Record Management System

Version 0.6, Draft

INFORMATION GUIDE

COPYRIGHT

Samsung Electronics Co. Ltd.

This material is copyrighted by Samsung Electronics. Any unauthorized reproductions, use or disclosure of this material, or any part thereof, is strictly prohibited and is a violation under the Copyright Law Samsung Electronics reserves the right to make changes in specifications at any time and without notice. The information furnished by Samsung Electronics in this material is believed to be accurate and reliable, but is not warranted true in all cases.

Trademarks and Service Marks

The Samsung Logo is the trademark of Samsung Electronics. Java is the trademark of Sun Microsystems.

All other company and product names may be trademarks of the respective companies with which they are associated.

About this Document

This document describes Record Management System with some sample code snippets.

Scope

This document is intended for Java ME developers who wish to develop Java ME applications. It assumes good knowledge of java programming language.
To know about Java ME basics and Java programming language, refer to the Knowledge Base under Samsung Mobile Innovator (SMI).

http://innovator.samsungmobile.com/platform.main.do?platformId=3

Document History:

Date	Version	Comment
08/03/10	0.6	Draft

References:

User Interface Programming

http://developers.sun.com/mobility/midp/articles/databaserms/
http://www.ibm.com/developerworks/library/wi-rms/

Abbreviations:

1	Java ME	Java Micro Edition
2	MIDP	Mobile Information Device Profile
3	API	Application Programming Interface
4	RMS	Record Management System

Table of Contents

Table of Figures

Introduction

Java ME enabled devices do not have robust files system to save application data. Therefore Record Management store is used to create file system like environment that is used to store and maintain persistence in Java ME enabled devices.

At the API level, a record store is represented by an instance of the *javax.microedition.rms.RecordStore* class. All RMS classes and interfaces are defined in the *javax.microedition.rms.*;* package.

Overview

The Java ME Record Management System (RMS) provides a mechanism through which MIDlets can insert records, read records and search for particular records and sort records stored by the RMS.

A key subsystem of the Mobile Information Device Profile (MIDP) is the Record Management System (RMS), an application programming interface (API) that gives MIDP applications local, on-device data persistence.

The Java ME *record management system* (RMS) provides a mechanism through which MIDlets can persistently store data and retrieve it later. In a record-oriented approach, Java ME RMS comprises multiple *record stores.* An overview of Java ME RMS and MIDlet interfacing is given in Figure 1.
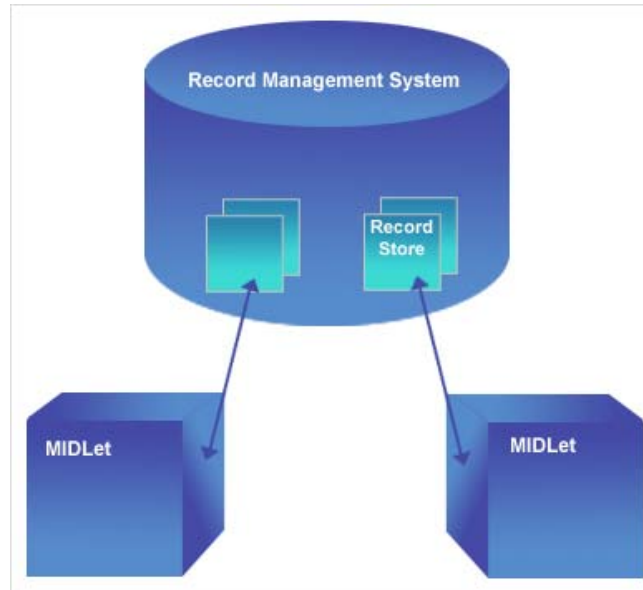
Figure 1 : Overview of Java ME RMS and MIDlet interfacing

The key concepts of the Record Management System are:

Records:

A *record* is an individual data item. RMS places no restrictions on what goes *into* a record: a record can contain a number, a string, an array, an image -- anything that a sequence of bytes can represent.

In RMS a record doesn't have any fields. Or, to put it more precisely, a record consists of a single binary field of variable size. *The responsibility for interpreting the contents of a record falls entirely on the application.* RMS provides the storage and a unique identifier, nothing else. While this division of labor complicates things for applications, it keeps RMS small and flexible -- important attributes for a MIDP subsystem.

Record Stores:

A *record* store is an ordered collection of records. Records are not independent entities: each must belong to a record store, and all record access occurs through the record store. In fact, the record store guarantees that records are read and written *atomically*, with no possibility of data corruption.
When a record is created, the record store assigns it a unique identifier, an integer called the *record ID*. The first record added to a record store has a record

ID of 1, the second a record ID of 2, and so on. A record ID is *not* an index: record deletions do not renumber existing records or affect the value of the next record ID. It basically store collection of records organized as rows (records) and columns (fields). Each row consists of two columns: One containing a unique integer that identifies the row in the record store.

Record ID	Data(Array of bytes)

Names are used to identify record stores within a MIDlet suite. A record store's *name* consists of 1 to 32 Unicode characters, and must be unique within the MIDlet suite that created the record store. In MIDP 1.0, record stores cannot be shared by different MIDlet suites. MIDP 2.0 optionally allows a MIDlet suite to share a record store with other suites, in which case the record store is identified by the names of the MIDlet suite and its vendor, along with the record store name itself.

Record stores also maintain time-stamp and version information so applications can discover when a record store was last modified. For close tracking, applications can register a listener to be notified whenever a record store is modified.

At the API level, a record store is represented by an instance of the *javax.microedition.rms.RecordStore* class.

The RecordStore API

All RMS classes and interfaces are defined in the *javax.microedition.rms* package. There are Four Interfaces and one Class in RecordStore API package as shown in table below

Interfaces	Description
RecordComparator	An interface defining a comparator which compares two records (in an implementation-defined manner) to see if they match or what their relative sort order is.
RecordEnumeration	An interface representing a bidirectional record store Record enumerator.
RecordFilter	An interface defining a filter which

	examines a record to see if it matches (based on application-defined criteria).
RecordListener	A listener interface for receiving Record Changed/Added/Deleted events from a record store.

RecordStore is only class in RecordStore API , a record store is represented by an instance of the *javax.microedition.rms.RecordStore* class.

Table below describes *RecordStore* class

Methods	Description
addRecord(byte[] data, int offset, int numBytes)	This method allows adding new record in a record store.
addRecordStoreListener(RecordListener listener)	Add the listener to the record store
closeRecordStore()	This method is called when MIDlet requests to have the record store closed
deleteRecord(int recordId)	This method is called to delete the record from the record store
deleteRecordStore(String recordStoreName)	This method is called when the MIDlet want to delete the record store
enumerateRecords(RecordFilter filter, RecordComparator comparator, boolean keepUpdated)	This method Returns an enumeration for traversing a set of records in the record store in an optionally specified order.
getLastModified()	Returns the time of the last Record modified
getName()	This method returns the name of the record store.
getNextRecordID()	Returns the recordId of the next record to be added to the record store.
getNumRecords()	Returns the number of records currently in the record store.
getRecord(int recordId)	Returns a copy of the data stored in the given record.
getRecord(int recordId, byte[] buffer,	Returns the data stored in the given

int offset)	record.
getRecordSize(int recordId)	Returns the size (in bytes) of the MIDlet data available in the given record.
getSize()	Returns the amount of space, in bytes, that the record store occupies.
getSizeAvailable()	Returns the amount of additional room (in bytes) available for this record store to grow.
getVersion()	Each time a record store is modified (by addRecord, setRecord, or deleteRecord methods) its version is incremented.
listRecordStores()	Returns an array of the names of record stores owned by the MIDlet suite.
openRecordStore(String recordStoreName, boolean createIfNecessary)	Open (and possibly create) a record store associated with the given MIDlet suite.
openRecordStore(String recordStoreName, boolean createIfNecessary, int authmode, boolean writable)	Open (and possibly create) a record store that can be shared with other MIDlet suites.
openRecordStore(String recordStoreName, String vendorName, String suiteName)	Open a record store associated with the named MIDlet suite.
removeRecordListener(RecordListener listener)	Removes the specified RecordListener.
setMode(int authmode, boolean writable)	Changes the access mode for this RecordStore.
setRecord(int recordId, byte[] newData, int offset, int numBytes)	Sets the data in the given record to that passed in.

Exceptions

The checked exceptions in package *javax.microedition.rms* are as shown in table

Exceptions	Description
InvalidRecordIDException	This exception is thrown when an operation cannot be performed because the record ID is invalid.

RecordStoreFullException	This exception is thrown when there is no more space available in the record store.
RecordStoreNotFoundException	This Exception is thrown when the MIDlet tries to non-existing record store.
RecordStoreNotOpenException	This Exception is thrown when the MIDlet is tries to access a record store which is already closed.
RecordStoreException	This Exception is thrown to indicate general Exceptions occurred in the record store operation.

Basic RMS operations

Create a RecordStore

To create a recordstore you can use *RecordStore.openRecordStore()* with the second parameter set to true.

```
//open record if present else create a record
RecordStore rs = null;
try {
    rs = RecordStore.openRecordStore("RecordName", true);
} catch (Exception exe) {
    //unable to open record or create record
    exe.printStackTrace();
} finally{
    try{
    //to close the record store
     rs. closeRecordStore();
    }catch(Exception ex){
    ex. printStackTrace();

    }
}
```

Close Record Store

To close the record store you can call *closeRecordStore()* it will close specific Record Store

```
...
try{
rs. closeRecordStore();
} catch (Exception ex){


}
…
```

Add Record to RecordStore

To add record get bytes of record and call *addRecord()* to add the record to the record store

```
…
    String str= "firstRecord";
    //get bytes of record
    byte[] rec = str.getBytes();
    try {
       // added record to recordstore
       rs.addRecord(rec, 0, rec.length);
    } catch (Exception e) {
       e.printStackTrace();
    }
…
```

Read Record from RecordStore

Set the array of bytes using the record size and call *getRecord()* method to retrieve the Record Data.

```
…
byte[] getData;
```

```
        try {
          for (int i = 1; i <= rs.getNumRecords(); i++) {
            getData = new byte[rs.getRecordSize(i)];
            rs.getRecord(i, getData, 0);
            System.out.println("RecordData=" + new String(getData));
          }
        } catch (Exception exe) {
}
…
```

Set Record into Record Store

Sets the data in the given record by passed Record ID & data in byte array format. After this method returns, a call to *getRecord(int recordId)* will return an array of numBytes size containing the data supplied here.

```
…
try{
        byte []data= "rmsdata".getBytes();
         rs.setRecord(1, data,0, data.length);

}catch(Exception ex){
}
…
```

Delete Records form Record Store

To delete the record of specific id call the *deleteRecord(int recordID)*

```
…
try{

      for(int i=rs. getNumRecords(); i>0;i--){
      rs.deleteRecord(i);

   }
} catch(Exception ex){

…
}
```

Enumerating Records

Instead of checking each record ID RMS to return you an enumeration of valid record IDs,using the *RecordEnumeration* interface.

hasNextElement() & *nextRecordId()*   has use to move forward through the enumeration:

```
...
RecordStore rs = … //Open record store
RecordEnumeration enum = ... //take record enumeration

try {
   while( enum.hasNextElement() ){
      int id = enum.nextRecordId();
      byte[] data = rs.getRecord( id );
      ... // do something here
   }
}
catch( RecordStoreException e ){
   // handle the error here
}
...
```

To move backward, using *hasPreviousElement()* and *previousRecordId()*. Note that both *nextRecordId()* and *previousRecordId()* throw *InvalidRecordIDException* if a record is deleted from a record store while the enumeration is active, if it's not tracking changes to the record store.

Add Stream Record to Record Store

Different java data types to the RecordStore can be added by reading through streams

```
…

ByteArrayOutputStream byteOutStream = new ByteArrayOutputStream();
DataOutputStream dataOutStream = new DataOutputStream(byteOutStream);
dataOutStream.writeUTF(userName.getString());
```

```
dataOutStream.writeUTF(password.getString());
recordStore.addRecord( byteOutStream.toByteArray(),0,
                                    byteOutStream.toByteArray().length);

…
```

Read Stream data from RecordStore

```
…

byte[] userdata = recordStore.getRecord(1);
ByteArrayInputStream byteArrayStream = new
ByteArrayInputStream(userdata);
DataInputStream dataInputStream = new DataInputStream(byteArrayStream);
logindata[0] = dataInputStream.readUTF();
logindata[1] = dataInputStream.readUTF();
dataInputStream.close();
byteArrayStream.close();
…
```

Deleting the Record Store

```
…
private final String S_LOGIN = "loginRS";

…
try{
        RecordStore.deleteRecordStore(S_LOGIN);
}catch(Exception ex){
}

…
```

Interface RecordComparator API

RecordComparator defining a comparator which compares two records (in an implementation-defined manner) to see if they match or what their relative sort order is. The application implements this interface to compare two candidate

records. The return value must indicate the ordering of the two records. The compare method is called by *RecordEnumeration* to sort and return records in an application specified order

EQUIVALENT: In terms of search or sort order, the two records are the same.

FOLLOWS: In term of search or sort order left (first parameter) record follows the right (second parameter).

PRECEDES: In terms of search or sort order left (first parameter) record precedes the right (second parameter) record.

```
public class CompareRecord implements RecordComparator{

public int compare(byte[] rec1, byte[] rec2){
                …
               If(…)
                        return EQUIVALENT;
               else if(…)
                        return FOLLOWS;
               else if(…)
                        return PRECEDES;
               …
        }

}
```

Interface RecordFilter API

RecordFilter defining a filter which examines a record to see if it matches (based on an application-defined criteria). The application implements the *match()* method .

Returns *true* if record found
Return *false* if record not found

```
public class FilerRecord implements RecordFilter{

        public boolean match(byte[] candidate){
```

```
            if(…)
                return true;
            else
                return false;
    }


}
```

Interface RecordListener API

A listener interface for receiving Record Changed/Added/Deleted events from a record store.

Here are the basic steps for working with a *RecordListener*:

- Open (create) a record store.
- Create a new listener
- Implement each method in the RecordListener interface

void recordAdded(RecordStore recordStore, int recordId)
void recordChanged(RecordStore recordStore, int recordId)
void recordDeleted(RecordStore recordStore, int recordId)

```
public class RecordUpdate implements RecordListener{

…
        public void recordAdded(RecordStore recordStore,
                    int recordId){
        }
        public void recordChanged(RecordStore recordStore,
                    int recordId){
        }
        public void recordDeleted(RecordStore recordStore,
                    int recordId){
        }
…
```

}

Sample code snippet:

```
import javax.microedition.midlet.MIDlet;
import javax.microedition.lcdui.Display;
import javax.microedition.lcdui.Displayable;
import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.CommandListener;
import javax.microedition.lcdui.Form;
import javax.microedition.lcdui.TextField;
import javax.microedition.lcdui.Alert;
import javax.microedition.lcdui.AlertType;
import javax.microedition.rms.RecordStore;
import java.io.ByteArrayInputStream;
import java.io.DataInputStream;
import java.io.ByteArrayOutputStream;
import java.io.DataOutputStream;

public class LoginMidlet extends MIDlet implements CommandListener{

    private final Command cmdExit = new Command("Exit",Command.EXIT,1);
    private final Command cmdSave = new Command("Save",Command.ITEM,1);

    private final String  S_LOGIN ="loginRS";
    private Display display = null;
    private Form loginForm = null;
    private TextField userName = null;
    private TextField password = null;
    private String logindata[]=null;
    private RecordStore recordStore = null;
    private Alert alert = null;

    public LoginMidlet(){
        init();
        logindata = openRecordStore();
        closeRecord();
        if(logindata!=null){
            userName.setString(logindata[0]);
            password.setString(logindata[1]);
```

```
    }
    display.setCurrent(loginForm);
}

public void startApp() {
}

public void pauseApp() {
}

public void destroyApp(boolean unconditional) {

}
private void init(){

    display      = Display.getDisplay(this);
    loginForm    = new Form("Login Screen");
    loginForm.addCommand(cmdExit);
    loginForm.addCommand(cmdSave);
    loginForm.setCommandListener(this);
    userName = new TextField("User Name",null,30,TextField.ANY);
    password = new TextField("Password",null,30,TextField.PASSWORD);
    loginForm.append(userName);
    loginForm.append(password);
    alert = new Alert("","",null,AlertType.INFO);


}
private void showMessage(String title,String text){
    alert.setTitle(title);
    alert.setString(text);
    alert.setTimeout(400);
    display.setCurrent(alert, loginForm);

}
private void closeRecord(){
    try {
        recordStore.closeRecordStore();
    } catch (Exception ex) {
        ex.printStackTrace();
```

```
        }
    }
    private String[] openRecordStore(){
        String logindata[]=null;
        try {

            recordStore = RecordStore.openRecordStore(S_LOGIN, true);
            if(recordStore.getNumRecords()>0){
                logindata = new String[2];
                byte[] userdata = recordStore.getRecord(1);
                ByteArrayInputStream byteArrayStream = new
ByteArrayInputStream(userdata);
                DataInputStream dataInputStream = new
DataInputStream(byteArrayStream);
                logindata[0] = dataInputStream.readUTF();
                logindata[1] = dataInputStream.readUTF();
                dataInputStream.close();
                byteArrayStream.close();
            }

        } catch (Exception ex) {
            ex.printStackTrace();
        }
        return logindata;
    }
    private void deleteLoginRecord(){
        try {

            RecordStore.deleteRecordStore(S_LOGIN);

        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
    private void addRecord(){
        try {
            deleteLoginRecord();
            openRecordStore();
            ByteArrayOutputStream byteOutStream = new ByteArrayOutputStream();
```

```
        DataOutputStream dataOutStream = new
DataOutputStream(byteOutStream);
        dataOutStream.writeUTF(userName.getString());
        dataOutStream.writeUTF(password.getString());
        recordStore.addRecord( byteOutStream.toByteArray(),0,
byteOutStream.toByteArray().length);
        closeRecord();
    } catch (Exception ex) {
        ex.printStackTrace();
    }

 }

  public void commandAction(Command cmd,Displayable dis){
    if(cmd.equals(cmdExit)){
        destroyApp(true);
        notifyDestroyed();
    }else if(cmd.equals(cmdSave)){
        if((userName.getString().length()>0) &&
(password.getString().length()>0)){
            addRecord();
            showMessage("Info","Data Successfully Saved");

        }else{
            showMessage("Info","TextField is empty!!!");
        }
    }
  }
}
```