FORUM NOKIA

Introduction To The PIM API

Version 1.0; November 25, 2004

Java™

NOKIA

Contents

Change History

November 25, 2004	Version 1.0	Initial document release

1 Introduction

This document is an introduction to the PIM API [JSR-075], including a brief description of the API and an example. It assumes familiarity with Java™ programming and the basics of Mobile Information Device Profile (MIDP) programming, as described in the Forum Nokia document *MIDP 1.0: Introduction To MIDlet Programming* [MIDPPROG]. The PIM API is subject to security restrictions, and therefore you should also be familiar with MIDP 2.0 security framework concepts; the Forum Nokia *MIDP 2.0: Tutorial On Signed MIDlets* [SIGNMID] provides insight into this security model.

The PIM API was specified in JSR-75: PDA Optional Packages for the J2ME™ Platform, which includes two Java™ 2 Platform, Micro Edition (J2ME™) optional packages oriented to support features typical of PDA-like devices. The optional packages give access to personal information management (PIM API) databases and the ability to handle local file systems (FileConnection API). These two packages are independent of each other, and thus devices may contain either one or both.

2 PIM API

2.1 Introduction

The PIM API is an optional J2ME package that gives support for accessing and modifying the PIM databases that may exist in a MIDP device. The intention is to give a standardized interface to those databases, which could be used across many types of devices in a secure fashion. As with most APIs that imply some I/O operation in the device, invoking the PIM API should be done in a thread different than the GUI thread to avoid potential deadlocks.

The PIM API currently supports three types of databases or lists: Contact lists, Event lists, and To-Do lists. The three types of databases are not necessarily always available in a particular device, but the specification requires that, given the API has been implemented in a device, at least one database of one type should be available. An implementation may contain more than one list of the same type, for example, a mobile device can have a contact list contained in the device's memory and another in the device's SIM card.

The PIM API refers to the databases as `PIMList` objects and to the individual data pieces as `PIMItems`. Three types of `PIMLists` are supported by the API: `ContactList`, `EventList`, and `ToDoList`. The `PIM` class has two `openPIMList` methods to get access to a given `PIMList`. The method will take the type of database and access mode used (READ_ONLY, WRITE_ONLY, READ_WRITE). The `PIM` class is a singleton that can be obtained using the `getInstance()` static method. To reduce the need for security permissions, it is recommended to request the minimum access rights needed when opening a database.

Each database is identified by a unique name that is used when opening the database. This name is implementation dependent but the `listPIMLists` method can be used to get a list of all the database names available for a given type. Moreover, in Nokia devices this name corresponds to the localized name of the database. This means that the main contact database could be called "Contacts," "Contactos," or "Puhelinluettelo," depending on the device's language. In addition, high-end devices have the ability to let the user customize this name or create new lists. Therefore, it is strongly advised to use `listPIMList` instead of a hard-coded list name when opening a database.

`PIMList` contains several `items()` methods to access the full content of `PIMItems` as well as selecting a particular subset of them. These methods should be used with care since they can be slow if the underlying database is very large.

`PIMList` also gives support for creating, modifying, and deleting `PIMItems`. It is important to notice that when creating new items or modifying an existing one, the modifications are not reflected in the native database until the `commit()` method is called. On the contrary, deleting an entry is done immediately upon request. To do any of these operations, the database has to be open in WRITE_ONLY or READ_WRITE mode.

There are three types of `PIMItems` corresponding to each type of `PIMList`: `Contact`, `Event`, and `ToDo`. They contain the actual data for a particular item. The data is stored in fields that contain data such as telephone numbers or addresses. These fields are different for `Contact`, `Event`, and `ToDo` objects and are identified by numerical constants defined on each of the interfaces. A field may contain more than one entry for the same field type, for instance, a `Contact` may contain multiple telephone numbers.

Each `PIMList` doesn't necessarily support all the fields described in the API. Instead, each implementation will define the supported fields, and in case of Nokia devices, this corresponds to the fields on the underlying native database. Hence, it is highly recommended to verify that a field is available in a given `PIMList` by calling the `isSupportedField()` or `getSupportedFields()` methods before attempting to read a field. Note that even databases residing in the same device may have a different set of fields; for example, the contact list from a device's memory contains several extra fields compared to the contact list stored in a SIM card which contains only two fields. These different sets of fields should be accounted for when designing an application so that it is compatible with a wide variety of devices. Each supported field may contain zero or more entries for a given `PIMItem`. The `countValues()` method should be used to ensure that a `PIMItem`'s field has been populated and to count how many entries it contains. Failing to do so may lead to `IndexOutOfBounds` exceptions being thrown.

Each entry may also have extra attributes to further describe an item. For example, TEL fields frequently have attributes to indicate if it is a fixed line, a fax, or a mobile line number. It is important to verify that a `PIMList` supports the attributes you are interested in; this can be done by using the `isSupportedAttribute()` and `getSupportedAttributes()` methods.

The fields and attributes described in the API are logical values, and often it is necessary to produce a human-readable label for a specific item. The `getFieldLabel()` and `getAttributeLabel()` methods in `PIMList` will return localized labels as indicated by the implementation. The `getSupportedFields()` method is also useful for a good user interface, given that it returns the supported fields in the order recommended for the device's UI.

The API also supports categories to group sets of fields. Nokia devices support categories conforming to the support for them in the native databases.

The PIM API contains methods for serializing and deserializing `PIMItems`, thus enabling MIDlets to operate with external applications. The serialization format is the standard vCard [VCARD] for `Contacts` and vCalendar [VCALENDAR] for `Event` and `ToDo` items. The exact version of the supported standards is implementation dependent and it can be obtained with the `PIM.supportedSerialFormats` method.

When it is necessary to verify that the PIM API is actually available in a particular device, the system property `microedition.pim.version` should be checked. If the package is available, this property will contain the version of the API available, currently version 1.0, otherwise it will be null.

2.2 Security

Access to personal data has obvious security and privacy implications. Many of the operations will require the MIDlet to acquire appropriate permission, either by explicit user approval or by being granted permission as a trusted MIDlet. It is important to realize that in these operations a `SecurityException` can be thrown and must be handled properly.

MIDP 2.0 MIDlets are either trusted or untrusted [SIGNMID]. In the latter case, the device cannot assure the MIDlet's origin and integrity, and therefore calling restricted APIs is not allowed without explicit user permission. This means that whenever you need to open or modify a PIM database, a user prompt will appear and the user must explicitly authorize the operation.

In the case of trusted MIDlets, the device can determine their origin and integrity by means of X.509 certificates. Those MIDlets may acquire permissions automatically depending on the security domain they belong to. If a trusted MIDlet wants to acquire PIM API-specific permissions, they must be stated in the Java Application Descriptor (JAD) file under the `MIDlet-Permission` property, and the MIDlet must be properly signed.

There are several permissions for the PIM API to access each database type with read and write rights:

- `javax.microedition.pim.ContactList.read`
- `javax.microedition.pim.ContactList.write`
- `javax.microedition.pim.EventList.read`
- `javax.microedition.pim.EventList.write`

- `javax.microedition.pim.ToDoList.read`
- `javax.microedition.pim.ToDoList.write`

These permission are checked when the `openPIMList` and `listPIMLists` methods are called. Note that when a database is opened as READ and an attempt to write is executed, a `SecurityException` will be thrown.

The MIDP 2.0 specification [MIDP 2.0] identifies two functions groups related to user data manipulation. These are the "Read User Data Access" and "Write User Data Access" groups, and they include the respective read and write permissions. The function groups give a higher-level view of permission to users, allowing them to change permissions in blocks. Using the function groups, users could, for instance, authorize a third party to access all the PIM databases without having to approve each permission. Table 1 shows the defaults and allowed permission modes for untrusted and trusted third-party MIDlets.

Function group	**Trusted third-party domain**		**Untrusted domain**	
	Default setting	**Allowed settings**	**Default setting**	**Allowed settings**
Read User Data Access	Oneshot	Session, Blanket, Oneshot, No	Oneshot	Oneshot, No
Write User Data Access	Oneshot	Session, Blanket, Oneshot, No	No	Oneshot, No

Table 1: Allowed and default permission modes

2.3 Nokia Implementation Notes

Since the PIM API leaves room for important variations among different manufacturers, it is important to be aware of certain implementation details for Nokia devices. The Nokia implementation API may differ between the different Developer Platforms, namely the Series 40 and Series 60 Developer Platforms, but it will be consistent in different devices of the same platform.

The main source for implementation-specific details is the mapping of native fields to PIM API fields. This is described in detail in the following section.

2.3.1 Contact list database

At least one contact database is available in Nokia devices, but it is common to have two databases — one for the contact list in the main memory and another for contacts in the SIM card. More databases might be available, depending on the device. These databases can be selected by name with the `openPIMList(int pimListType, int mode, String name)` method. The names used to identify the database correspond to the localized name used by the device, and it is possible in certain cases that they could even be user defined. Therefore, the name should not be hard coded in the application; instead, it should be obtained with the `listPIMLists` method. The order of the database names returned by the `listPIMLists` method is also the preferred order in the device. Commonly, there are two databases that may be used. The first is the "in-memory" database, which is used by default, and the second is the list in the SIM card. The approach to open the default database is to either call `openPIMList(int pimListType, int mode)` or pick the first entry from `listPIMLists` and call `openPIMList(int pimListType, int mode, String name)`.

The fields and attribute contained in each database vary depending on the device model and the type of database. The following tables list the supported fields for SIM, Series 40, and Series 60 contact databases.

Native contact field	PIM API contact field type	PIM API contact field attributes
Name	Contact.FORMATTED_NAME	
Phone number	Contact.TEL	Contact.ATTR_PREFERRED

Table 2: Supported fields for SIM contact database

Native contact field	PIM API contact field type	PIM API contact field attributes
Name	Contact.FORMATTED_NAME	PIMItem.ATTR_NONE
General phone number	Contact.TEL	Contact.ATTR_NONE
Mobile number	Contact.TEL	Contact.ATTR_MOBILE
Home number	Contact.TEL	Contact.ATTR_HOME
Office number	Contact.TEL	Contact.ATTR_WORK
Fax number	Contact.TEL	Contact.ATTR_FAX
E-mail	Contact.EMAIL	Contact.ATTR_HOME
Web address	Contact.URL	PIMItem.ATTR_NONE
Postal address	Contact.ADDR	Contact.ATTR_HOME
Note	Contact.NOTE	PIMItem.ATTR_NONE
Image	Contact.PHOTO	PIMItem.ATTR_NONE

Table 3: Supported fields for Series 40 contact database

Native contact field	PIM API contact field type	PIM API contact field attributes
First name	Contact.NAME_GIVEN	PIMItem.ATTR_NONE
Last name	Contact.NAME_FAMILY	PIMItem.ATTR_NONE
Last name <space> First name	Contact.FORMATTED_NAME	
Company	Contact.ORG	PIMItem.ATTR_NONE
Job title	Contact.TITLE	PIMItem.ATTR_NONE
Address	Contact.ADDR	PIMItem.ATTR_NONE
Address (home)	Contact.ADDR	Contact.ATTR_HOME
Address (business)	Contact.ADDR	Contact.ATTR_WORK
Telephone	Contact.TEL	PIMItem.ATTR_NONE
Telephone (home)	Contact.TEL	Contact.ATTR_HOME
Telephone (business)	Contact.TEL	Contact.ATTR_WORK
Mobile	Contact.TEL	Contact.ATTR_MOBILE
Mobile (home)	Contact.TEL	Contact.ATTR_MOBILE + Contact.ATTR_HOME
Mobile (business)	Contact.TEL	Contact.ATTR_MOBILE + Contact.ATTR_WORK

Native contact field	PIM API contact field type	PIM API contact field attributes
Fax	Contact.TEL	Contact.ATTR_FAX
Fax (home)	Contact.TEL	Contact.ATTR_FAX + Contact.ATTR_HOME
Fax (business)	Contact.TEL	Contact.ATTR_FAX + Contact.ATTR_WORK
E-mail	Contact.EMAIL	PIMItem.ATTR_NONE
E-mail (home)	Contact.EMAIL	Contact.ATTR_HOME
E-mail (business)	Contact.EMAIL	Contact.ATTR_WORK
Pager	Contact.TEL	Contact.ATTR_PAGER
Birthday	Contact.BIRTHDAY	PIMItem.ATTR_NONE
URL	Contact.URL	PIMItem.ATTR_NONE
URL (home)	Contact.URL	Contact.ATTR_HOME
URL (business)	Contact.URL	Contact.ATTR_WORK
Note	Contact.NOTE	PIMItem.ATTR_NONE
Photo	Contact.PHOTO	PIMItem.ATTR_NONE

Table 4: Supported fields for Series 60 contact database

In Series 60 devices, the addresses are an array of strings with the different components of the address. The indexes of this string array are indicated in Table 5.

Address component	PIM contact field index
P.O. Box	Contact.ADDR_POBOX
Extension	Contact.ADDR_EXTRA
Street	Contact.ADDR_STREET
Postal/ZIP	Contact.ADDR_POSTALCODE
City	Contact.ADDR_LOCALITY
State/province	Contact.ADDR_REGION
Country/region	Contact.ADDR_COUNTRY

Table 5: Array of address string components for Series 60 devices

As shown in Tables 2 through 4, there is a wide variation in the fields supported among different database types. An application must handle these differences appropriately, for example by targeting a specific device model or using the least-common-denominator approach. The `isSupportedField` method can be used to detect which database type you're actually dealing with.

2.3.2 Event list database

Once again, there are different fields supported in Series 40 and Series 60 devices. Both Series 40 and Series 60 devices support several events list databases whose names are localized, representing different categories of events. However, in an abstract way the following lists are returned by `listPIMLists`:

- Series 40: {"Meeting", "Call", "Birthday", "Memo", "Reminder"}

- Series 60: {"Appointment", "Event", "Anniversary"}

The order of these lists is preserved when calling `listPIMLists`, so although the first entry can have a localized name of "Meeting" in a Series 40 device, the logical meaning of the first entry is the same.

The supported fields in the event databases are different in each type of list. Table 6 indicates the supported fields and mapping of the native fields in a Series 40 device.

Event database	Native field	PIM API field
Meeting	Subject	Event.SUMMARY
	Location	Event.LOCATION
	Start time	Event.START
	End time	Event.END
	Alarm	Event.ALARM
Call	Phone number	Event.SUMMARY
	Name	Event.NOTE
	Time	Event.START
	Alarm	Event.ALARM
Birthday	Name	Event.SUMMARY
	Year of birth	Event.NOTE
	Time	Event.START
	Alarm	Event.ALARM
Memo	Subject	Event.SUMMARY
	Start date	Event.START
	End date	Event.END
	Alarm	Event.ALARM
Reminder	About	Event.SUMMARY
	Start time	Event.START
	Alarm	Event.ALARM

Table 6: Supported fields in the Series 40 event database

In Series 60 devices, the event database has a different mapping, as indicated in Table 7.

Event database	Native field	PIM API field
Meeting	Subject	Event.SUMMARY
	Location	Event.LOCATION
	Start time	Event.START
	Start date	
	End time	Event.END
	End date	

Event database	Native field	PIM API field
	Alarm	Event.ALARM
Memo	Subject	Event.SUMMARY
	Start date	Event.START
	End date	Event.END
Anniversary	Occasion	Event.SUMMARY
	Date, time	Event.START

Table 7: Supported fields in the Series 60 event database

In both Series 40 and Series 60 devices, events can have a repeating rule obtained using the `getRepeat` method. The mapping of `RepeatRule` attributes to the native fields is shown in Table 8.

Series 40 repeat type	Series 60 repeat type	PIM API RepeatRule attribute
Never	Not repeated	`getRepeat` returns null
Every day	Daily	(FREQUENCY, DAILY)
Every week	Weekly	(FREQUENCY, WEEKLY)
Every two week(s)	Fortnightly	(FREQUENCY, WEEKLY) + (INTERVAL, 2)
Every month	Monthly	(FREQUENCY, MONTHLY)
Every year	Yearly	(FREQUENCY, YEARLY)

Table 8: Mappings of repeat type in Series 40 and Series 60 devices

2.3.3 ToDo database

Series 40 and Series 60 devices support the same set of attributes for the ToDo databases. They support only one ToDo database, so there is no need to look up the database name. The mapping of fields is in Table 9.

Native field	PIM API ToDo field
Subject	ToDo.SUMMARY
Priority	ToDo.PRIORITY
Due date	ToDo.DUE
Completed	ToDo.COMPLETED
Completion date	ToDo.COMPLETION_DATE

Table 9: Supported ToDo fields in Series 40 and Series 60 devices

ToDo items may also have an order of priority; while the PIM API supports ten levels of priority, current Nokia devices support only four. These are mapped as shown in Table 10.

PIM API priority	Native priority
0	Not used
1, 2, 3	High
4, 5, 6	Medium
7, 8, 9	Low

Table 10: Mapping of ToDo priorities

3 Event-Sharing MIDlet

The example developed in this tutorial is an application for sharing a particular event with other users. The application will provide the means to build an event, serialize it as vCalendar, and send its contents as a message to another device, perhaps obtaining its number from the contact list. It will also add it to the local event database. The message is sent in a Short Message Service (SMS) with the Wireless Messaging API [WMA], making use of the serialization facilities provided in the PIM API. The example is deliberately simple; a real-world application should include features such as conflict resolution and acknowledgments. Also note that, in general, all exceptions, when caught, are simply displayed to the user instead of giving a user-friendly message.

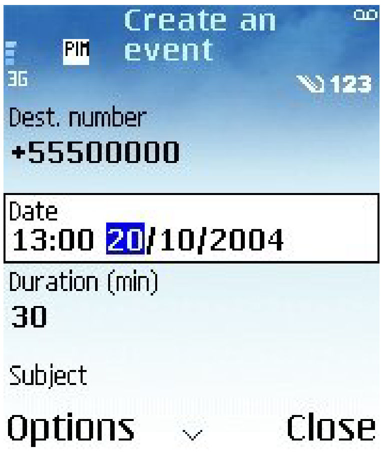
The MIDlet will start with a screen to define the event and send it. In this screen the user can set the destination number, the topic of the event, the start date/time, and the duration in minutes. It is also possible to look up the destination number in the contact list by using the “Find Contacts” menu item. This will display a list of contact names and phone number(s). Once a message is sent, the event is inserted in the local database.

The MIDlet will establish itself by listening for incoming messages on a given SMS port (6553). When a message is received, a new screen will pop up, the message will be decoded, and the event will be displayed on the screen. At that moment, the recipient can accept the insertion of new data to the calendar.

The MIDlet also registers itself in the `PushRegistry` waiting for incoming messages in the given SMS port. This registration is done statically using a MIDlet-Push entry in the JAD file.

All operations that are related to reading or writing the PIM databases are done in a separate thread to avoid deadlocks with the UI thread.

Figures 1, 2, and 3 depict screen shots of the application.


Create an event

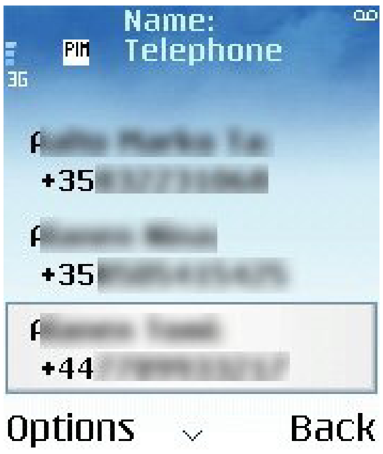
Dest. number
+55500000

Date
13:00 20/10/2004

Duration (min)
30

Subject

Figure 1: Sending meeting request screen

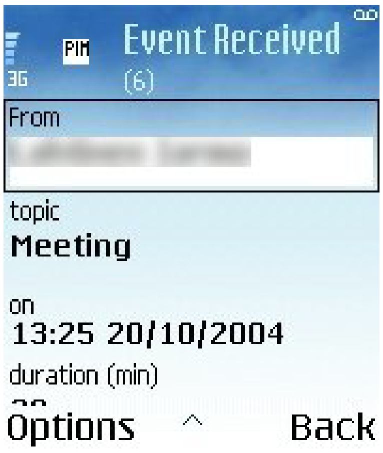


A[redacted]
+35[redacted]

A[redacted]
+35[redacted]

A[redacted]
+44[redacted]

Figure 2: Look up contacts screen



From
[redacted]

topic
Meeting

on
13:25 20/10/2004

duration (min)

Figure 3: Incoming meeting request screen

Figure 4 presents a simplified class diagram of the Event Sharing MIDlet application.

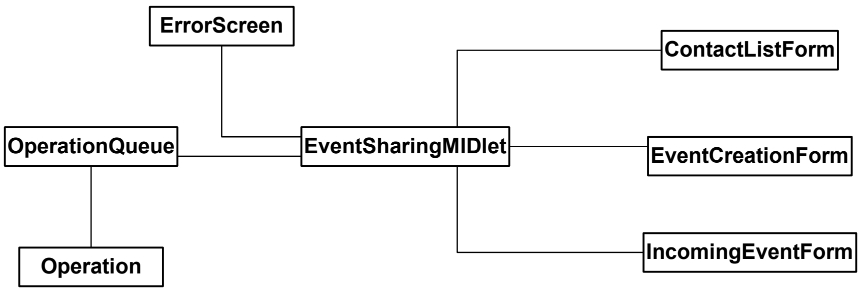


Figure 4: Event-sharing MIDlet class diagram

3.1 EventSharingMIDlet

This is the main class of the MIDlet; it contains the initialization of the MIDlet and the handling of transitions between different screens. All SMS handling is done in this class as well, including `MessageConnection` creation, reception of incoming messages, and sending of new messages. At startup, this class also checks if it has been awaken via `PushRegistry` because of an incoming message.

The class contains .an `OperationsQueue` instance, which is used by other classes to execute PIM-related operations in a separate thread.

```
import java.io.*;
import java.util.*;
import javax.microedition.io.*;
import javax.microedition.lcdui.*;
import javax.microedition.midlet.*;
import javax.microedition.pim.*;
import javax.wireless.messaging.*;

// Main class of the EventSharing MIDlet
public class EventSharingMIDlet
  extends MIDlet
  implements MessageListener
{
  private final static int DEFAULT_PORT = 6553;
  private final static String SMS_PREFIX = "sms://:";

  private final Image logo;
  private final EventCreationScreen mainScreen;

  private int port;
  private MessageConnection conn;
  private Message nextMessage;
  private OperationsQueue queue = new OperationsQueue();
  private final static int CONTACTS_LIST_OP = 0;
  private final Display display;


  public EventSharingMIDlet()
  {
    // init basic parameters
    logo = makeImage("/logo.png");
    try
    {
      port = Integer.parseInt(getAppProperty("port"));
    }
    catch (Exception e)
    {
      // in case the property is missing or with a wrong format
      port = DEFAULT_PORT;
    }
    ErrorScreen.init(logo, Display.getDisplay(this));
    mainScreen = new EventCreationScreen(this, port);
    display = Display.getDisplay(this);
  }


  void sendMessage(String number, byte[] text)
  {
    showMessage("Going to send the mgs " + text + " " + conn);
    if (conn != null)
    {
      try
      {
        // create a new message
        BinaryMessage requestSMS = (BinaryMessage) conn.newMessage(
          MessageConnection.BINARY_MESSAGE);
        String address = new StringBuffer("sms://")
          .append(number).append(":").append(port).toString();
        requestSMS.setAddress(address);

        requestSMS.setPayloadData(text);
        conn.send(requestSMS);
      }
      catch (IOException e)
      {
        showMessage(e.getMessage());
      }
      catch (SecurityException e)
      {
        showMessage(e.getMessage());
      }
    }
  }


  public void startApp()
  {
    Displayable current = Display.getDisplay(this).getCurrent();
    if (current == null)
    {
      // check that the API is available
      boolean isAPIAvailable =
        (System.getProperty("microedition.pim.version") != null);
      // shows splash screen
      StringBuffer splashText =
        new StringBuffer(getAppProperty("MIDlet-Name")).
          append("\n").append(getAppProperty("MIDlet-Vendor")).
          append(isAPIAvailable?"":"\nPIM API not available");
      Alert splashScreen = new Alert(null,
        splashText.toString(),
        logo,
        AlertType.INFO);
      splashScreen.setTimeout(3000);
      if (!isAPIAvailable)
      {
        display.setCurrent(splashScreen, mainScreen);
      }
      else
      {
        display.setCurrent(splashScreen, mainScreen);
        // List connections from PushRegistry
        String smsConnections[] = PushRegistry.listConnections(true);

        // Check the connections. We'll assume only one is of interest
        for (int i = 0; i < smsConnections.length; i++)
        {
          if (smsConnections[i].startsWith(SMS_PREFIX))
          {
            try
            {
              conn =
                (MessageConnection) Connector.open(smsConnections[i]);
              BinaryMessage incomingMessage =
                (BinaryMessage) conn.receive();
              showIncomingMessage(incomingMessage);
              conn.close();
            }
            catch (IOException e)
            {
              showMessage(e.getMessage());
            }
            catch (SecurityException e)
            {
              showMessage(e.getMessage());
            }
          }
        }

        // Build the connection string
        String connection = SMS_PREFIX + port;
        try
        {
          // Initiate the connection and add listener
          conn = (MessageConnection) Connector.open(connection);
          conn.setMessageListener(this);
        }
        catch (IOException e)
        {
          showMessage(e.getMessage());
        }
        catch (SecurityException e)
        {
          showMessage(e.getMessage());
        }
      }
    }
    else
    {
      Display.getDisplay(this).setCurrent(current);
    }
  }


  public void pauseApp()
  {
  }


  public void destroyApp(boolean unconditional)
  {
    try
    {
      // Close the connection on exit
      if (conn != null)
      {
        conn.close();
      }
    }
    catch (IOException e)
    {
      // Ignore since we are closing anyway
    }
    queue.abort();
  }


  // Asynchronous callback for inbound message.
  public void notifyIncomingMessage(MessageConnection conn)
  {
    if (conn == this.conn && conn != null)
    {
      try
      {
        // Create a ReceiveScreen upon message removal
        BinaryMessage incomingMessage = (BinaryMessage) conn.receive();
        showIncomingMessage(incomingMessage);
      }
      catch (IOException e)
      {
        showMessage(e.getMessage());
      }
      catch (SecurityException e)
      {
        showMessage(e.getMessage());
      }
    }
  }


  void showIncomingMessage(BinaryMessage msg)
  {
    display.setCurrent(new IncomingEventForm(this, msg));
  }
```

```
    void showMessage(String message)
    {
      ErrorScreen.showError(message, mainScreen);
    }


    void showMessage(String message, Displayable mainScreen)
    {
      ErrorScreen.showError(message, mainScreen);
    }


    // adds an operation the queue
    void enqueueOperation(Operation operation)
    {
      queue.enqueueOperation(operation);
    }


    // shows the main screen
    void showMain()
    {
      display.setCurrent(mainScreen);
    }


    // shows the contacts list screen
    void showContactsList()
    {
      display.setCurrent(
        new ContactListForm(EventSharingMIDlet.this));
    }


    // callback from contacts list when a particular
    // telephone number has been selected
    void contactSelected(String telephoneNumber)
    {
      mainScreen.setTargetPhoneNumber(telephoneNumber);
      showMain();
    }


    // loads a given image by name
    static Image makeImage(String filename)
    {
      Image image = null;

      try
      {
        image = Image.createImage(filename);
      }
      catch (Exception e)
      {
        // use a null image instead
      }

      return image;
    }

  }
```

3.2 EventCreationForm

This form contains the functionality to create an event and send it to another device. It contains several fields to define the topic of the event, starting time, and duration. In this screen, it is necessary to enter the destination phone number. This number can be looked up on the contacts database. To do this, the "Find Contacts" command will request for a `ContactListScreen` to be displayed and the selected number will be inserted in the destination number field.

Once the fields are filled and the "Send" command is given, the contents of the fields are validated and a new Event is created, imported into the first `EventList` database, and serialized as a vCalendar message. This message is later sent to the destination number via a request to `EventSharingMIDlet`. All of this is done in a separate thread whose logic is contained in the inner class `SendEventMesssageOperation`.

```
import java.io.*;
import java.util.*;
import javax.microedition.io.*;
import javax.microedition.lcdui.*;
import javax.microedition.midlet.*;
import javax.microedition.pim.*;
import javax.wireless.messaging.*;

// Screen used to create a new event and send it to a contact
class EventCreationScreen
 extends Form
 implements CommandListener
{
  private final Command sendCommand, exitCommand, findCommand;
  private final TextField number, duration, topic;
  private final DateField dateTime;
  private final EventSharingMIDlet parent;
  private final int port;


  public EventCreationScreen(EventSharingMIDlet parent, int port)
  {
    super("Create an event");
    this.parent = parent;
    this.port = port;

    // init UI
    sendCommand = new Command("Send", Command.OK, 1);
    findCommand = new Command("Find Contacts", Command.OK, 0);
    exitCommand = new Command("Exit", Command.EXIT, 1);

    number = new TextField("Dest. number",
                           "",
                           20,
                           TextField.ANY);
    dateTime = new DateField("Date",
                             DateField.DATE_TIME,
                             TimeZone.getDefault());
                             dateTime.setDate(new Date());
    duration = new TextField("Duration (min)",
                             "30",
                             24,
                             TextField.ANY);
    topic = new TextField("Subject",
                          "",
                          24,
                          TextField.ANY);

    // create the form
    append(number);
    append(dateTime);
    append(duration);
    append(topic);

    addCommand(findCommand);
    addCommand(sendCommand);
    addCommand(exitCommand);
    setCommandListener(this);
  }


  void setTargetPhoneNumber(String phoneNumber)
  {
    number.setString(phoneNumber);
  }


  public void commandAction(Command cmd, Displayable displayable)
  {
    if (cmd == sendCommand)
    {
      // do a sanity check and then
      // send the message in a separate thread
      parent.enqueueOperation(
        new SendEventMessageOperation(number.getString(),
                                      dateTime.getDate(),
                                      duration.getString(),
                                      topic.getString()));
    }
    else if (cmd == exitCommand)
    {
      parent.notifyDestroyed();
    }
    else if (cmd == findCommand)
    {
      parent.showContactsList();
    }
  }


  private class SendEventMessageOperation implements Operation
  {
    private final String number, duration, topic;
    private final Date date;


    SendEventMessageOperation(String number,
      Date date,
      String duration,
      String topic)
    {
      this.number = number;
      this.date = date;
      this.duration = duration;
      this.topic = topic;
    }


    public void execute()
    {
      int length = 0;
      try
      {
        length = Integer.parseInt(duration);
      }
      catch (NumberFormatException e)
      {
        parent.showMessage("Duration not valid",
          EventCreationScreen.this);
        return;
```

```
}
if (length <= 0)
{
  parent.showMessage("Duration needs to be positive",
    EventCreationScreen.this);
}
else if (number == null || number.length() == 0)
{
  parent.showMessage("Number not entered",
    EventCreationScreen.this);
}
else if (topic == null || topic.length() == 0)
{
  parent.showMessage("Subject not entered",
    EventCreationScreen.this);
}
else
{
  ByteArrayOutputStream out = new ByteArrayOutputStream();
  EventList eventList = null;
  try
  {
    PIM pim = PIM.getInstance();
    String listNames[] = pim.listPIMLists(PIM.EVENT_LIST);
    if (listNames.length > 0)
    {
      eventList = (EventList) pim.openPIMList(PIM.EVENT_LIST,
                                              PIM.READ_WRITE,
                                              listNames[0]);
      Event newEvent = eventList.createEvent();
      if (eventList.isSupportedField(Event.SUMMARY))
      {
        newEvent.addString(Event.SUMMARY,
          PIMItem.ATTR_NONE,
          topic);
      }
      if (eventList.isSupportedField(Event.START))
      {
        newEvent.addDate(Event.START,
          PIMItem.ATTR_NONE,
          date.getTime());
      }
      if (eventList.isSupportedField(Event.END))
      {
        newEvent.addDate(Event.END,
          PIMItem.ATTR_NONE,
          date.getTime() + 60 * 1000 * length);
      }
      // let's check that VCALENDAR/1.0 is supported
      String supportedFormats[] = PIM.getInstance()
        .supportedSerialFormats(PIM.EVENT_LIST);
      for (int i=0;i<supportedFormats.length;i++) {
        if (supportedFormats[i].equals("VCALENDAR/1.0")) {
          PIM.getInstance().toSerialFormat(newEvent,
                                           out,
                                           "UTF-8",
                                           "VCALENDAR/1.0");
          break;
        }
      }
      if (out.size() == 0)
      {
        parent.showMessage("VCALENDAR/1.0 not supported",
          EventCreationScreen.this);
      }
      // let's add the event locally
      // an advanced version should wait for an ack
      eventList.importEvent(newEvent);
      newEvent.commit();
```

```
          }
          else
          {
            parent.showMessage("No Event list available",
              EventCreationScreen.this);
          }
        }
        catch (PIMException e)
        {
          parent.showMessage(e.getMessage(), EventCreationScreen.this);
        }
        catch (SecurityException e)
        {
          parent.showMessage(e.getMessage(), EventCreationScreen.this);
        }
        catch (UnsupportedEncodingException e)
        {
          // should not happen since UTF-8 is mandatory
        }
        finally
        {
          try
          {
            if (eventList != null)
            {
              eventList.close();
            }
          }
          catch (PIMException e)
          {
            // ignore, we are closing anyway
          }
        }
        // out could be empty if there is no support
        // for VCALENDAR/1.0
        if (out.size() > 0)
        {
          parent.sendMessage(number, out.toByteArray());
        }
      }

    }
  }

}
```

3.3 IncomingEventForm

This form is activated when an incoming message is received. When this happens, the message is decoded, assuming it uses a format accepted by the PIM implementation, and its contents are displayed on the screen. The address of the incoming messages is parsed to extract the phone number and then the contacts database is searched attempting to identify the name of the sender based on the phone number.

```
import java.io.*;
import java.util.*;
import javax.microedition.lcdui.*;
import javax.microedition.pim.*;
import javax.wireless.messaging.*;

// Forms displayed when a message is received
class IncomingEventForm
  extends Form
  implements CommandListener
{
  private Command acceptCommand, exitCommand, backCommand;
  private TextField number, duration, topic;
  private DateField dateTime;
  private final EventSharingMIDlet parent;
  private Event event;


  IncomingEventForm(EventSharingMIDlet parent, BinaryMessage msg)
  {
    super("Event Received");
    this.parent = parent;

    backCommand = new Command("Back", Command.BACK, 2);
    exitCommand = new Command("Exit", Command.EXIT, 2);

    addCommand(backCommand);
    addCommand(exitCommand);

    setCommandListener(this);
    parent.enqueueOperation(new LoadEvent(msg));
  }


  public void commandAction(Command cmd, Displayable displayable)
  {
    if (cmd == acceptCommand)
    {
      // insert the event in a different thread
      parent.enqueueOperation(new InsertEvent());
    }
    else if (cmd == exitCommand)
    {
      parent.notifyDestroyed();
    }
    else if (cmd == backCommand)
    {
      parent.showMain();
    }
  }


  // finds a name based on the number in all contact lists
  private String findName(String number)
  {
    String[] allLists = PIM.getInstance()
          .listPIMLists(PIM.CONTACT_LIST);
    if (allLists.length > 0)
    {
      String results[] = new String[allLists.length];
      for (int i = 0; i < allLists.length; i++)
      {
        results[i] = findNameInList(number, allLists[i]);
      }
      // if there is more than one, only the first is returned
      for (int i = 0; i < allLists.length; i++)
      {
        if (results[i] != null)
        {
          return results[i];
        }
      }
    }
    return null;
  }


  // finds a name based on the number in a particular contact list
  // if for some reason it cannot be found, return null
  private String findNameInList(String number, String list)
  {
    ContactList contactList = null;
```

```
    try
    {
      contactList = (ContactList) PIM.getInstance()
            .openPIMList(PIM.CONTACT_LIST, PIM.READ_ONLY, list);
      if (contactList.isSupportedField(Contact.TEL)
        && contactList.isSupportedField(Contact.FORMATTED_NAME))
      {
        Contact pattern = contactList.createContact();
        pattern.addString(Contact.TEL, PIMItem.ATTR_NONE, number);
        Enumeration matching = contactList.items(pattern);
        if (matching.hasMoreElements())
        {
          // will only return the first match
          Contact ci = (Contact) matching.nextElement();
          // FORMATTED_NAME is mandatory
          return ci.getString(Contact.FORMATTED_NAME, 0);
        }
      }
    }
    catch (PIMException e)
    {
      //just ignore and return null;
    }
    catch (SecurityException e)
    {
      //just ignore and return null;
    }
    finally
    {
      if (contactList != null)
      {
        try
        {
          contactList.close();
        }
        catch (PIMException e)
        {
          // ignore, nothing can be done here
        }
      }
    }
    return null;
  }


  private class InsertEvent implements Operation
  {
    public void execute()
    {
      EventList eventList = null;
      try
      {
        PIM pim = PIM.getInstance();
        String listNames[] = pim.listPIMLists(PIM.EVENT_LIST);
        if (listNames.length>0)
        {
          eventList = (EventList) pim.openPIMList(PIM.EVENT_LIST,
                                        PIM.READ_WRITE,
                                        listNames[0]);
          // Check that the fields are supported
          if (eventList.isSupportedField(Event.SUMMARY) &&
            eventList.isSupportedField(Event.START) &&
            eventList.isSupportedField(Event.END))
          {
            // event cannot be null at this stage
            eventList.importEvent(event).commit();
            parent.showMessage("Event inserted in the local database");
          }
        }
        else
        {
          parent.showMessage("No Event list found");
        }
      }
      catch (PIMException e)
      {
        parent.showMessage(e.getMessage(), IncomingEventForm.this);
      }
      catch (SecurityException e)
      {
        parent.showMessage(e.getMessage(), IncomingEventForm.this);
      }
      finally
      {
        if (eventList != null)
        {
          try
          {
            eventList.close();
          }
          catch (PIMException e)
          {
            // nothing to do here, just ignore
          }
        }
      }
    }
  }

  // This operation reads an event from a TextMessage
  private class LoadEvent implements Operation
  {
    private final BinaryMessage msg;


    LoadEvent(BinaryMessage msg)
    {
      this.msg = msg;
    }


    public void execute()
    {
      PIMItem items[] = null;
      try
      {
        items = PIM.getInstance().fromSerialFormat(
              new ByteArrayInputStream(msg.getPayloadData()),
              "UTF-8");
      }
      catch (PIMException e)
      {
        parent.showMessage(e.getMessage(), IncomingEventForm.this);
      }
      catch (UnsupportedEncodingException e)
      {
        // should not happen since UTF-8 is mandatory
      }
      if (items != null && items.length>0)
      {
        // let's assume that only one event
        // was contained in the message
        event = (Event) items[0];
        // Sanity check
        int summaryCount = event.countValues(Event.SUMMARY);
        int startCount = event.countValues(Event.START);
        int endCount = event.countValues(Event.END);
        if (summaryCount>0 && startCount>0 && endCount>0)
        {
          topic = new TextField("topic",
                                event.getString(Event.SUMMARY, 0),
                                24,
                                TextField.ANY);
          String source = msg.getAddress();
          // assume the message starts with sms:// and has a port number
          String phoneNumber = source.substring(6,
                source.lastIndexOf(':'));
          String sourceName = findName(phoneNumber);

          if (sourceName != null)
          {
            number = new TextField("From",
                                   sourceName,
                                   20,
                                   TextField.ANY |
                                   TextField.UNEDITABLE);
          }
          else
          {
            number = new TextField("From",
                                   phoneNumber,
                                   20,
                                   TextField.PHONENUMBER |
                                   TextField.UNEDITABLE);
          }
          long startDate = event.getDate(Event.START, 0);
          long length = event.getDate(Event.END, 0) - startDate;
          // get the length in minutes
          length /= 60000;
          dateTime = new DateField("on",
                                   DateField.DATE_TIME,
                                   TimeZone.getDefault());
          dateTime.setDate(new Date(startDate));

          duration = new TextField("duration (min)",
                                   "" + length,
                                   24,
                                   TextField.ANY);

          append(number);
          append(topic);
          append(dateTime);
          append(duration);

          acceptCommand = new Command("Accept", Command.OK, 1);
          addCommand(acceptCommand);
        }
        else
        {
          append(new StringItem("", "Incoming event was incomplete"));
        }
      }
      else
      {
        event = null;
        acceptCommand = null;
        append(new StringItem("", "Error during message decoding"));
      }
    }
  }

}
```

3.4 ContactsListForm

This form displays the list of contacts with their phone numbers so that the user can select one. Contacts without a phone number are discarded and those with multiple numbers are included with an entry for each number. Numbers are sorted with the mobile numbers appearing first.

The user can select a particular contact/number combination and return that information to the `EventCreationScreen`.

```
import java.util.*;
import javax.microedition.lcdui.*;
import javax.microedition.pim.*;

// This form shows a list of the contacts in the local databases
class ContactListForm
  extends List
  implements CommandListener
{
  private final Command exitCommand, selectCommand, backCommand;
  private final EventSharingMIDlet parent;
  private boolean available;
  private Vector allTelNumbers = new Vector();


  public ContactListForm(EventSharingMIDlet parent)
  {
    super("Contacts", Choice.IMPLICIT);
    this.parent = parent;

    // init UI
    selectCommand = new Command("Select", Command.OK, 0);
    backCommand = new Command("Back", Command.BACK, 1);
    exitCommand = new Command("Exit", Command.EXIT, 1);

    addCommand(backCommand);
    addCommand(exitCommand);
    setCommandListener(this);
    setFitPolicy(Choice.TEXT_WRAP_ON);

    // load the list of names in a differnt thread
    parent.enqueueOperation(new LoadContacts());
  }


  public void commandAction(Command cmd, Displayable displayable)
  {
    // if no names are available return
    if (!available)
    {
      parent.showMain();
      return;
    }
    else if (cmd == selectCommand)
    {
      int selected = getSelectedIndex();
      if (selected >= 0)
      {
        // will get the number from the list
        parent.contactSelected(
          (String) allTelNumbers.elementAt(selected));
      }
      else
      {
        parent.showMain();
      }
    }
    else if (cmd == backCommand)
    {
      parent.showMain();
    }
    else if (cmd == exitCommand)
    {
      parent.notifyDestroyed();
    }
  }


  // loads the names of a named contact list
  private void loadNames(String name)
    throws PIMException, SecurityException
  {
    ContactList contactList = null;
    try
    {
      contactList = (ContactList) PIM.getInstance()
        .openPIMList(PIM.CONTACT_LIST, PIM.READ_ONLY, name);


      // First check that the fields we are interested in are supported
      // by the PIM List
      if (contactList.isSupportedField(Contact.FORMATTED_NAME)
          && contactList.isSupportedField(Contact.TEL))
      {
        // Put an informative title while reading the contacts
        setTitle("Contacts");
        append("Reading contacts...", null);
        Enumeration items = contactList.items();
        // change the title to use the localized name.
        // getFieldLabel returns an i18n version
        delete(0);
        setTitle(contactList.getFieldLabel(Contact.FORMATTED_NAME)
                 + ": "
                 + contactList.getFieldLabel(Contact.TEL));
        Vector telNumbers = new Vector();
        while (items.hasMoreElements())
        {
          Contact contact = (Contact) items.nextElement();
          int telCount = contact.countValues(Contact.TEL);
          int nameCount = contact.countValues(Contact.FORMATTED_NAME);

          // we're only interested in contacts with a phone number
          // nameCount should always be > 0 since FORMATTED_NAME is
          // mandatory
          if (telCount > 0 && nameCount > 0)
          {
            // here we use the first entry in formatted named
            String contactName =
              contact.getString(Contact.FORMATTED_NAME, 0);
            // go through all the phone numbers
            for (int i = 0; i < telCount; i++)
            {
              // check if it is a cell phone and put it at the beginning
              // this doesn't necessarily work since in many cases it is
              // up to the user to indicate whether it is a mobile phone
              int telAttributes =
                contact.getAttributes(Contact.TEL, i);
              String telNumber =
                contact.getString(Contact.TEL, i);
              // check if ATTR_MOBILE is supported
              if (contactList.
                isSupportedAttribute(Contact.TEL,
                                     Contact.ATTR_MOBILE))
              {
                if ((telAttributes & Contact.ATTR_MOBILE) != 0)
                {
                  telNumbers.insertElementAt(telNumber, 0);
                }
                else
                {
                  telNumbers.addElement(telNumber);
                }
              }
              else
              {
                telNumbers.addElement(telNumber);
              }
              allTelNumbers.addElement(telNumber);
            }
            // Shorten names which are too long
            if (contactName.length()>20)
            {
              contactName = contactName.substring(0, 17)
                + "...";
            }
            // insert elements in the list in order
            for (int i = 0; i < telNumbers.size(); i++)
            {
              append(contactName
                + ": "
                + telNumbers.elementAt(i), null);
            }
            telNumbers.removeAllElements();
          }
        }
        available = true;
      }
      else
      {
        append("Contact list required items not supported", null);
        available = false;
      }
    }
    finally
    {
      // always close it
      if (contactList != null)
      {
        contactList.close();
      }
    }
  }


  // load the names in a separate thread
  private class LoadContacts implements Operation
  {
    public void execute()
    {
      try
      {
        // go through all the lists
        String[] allContactLists = PIM.getInstance()
          .listPIMLists(PIM.CONTACT_LIST);
        if (allContactLists.length != 0)
        {
          for (int i = 0; i < allContactLists.length; i++)
          {
            loadNames(allContactLists[i]);
          }
          addCommand(selectCommand);
        }
        else
        {
          append("No Contact lists available", null);
          available = false;
        }
```

```
            }
            catch (PIMException e)
            {
              parent.showMessage(e.getMessage(), ContactListForm.this);
              available = false;
              append("Press a key to return", null);
            }
            catch (SecurityException e)
            {
              parent.showMessage(e.getMessage(), ContactListForm.this);
              available = false;
              append("Press a key to return", null);
            }
          }
        }
      }
```

3.5 Support Classes

The MIDlet uses other supporting classes, including:

- `ErrorScreen`: Used to report error messages.
- `OperationsQueue` and `Operation`: This creates a queue of operations that are executed in a separate thread.

4 References

[JSR-075] JSR-75: PDA Optional Packages for the J2ME™ Platform, Java Community Process, 2004, http://jcp.org/aboutJava/communityprocess/final/jsr075/index.html

[MIDPPROG] *MIDP 1.0: Introduction to MIDlet Programming,* Forum Nokia, 2004, http://www.forum.nokia.com | Technologies | Java | Code and Examples

[SIGNMID] *MIDP 2.0: Tutorial On Signed MIDlets,* Forum Nokia, 2004, http://www.forum.nokia.com/documents

[MIDP 2.0] *Mobile Information Device Profile 2.0*, Java Community Process, 2002, http://jcp.org/aboutJava/communityprocess/final/jsr118/index.html

[VCARD] *vCard The Electronic Business Card Version 2.1*, versit Consortium 1996, http://www.imc.org/pdi/vcard-21.txt

[VCALENDAR] *vCalendar The Electronic Calendaring and Scheduling Exchange Format Version 1.0*, versit Consortium 1996, http://www.imc.org/pdi/vcal-10.txt

5 Evaluate This Document

In order to improve the quality of documentation, we kindly ask you to fill in the document survey.