least a Java Heap Size of 512 KB
- Sound mixer with at least 2 sounds
- At least a JAD file size of 5 KB
- At least a JAR file size of 64 KB
- At least a RMS data size of 30 KB

Any Motorola JTWI device will implement the following and pass the corresponding TCK:

- CLDC 1.0 or CLDC 1.1
- MIDP 2.0 (JSR 118)
- Wireless Messaging API 1.1 (JSR 120)
- Mobile Media API 1.1 (JSR 135)

CLDC related content for JTWI

JTWI is designed to be implemented on top of CLDC 1.0 or CLDC 1.1. The configuration provides the VM and the basic APIs of the application environment. If floating point capabilities are exposed to Java Applications, CLDC 1.1 will be implemented.

The following CLDC requirements will be supported:

- Minimum Application thread count will allow a MIDlet suite to create a minimum of 10 simultaneous running threads
- Minimum Clock Resolution – The method java.land.System.currentTimeMillis () will record the elapsed time in increments not to exceed 40 msec. At least 80% of test attemps will meet the time elapsed requirement to achieve acceptable conformance.
- Names for Encodings will support at least the preferred MIME name as defined by IANA (http://www.iana.org/assignments/character-sets) for the supported character encodings. If not preferred name has been defined, the registered name will be used (i.e UTF-16).
- Character Properties will provide support for character properties and case conversions for the characters in the Basic Latin and Latin-1 Supplement blocks of Unicode 3.0. Other Unicode character blocks will be supported as necessary.
- Unicode Version will support Unicode characters. Character information is based on the Unicode Standard version 3.0. Since the full character tables required for Unicode support can be excessively large for devices with tight memory budgets, by default, the character property and case conversion facilities in CLDC assume the presence of ISO Latin-1 range of characters only. Refer to JSR 185 for more information.
- Custom Time Zone Ids will permit use of custom time zones which adhere to the following time xone format:
    - General Time Zone: For time zones representing a GMT offset value, the following syntax is used:
        - Custom ID:
            - GMT Sign Hours: Minutes
            - GMT Sign Hours Minutes
            - GMT Sign Hours Hours
        - Sign: one of:
            - + -
        - Hours:
            - Digit
            - Digit Digit
        - Minutes:
            - Digit Digit

- Digit: one of:
    - 0 1 2 3 4 5 6 7 8 9

NOTE: Hours will be between 0 and 23, and minutes will be between 00 and 50. For example, GMT +10 and GMT +0010 equates to ten hours and ten minutes ahead of GMT.

When creating a TimeZone, the specified custom time zone ID is normalized in the following syntax:

- NormalizedCustomID:
    - GMT Sign TwoDigitHours: Minutes
    - Sign: one of:
        - + -
    - TwoDigitHours:
        - Digit Digit
    - Minutes:
        - Digit Digit
    - Digit: one of:
        - 0 1 2 3 4 5 6 7 8 9

MIDP 2.0 specific information for JTWI

MIDP 2.0 provides the library support for user interface, persistent storage, networking, security, and push functions. MIDP 2.0 contains a number of optional functions, some of which will be implemented as outlined below. The following JTWI requirements for MIDP 2.0 will be supported:

- Record Store Minimum will permit a MIDlet suite to create at least 5 independent RecordStores. This requirement does not intend to mandate that memory be reserved for these Record Stores, but it will be possible to create the RecordStores if the required memory is available.
- HTTP Support for Media Content will provide support for HTTP 1.1 for all supported media types. HTTP 1.1 conformance will match the MIDP 2.0 specification. See package.javax.microedition.io for specific requirements.
- JPEG for Image Objects – ISO/IEC JPEG together wil JFIF will be supported. The support for ISO/IEC JPEG only applies to baseline DCT, non-differential, Huffman coding, as defined in JSR 185 JTWI specification, symbol 'SOF0'. This support extends to the class javax.microedition.lcdui.Image, including the methods outlined above. This mandate is voided in the event that the JPEG image format becomes encumbered with licensing requirements.
- Timer Resolution will permit an application to specify the values for the firstTime, delay, and period parameters of java.util.timer.schedule () methods with a distinguishable resolution of no more than 40 ms. Various factors (such as

garbage collection) affect the ability to achieve this requirement. At least 80% of test attempts will meet the schedule resolution requirement to achieve acceptable conformance.

- Minimum Number of Timers will allow a MIDlet to create a minimum of 5 simultaneously running Timers. This requirement is independent of the minimum specified by the Minimum Application Thread Count.
- Bitmap Minimums will support the loading of PNG images with pixel color depths of 1, 2, 4, 8, 16, 24, and 32 bits per pixel per the PNG format specification. For each of these color depths, as well as for JFIF image formats, a compliant implementation will support images up to 76800 total pixels.
- TextField and TextBox and Phonebook Coupling – when the center select key is pressed while in a TextBox or TextField and the constraint of the TextBox or TextField is TextField.PHONENUMBER, the names in the Phonebook will be displayed in the "Insert Phonenumber?" screen.
- Supported characters in TextField and TextBox – TextBox and TextField with input constraint TextField.ANY will support inputting all the characters listed in JSR 185.
- Supported characters in EMAILADDR and URL Fields – Class javax.microedition.lcdui.TextBox and javax.microedition.lcdui.TextField with either of the constraints TextField.EMAILADDR or TextField.URL will allow the same characters to be input as are allowed for input constraint TextField.ANY
- Push Registry Alarm Events will implement alarm-based push registry entries.
- Identification of JTWI via system property – to identify a compliant device and the implemented version of this specification, the value of the system property microedition.jtwi.version will be 1.0

Wireless Messaging API 1.1 (JSR 120) specific content for JTWI

WMA defines an API used to send and receive short messages. The API provides access to network-specific short message services such as GSM SMS or CDMA short messaging. JTWI will support the following as it is outlined in the JSR 120 chapter of this developer guide:

- Support for SMS in GSM devices
- Cell Broadcast Service in GSM devices
- SMS Push

Mobile Media API 1.1 (JSR 135) specific content for JTWI

The following will be supported for JTWI compliance:

- HTTP 1.1 Protocol will be supported for media file download for all supported media formats
- MIDI feature set specified in MMAPI (JSR 135) will be implemented. MIDI file playback will be supported.
- VolumeControl will be implemented and is required for controlling the colume of MIDI file playback.
- JPEG encoding in video snapshots will be supported if the handset supports the video feature set and video image capture.
- Tone sequence file format will be supported. Tone sequences provide an additional simple format for supporting the audio needs of many types of games and other applications.

MIDP 2.0 Security specific content for JTWI

- The Motorola C975 follows the security policy outlined in the Security chapter of this developer guide.

19
MIDP 2.0 Security Model

The following sections describe the MIDP 2.0 Default Security Model for the Motorola C975 handset. The chapter discusses the following topics:

- Untrusted MIDlet suites and domains
- Trusted MIDlet suites and domains
- Permissions
- Certificates

For a detailed MIDP 2.0 Security process diagram, refer to the Motocoder website (http://www.motocoder.com).

Refer to the table below for the default security feature/class support for MIDP 2.0:

Feature/Class	Implementation
All methods for the Certificate interface in the javax.microedition.pki package	Supported
All fields, constructors, methods, and inherited methods for the CertificateException class in the javax.microedition.pki package	Supported
MIDlet-Certificate attribute in the JAD	Supported
A MIDlet suite will be authenticated as stated in Trusted MIDletSuites using X.509 of MIDP 2.0 minus all root certificates processes and references	Supported
Verification of SHA-1 signatures with a MD5 message digest algorithm	Supported
Only one signature in the MIDlet-Jar-RSA-SHA1 attribute	Supported
All methods for the Certificate interface in the javax.microedition.pki package	Supported
All fields, constructors, methods, and inherited methods for the CertificateException class in the javax.microedition.pki package	Supported
Will preload two self authorizing Certificates	Supported
All constructors, methods, and inherited methods for the MIDletStateChangeException class in the javax.microedition.midlet	Supported

package	
All constructors and inherited methods for the MIDletStateChangeException class in the javax.microedition.midlet package	Supported

Please note the domain configuration is selected upon agreement with the operator.

Untrusted MIDlet Suites

A MIDlet suite is untrusted when the origin or integrity of the JAR file cannot be trusted by the device.

The following are conditions of untrusted MIDlet suites:

- If errors occur in the process of verifying if a MIDlet suite is trusted, then the MIDlet suite will be rejected.
- Untrusted MIDlet suites will execute in the untrusted domain where access to protected APIs or functions is not allowed or allowed with explicit confirmation from the user.

Untrusted Domain

Any MIDlet suites that are unsigned will belong to the untrusted domain. Untrusted domains handsets will allow, without explicit confirmation, untrusted MIDlet suites access to the following APIs:

- `javax.microedition.rms` – RMS APIs
- `javax.microedition.midlet` – MIDlet Lifecycle APIs
- `javax.microedition.lcdui` – User Interface APIs
- `javax.microedition.lcdui.game` – Gaming APIs
- `javax.microedition.media` – Multimedia APIs for sound playback
- `javax.microedition.media.control` – Multimedia APIs for sound playback

The untrusted domain will allow, with explicit user confirmation, untrusted MIDlet suites access to the following protected APIs or functions:

- `javax.microedition.io.HttpConnection` – HTTP protocol
- `javax.microedition.io.HttpsConnection` – HTTPS protocol

Trusted MIDlet Suites

Trusted MIDlet suites are MIDlet suites in which the integrity of the JAR file can be authenticated and trusted by the device, and bound to a protection domain. The Motorola C975 will use x.509PKI for signing and verifying trusted MIDlet suites.

Security for trusted MIDlet suites will utilize protection domains. Protection domains define permissions that will be granted to the MIDlet suite in that particular domain. A MIDlet suite will belong to one protection domain and its defined permissible actions. For implementation on the Motorola C975, the following protection domains should exist:

- Manufacturer – permissions will be marked as "Allowed" (Full Access). Downloaded and authenticated manufacturer MIDlet suites will perform consistently with MIDlet suites pre-installed by the manufacturer.
- Operator – permissions will be marked as "Allowed" (Full Access). Downloaded and authenticated operator MIDlet suites will perform consistently with other MIDlet suites installed by the operator.
- 3rd Party – permissions will be marked as "User". User interaction is required for permission to be granted. MIDlets do not need to be aware of the security policy except for security exceptions that will occur when accessing APIs.
- Untrusted – all MIDlet suites that are unsigned will belong to this domain.

Permissions within the above domains will authorize access to the protected APIs or functions. These domains will consist of a set of "Allowed" and "User" permissions that will be granted to the MIDlet suite.

Permission Types concerning the Handset

A protection domain will consist of a set of permissions. Each permission will be "Allowed" or "User", not both. The following is the description of these sets of permissions as they relate to the handset:

- "Allowed" (Full Access) permissions are any permissions that explicitly allow access to a given protected API or function from a protected domain. Allowed permissions will not require any user interaction.
- "User" permissions are any permissions that require a prompt to be given to the user and explicit user confirmation in order to allow the MIDlet suite access to the protected API or function.

User Permission Interaction Mode

User permission for the Motorola C975 handsets is designed to allow the user the ability to either deny or grant access to the protected API or function using the following interaction modes (bolded term(s) is prompt displayed to the user):

- blanket – grants access to the protected API or function every time it is required by the MIDlet suite until the MIDlet suite is uninstalled or the permission is changed by the user. (Never Ask)

- session – grants access to the protected API or function every time it is required by the MIDlet suite until the MIDlet suite is terminated. This mode will prompt the user on or before the final invocation of the protected API or function. (Ask Once Per App)
- oneshot – will prompt the user each time the protected API or function is requested by the MIDlet suite. (Always Ask)
- No – will not allow the MIDlet suite access to the requested API or function that is protected. (No Access)
- The prompt No, Ask Later will be displayed during runtime dialogs and will enable the user to not allow the protected function to be accessed this instance, but to ask the user again the next time the protected function is called.
-

User permission interaction modes will be determined by the security policy and device implementation. User permission will have a default interaction mode and a set of other available interaction modes. The user should be presented with a choice of available interaction modes, including the ability to deny access to the protected API or function. The user will make their decision based on the user-friendly description of the requested permissions provided for them.

The Permissions menu allows the user to configure permission settings for each MIDlet when the VM is not running. This menu is synchronized with available run-time options.

Implementation based on Recommended Security Policy

The required trust model, the supported domain, and their corresponding structure will be contained in the default security policy for Motorola's implementation for MIDP 2.0. Permissions will be defined for MIDlets relating to their domain. User permission types, as well as user prompts and notifications, will also be defined.

Trusted 3rd Party Domain

A trusted third party protection domain root certificate is used to verify third party MIDlet suites. These root certificates will be mapped to a location on the handset that cannot be modified by the user. The storage of trusted third party protection domain root certificates and operator protection domain root certificates in the handset is limited to 12 certificates.

If a certificate is not available on the handset, the third party protection domain root certificates will be disabled. The user will have the ability to disable root certificates through the browser menu and will be prompted to warn them of the consequences of disabling root certificates. These third party root certificates will not be used to verify downloaded MIDlet suites.

The user will be able to enable any disabled trusted third party protection domain root certificates. If disabled, the third party domain will no longer be associated with this certificate. Permissions for trusted third party domain will be "User" permissions; specifically user interaction is required in order for permissions to be granted.

The following table shows the specific wording to be used in the first line of the above prompt:

Protected Functionality	Top Line of Prompt	Right Softkey
Data Network	Use data network?	OK
Messaging	Use messaging?	OK
App Auto-Start	Launch <MIDlet names>?	OK
Connectivity Options	Make a local connection?	OK
User Data Read Capability	Read phonebook data?	OK
User Data Write Capability	Modify phonebook data?	OK
App Data Sharing	Share data between apps?	OK

The radio button messages will appear as follows and mapped to the permission types as shown in the table below:

MIDP 2.0 Permission Types	Runtime Dialogs	UI Permission Prompts
Oneshot	Yes, Always Ask	Always Ask
Session	Yes, Ask Once	Ask Once per App
Blanket	Yes, Always Grant Access	Never Ask
no access	No, Never Grant Access	No, Access

The above runtime dialog prompts will not be displayed when the protected function is set to "Allowed" (or full access), or if that permission type is an option for that protected function according to the security policy table flexed in the handset.

Security Policy for Protection Domains

The following table lists the security policy by function groups for each domain. Under each domain are the settings allowed for that function within the given domain, while the bolded setting is the default setting. The Function Group is what will be displayed to the user when access is requested and when modifying the permissions in the menu. The default setting is the setting that is effective at the time the MIDlet suite is first invoked and remains in effect until the user changes it.

Permissions can be implicitly granted or not granted to a MIDlet based on the configuration of the domain the MIDlet is bound to. Specific permissions cannot be defined for this closed class. A MIDlet has either been developed or not been developed

to utilize this capability. The other settings are options the user is able to change from the default setting.

Function Group	Trusted Third Party	Untrusted	Manufacturer	Operator
Data Network	**Ask Once Per App**, Always Ask, Never Ask, No Access	**Always Ask**, Ask Once Per App, No Access	Full Access	Full Access
Messaging	**Always Ask**, No Access	**Always Ask**, No Access	Full Access	Full Access
App Auto-Start	**Ask Once Per App**, Always Ask, Never Ask, No Access	**Ask Once Per App**, Always Ask, No Access	Full Access	Full Access
Connectivity Options	**Ask Once Per App**, Always Ask, Never Ask, No Access	**Ask Once Per App**, Always Ask, Never Ask, No Access	Full Access	Full Access
User Data Read Capability	**Always Ask**, Ask Once Per App, Never Ask, No Access	**No Access**	Full Access	Full Access
User Data Write Capability	**Always Ask**, Ask Once Per App, Never Ask, No Access	**No Access**	Full Access	Full Access
Multimedia Recording	**Ask Once Per App**, Always Ask, Never Ask, No Access	**No Access**	Full Access	Full Access

The table below shows individual permissions assigned to the function groups shown in the table above.

MIDP 2.0 Specific Functions		
Permission	**Protocol**	**Function Group**
javax.microedition.io.Connector.http	http	Data Network
javax.microedition.io.Connector.https	https	Data Network
javax.microedition.io.Connector.datagram	Datagram	Data Network
javax.microedition.io.Connector.datagramreceiver	datagram server (w/o host)	Data Network

javax.microedition.io.Connector.socket	Socket	Data Network
javax.microedition.io.Connector.serversocket	server socket (w/ o host)	Data Network
javax.microedition.io.Connector.ssl	Ssl	Data Network
javax.microedition.io.Connector.comm	Comm.	Connectivity Options
javax.microedition.io.PushRegistry	All	App Auto-Start
Phonebook API		
com.motorola.phonebook.readaccess	PhoneBookRecord.findRecordByName() PhoneBookRecord.findRecordByTelNo() PhoneBookRecord.findRecordByEmail() PhoneBookRecord.getNumberRecordsByName() PhoneBookRecord.getRecord() PhoneBookRecord.toVFormat() PhoneBookRecord.getCategoryName() PhoneBookRecord.getMailingListMembers() RecentCallDialed.getRecord() RecentCallReceived.getRecord()	User Data Read Capability
com.motorola.phonebook.writeaccess	PhoneBookRecord.add() PhoneBookRecord.update() PhoneBookRecord.delete() PhoneBookRecord.deleteAll() PhoneBookRecord.setPrimary() PhoneBookRecord.resetPrimary() PhoneBookRecord.fromVFormat() PhoneBookRecord.addCategory() PhoneBookRecord.deleteCategory() PhoneBookRecord.setCategoryView() PhoneBookRecord.createMailingList() PhoneBookRecord.addMailingListMember() PhoneBookRecord.deleteMailingListMember() RecentCallDialed.add()	User Data Write Capability

	RecentCallDialed.delete() RecentCallDialed.deleteAll()	
Wireless Messaging API - JSR 120		
javax.wireless.messaging.sms.send		Messaging
javax.wireless.messaging.sms.receive		Messaging
javax.microedition.io.Connector.sms		Messaging
javax.wireless.messaging.cbs.receive		Messaging
Multimedia Recording		
javax.microedition.media.RecordControl .startRecord	RecordControl.startRecord ( )	Multimedia Recording

Each phone call or messaging action will present the user with the destination phone number before the user approves the action. The handset will ensure that I/O access from the Mobile Media API follows the same security requirements as the Generic Connection Framework.

Displaying of Permissions to the User

Permissions will be divided into function groups and two high-level categories, with the function groups being displayed to the user. These two categories are Network/Cost related and User/Privacy related.

The Network/Cost related category will include net access, messaging, application auto invocation, and local connectivity function groups.

The user/privacy related category will include multimedia recording, read user data access, and the write user data access function groups. These function groups will be displayed in the settings of the MIDlet suite.

Only 3rd party and untrusted permissions can be modified or accessed by the user. Operator and manufacturer permissions will be displayed for each MIDlet suite, but cannot be modified by the user.

Trusted MIDlet Suites Using x.509 PKI

Using the x.509 PKI (Public Key Infrastructure) mechanism, the handset will be able to verify the signer of the MIDlet suite and bind it to a protection domain which will allow the MIDlet suite access to the protected API or function. Once the MIDlet suite is bound to a protection domain, it will use the permission defined in the protection domain to grant the MIDlet suite access to the defined protected APIs or functions.

The MIDlet suite is protected by signing the JAR file. The signature and certificates are added to the application descriptor (JAD) as attributes and will be used by the handset to verify the signature. Authentication is complete when the handset uses the root certificate (found on the handset) to bind the MIDlet suite to a protection domain (found on the handset).

Signing a MIDlet Suite

The default security model involves the MIDlet suite, the signer, and public key certificates. A set of root certificates are used to verify certificates generated by the signer. Specially designed certificates for code signing can be obtained from the manufacturer, operator, or certificate authority. Only root certificates stored on the handset will be supported by the Motorola C975 handset.

Signer of MIDlet Suites

The signer of a MIDlet suite can be the developer or an outside party that is responsible for distributing, supporting, or the billing of the MIDlet suite. The signer will have a public key infrastructure and the certificate will be validated to one of the protection domain root certificates on the handset. The public key is used to verify the signature of JAR on the MIDlet suite, while the public key is provided as a x.509 certificate included in the application descriptor (JAD).

MIDlet Attributes Used in Signing MIDlet Suites

Attributes defined within the manifest of the JAR are protected by the signature. Attributes defined within the JAD are not protected or secured. Attributes that appear in the manifest (JAR file) will not be overridden by a different value in the JAD for all trusted MIDlets. If a MIDlet suite is to be trusted, the value in the JAD will equal the value of the corresponding attribute in the manifest (JAR file), if not, the MIDlet suite will not be installed.

The attributes MIDlet-Permissions (-OPT) are ignored for unsigned MIDlet suites. The untrusted domain policy is consistently applied to the untrusted applications. It is legal for these attributes to exist only in JAD, only in the manifest, or in both locations. If these attributes are in both the JAD and the manifest, they will be identical. If the permissions requested in the HAD are different than those requested in the manifest, the installation will be rejected.

Methods:

1. MIDlet.getAppProperty will return the attribute value from the manifest (JAR) if one id defined. If an attribute value is not defined, the attribute value will return from the application descriptor (JAD) if present.

Creating the Signing Certificate

The signer of the certificate will be made aware of the authorization policy for the handset and contact the appropriate certificate authority. The signer can then send its distinguished name (DN) and public key in the form of a certificate request to the certificate authority used by the handset. The CA will create a x.509 (version 3) certificate and return to the signer. If multiple CAs are used, all signer certificates in the JAD will have the same public key.

Inserting Certificates into JAD

When inserting a certificate into a JAD, the certificate path includes the signer certificate and any necessary certificates while omitting the root certificate. Root certificates will be found on the device only.

Each certificate is encoded using base 64 without line breaks, and inserted into the application descriptor as outlined below per MIDP 2.0.

```
MIDlet-Certificate-<n>-<m>: <base64 encoding of a
certificate>
```

Note the following:

`<n>`:= a number equal to 1 for first certification path in the descriptor, or 1 greater than the previous number for additional certification paths. This defines the sequence in which the certificates are tested to see if the corresponding root certificate is on the device.

`<m>`:= a number equal to 1 for the signer's certificate in a certification path or 1 greater than the previous number for any subsequent intermediate certificates.

Creating the RSA SHA-1 signature of the JAR

The signature of the JAR is created with the signer's private key according to the EMSA-PKCS1 –v1_5 encoding method of PKCS #1 version 2.0 standard from RFC 2437. The signature is base64 encoded and formatted as a single MIDlet-Jar-RSA-SHA1 attribute without line breaks and inserted into the JAD.

It will be noted that the signer of the MIDlet suite is responsible to its protection domain root certificate owner for protecting the domain's APIs and protected functions; therefore, the signer will check the MIDlet suite before signing it. Protection domain root certificate owners can delegate signing MIDlet suites to a third party and in some instances, the author of the MIDlet.

Authenticating a MIDlet Suite

When a MIDlet suite is downloaded, the handset will check the JAD attribute MIDlet-Jar-RSA-SHA1. If this attribute is present, the JAR will be authenticated by verifying the signer certificates and JAR signature as described. MIDlet suites with application descriptors that do not have the attributes previously stated will be installed and invoked as untrusted. For additional information, refer to the MIDP 2.0 specification.

Verifying the Signer Certificate

The signer certificate will be found in the application descriptor of the MIDlet suite. The process for verifying a Signer Certificate is outlined in the steps below:

1. Get the certification path for the signer certificate from the JAD attributes MIDlet-Certificate-1<m>, where <m> starts a 1 and is incremented by 1 until there is no attribute with this name. The value of each attribute is abase64 encoded certificate that will need to be decoded and parsed.
2. Validate the certification path using the basic validation process as described in RFC2459 using the protection domains as the source of the protection domain root certificates.
3. Bind the MIDlet suite to the corresponding protection domain that contains the protection domain root certificate that validated the first chain from signer to root.
4. Begin installation of MIDlet suite.
5. If attribute MIDlet-Certificate-<n>-<m> with <n> being greater than 1 are present and full certification path could not be established after verifying MIDlet-Certificate-<1>-<m> certificates, then repeat step 1 through 3 for the value <n> greater by 1 than the previous value.

The following table describes actions performed upon completion of signer certificate verification:

Result	Action
Attempted to validate <n> paths. No public keys of the issuer for the certificate can be found, or none of the certificate paths can be validated.	Authentication fails, JAR installation is not allowed.
More than one full certification path is established and validated.	Implementation proceeds with the signature verification using the first successfully verified certificate path for authentication and authorization.
Only one certification path established and validated.	Implementation proceeds with the signature verification.

Verifying the MIDlet Suite JAR

The following are the steps taken to verify the MIDlet suite JAR:

1. Get the public key from the verified signer certificate.
2. Get the MIDlet-JAR-RSA-SHA1 attribute from the JAD.
3. Decode the attribute value from base64 yielding a PKCS #1 signature, and refer to RFC 2437 for more detail.
4. Use the signer's public key, signature, and SHA-1 digest of JAR to verify the signature. If the signature verification fails, reject the JAD and MIDlet suite. The MIDlet suite will not be installed or allow MIDlets from the MIDlet suite to be invoked as shown in the following table.
5. Once the certificate, signature, and JAR have been verified, the MIDlet suite is known to be trusted and will be installed (authentication process will be performed during installation).

The following is a summary of MIDlet suite verification including dialog prompts:

Initial State	Verification Result
JAD not present, JAR downloaded	Authentication can not be performed, will install JAR. MIDlet suite is treated as untrusted. The following error prompt will be shown, "Application installed, but may have limited functionality."
JAD present but is JAR is unsigned	Authentication can not be performed, will install JAR. MIDlet suite is treated as untrusted. The following error prompt will be shown, "Application installed, but may have limited functionality."
JAR signed but no root certificate present in the keystore to validate the certificate chain	Authentication can not be performed. JAR installation will not be allowed. The following error prompt will be shown, "Root certificate missing. Application not installed."
JAR signed, a certificate on the path is expired	Authentication can not be completed. JAR installation will not be allowed. The following error prompt will be shown, "Expired Certificate. Application not installed."
JAR signed, a certificate rejected for reasons other than expiration	JAD rejected, JAR installation will not be allowed. The following error prompt will be shown, "Authentication Error. Application not installed."
JAR signed, certificate path validated but signature verification fails	JAD rejected, JAR installation will not be allowed. The following error prompt will be shown, "Authentication Error. Application not installed."
Parsing of security attributes in JAD fails	JAD rejected, JAR installation will not be allowed. The following error prompt will be shown, "Failed Invalid File."

JAR signed, certificate path validated, signature verified	JAR will be installed. The following prompt will be shown, "Installed."

Carrier Specific Security Model

The MIDP 2.0 security model will vary based on carrier requests. Contact the carrier for specifics.

Bound Certificates

Bound certificates enable an efficient process to aid developers in the MIDlet development and testing phase when working with signed applications. Currently the delay for the developer occurs because specific flex files need to be created for each developer and for each domain being tested.

By implementing Bound certificates, the process of creating and supporting developer-specific flex files can be eliminated, which in turn simplifies the developer's environment, avoiding dependency from the flex tools. Bound certificates will take advantage of the High Assurance Boot system implemented at Motorola. The main difference becomes relevant during the creation of the signing certificate for the developer. Below are the steps necessary for the developer to follow:

- The MIDlet developer generates a signing key that contains public and private keys.
- The developer will send the CSR, containing the developer's public portion of the signing key, and the serial number(s) of the handset(s) the developer is using for testing, and the intended protected domains the MIDlet will be signed against to the Motorola Java Signing Center.
- The Signing Center constructs a developer certificate that includes the public key and a tag, that denotes this is a bound certificate.
- This bound certificate has the serial number for the unit appended to the certificate format and the resulting file is signed using the PCS Java CA.
- When the phone starts to load this type of certificate, it will identify the bound tag and pull the electronic number from the processor and use it to validate the signature of that certificate. Once this validation takes place, then the certificate will be used to validate the signature of the JAD file and if it passes, then it will install the JAR file on the product.

This implementation incorporates information about the target domain into the bound certificate used for signing. This information should be submitted along with developer's CSR and bound tag(s) of the target device(s). If Java security

manager has this information in runtime, it will be able to decide what domain to use for binding.

Following are the requirements concerning these bound certificates:

- An X.509 bound certificate shall support at least 10 serial numbers supplied in the special bound tag extension.
- A bound MIDlet shall be successfully installed on the target device, if at least one of serial numbers supplied in the bound certificate coincides with the processor's serial number retrieved from the target device.
- A bound MIDlet shall not be installed if bound tag verification fails.
- A bound MIDlet, after successful bound tag check, shall be successfully mapped to the hardcoded, SRP* compliant manufacturer domain, if the bound certificate includes information about target developer's domain where all permissions have type allowed.
- A bound MIDlet, after successful bound tag check, shall be successfully mapped to the hardcoded, SRP* compliant 3rd party domain, if the bound certificate includes information about target developer's domain where all permissions have type user.
- A bound MIDlet, after successful bound tag check, shall be mapped to a domain in accordance with flexed policy file, if the bound certificate either doesn't include any information about target developer's domain or includes information about domain unknown to the device. The MIDlet shall not be installed if the domain policy flexed on the target device doesn't include an appropriate domain.

* SRP - MIDP 2.0 Security Recommended Practice for GSM/UMTS compliant devices.

Appendix A: Key Mapping

Key Mapping for the Motorola C975

The table below identifies key names and corresponding Java assignments. All other keys are not processed by Java.

Key	Assignment
0	NUM0
1	NUM1
2	NUM2
3	NUM3
4	NUM4
5	SELECT, followed by NUM5
6	NUM6
7	NUM7
8	NUM8
9	NUM9
STAR (*)	ASTERISK
POUND (#)	POUND
JOYSTICK LEFT	LEFT
JOYSTICK RIGHT	RIGHT
JOYSTICK UP	UP
JOYSTICK DOWN	DOWN
SCROLL UP	UP
SCROLL DOWN	DOWN
SOFTKEY 1	SOFT1
SOFTKEY 2	SOFT2
MENU	SOFT3 (MENU)
SEND	SELECT Also, handled according to VSCL specification: incoming call accepted, if Java has high priority Also, call placed if pressed on lcdui.TextField or lcdui.TextBox with PHONENUMBER constraint set.

CENTER SELECT	SELECT
END	Handled according to VSCL specification: Pause/End/Resume/Background menu invoked.

Appendix B: Memory Management Calculation

Available Memory

The available memory on the Motorola C975 is the following:

- 4M shared memory for MIDlet storage
- 1.5 MB Heap size
- Recommended maximum MIDlet size is 200 Kb

Memory Calculation for MIDlets

The calculation for determining the amount of memory needed to run a MIDlet is computed by a formula. The details menu for the application show the Kilobytes required by the application as computed by the formula below:

(size of the JAD file)

\+ (size of the JAR file)

\+ (size of the data space used by the MIDlet)

==================================

(memory required to run the MIDlet)

Please note that the same memory calculation is applied while performing the memory check during the download of an application.

Appendix C: FAQ

Online FAQ

The MOTOCODER developer program is online and able to provide access to Frequently Asked Questions around enabling technologies on Motorola products.

Access to dynamic content based on questions from the Motorola J2ME developer community is available at the URL listed below.

http://www.motocoder.com

Appendix F: Spec Sheet

Motorola C975 Spec Sheet

Listed below are the spec sheets for the Motorola C975 handset. The spec sheet contains information regarding the following areas:

- Technical Specifications
- Key Features
- J2ME Information
- Motorola Developer Information
- Tools
- Other Related Information

MOTOROLA
intelligence everywhere™

Motorola C975

Developer Reference Sheet

Technical Specifications

Band/Frequency	UMTS 2100 MHz GSM 900/1800/1900 MHz GPRS (2U/4D, Class 10, B)
Region	North America
Technology	WAP 2.0, J2ME, SMS, EMS, MMS,
Connectivity	USB, via CE Bus
Dimensions	53.2 x 114 x 24.2 mm
Weight	139 grams
Display	1.9" 176 x 220
Operating System	Motorola
Chipset	TBD

Key Features

- 3D stereo sound
- Point to Point Video
- Integrated Digital Video/Still Camera
- Large Color Display
- Integrated MP3 Player
- iTAP Predictive Text Entry
- Transflash expandable memory

J2ME™ Information

CLDC v1.1 and MIDP v2.0 compliant

Maximum MIDlet suite size	200 KB
Heap size	1.5 MB
Maximum record store size	64 K
MIDlet storage available	TBD
Interface connections	HTTP 1.1, UDP, TCP
Maximum number of sockets	4
Supported image formats	GIF, JPEG, PNG, BMP
Double buffering	Supported
Encoding schemes	ISO8859_1, ISO10646
Input methods	Multitap, iTAP
Additional API's	JSR 118, JSR 120, JSR 135, JSR 139, JSR 184, JSR 185
Audio	MIDI, WAV, AMR, MP3, MP4, iMelody

Related Information

Motorola Developer Information:
Developer Resources at http://www.motocoder.com/

Tools:
CodeWarrior® Wireless Studio v7.0
J2ME™ SDK version v4.0
Motorola Messaging Suite v1.1

Documentation:
Creating Media for the C975

References:
J2ME™ specifications: http://www.java.sun.com/j2me
MIDP v2.0 specifications:
http://www.java.sun.com/products/midp
CLDC v1.0 specifications:
http://www.java.sun.com/products/cldc
WAP forum: http://www.wap.org
EMS standards: http://www.3GPP.org

Purchase:
Visit the Motocoder Shop at http://www.motocoder.com/
Accessories: http://www.motorola.com/consumer

MOTOROLA and the Stylized M Logo are registered in the U.S. Patent & Trademark Office. All other product or service names are the property of their respective owners. Java and all other Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

© Motorola, Inc. 2004.