Sensor API

Version 0.9, Draft

JSR 256

COPYRIGHT

Trademarks and Service Marks

About This Document

This document describes JSR 256 Sensor API followed by a sample code.

Scope

This document is for the users who have knowledge of Java ME and need a brief introduction to JSR 256.

Document History

Date	Version	Comment
02/06/09	0.9	Draft

References

- Sensor API JSR:

  http://jcp.org/en/jsr/detail?id=256

- Sensor API Article:

  http://developers.sun.com/mobility/apis/articles/opengles_mobilesensor/

Abbreviations

MIDP	Mobile Information Device Profile
CLDC	Connected Limited Device Configuration
AMS	Application Management Software
JSR	Java Specification Request
GCF	Generic Connection Framework
API	Application Programming Interface
URL	Uniform Resource Locator

Table of Contents

Introduction 6
Overview 6
Use Cases of Sensor 6
API Description 7
javax.microedition.sensor 7
javax.microedition.sensor.control 8
Sensor 9
Sensor Detection 9
Monitoring Sensor 13
Creating Connection 13
Data Capture 14
Synchronous Data 14
Asynchronous Data 15
Adding Conditions 16
LimitCondition 16
RangeCondition 17
ObjectCondition 17
Sensor Deactivation 18
Detecting Sensor API's presence 18
Security & Permissions 18
Private Sensor 19
Protected Sensor 19
Public Sensor 19
PushRegistry 19
Availability-based push 20
Condition-based push 20
Sensor Example 21
Class: SensorMIDlet 21
Class: SensorCanvas 24

List of Tables

Table 1: Sensor Package Interface Information 7
Table 2: Sensor Package Class Information 8
Table 3: Control Package Interface Information..... 8
Table 4: Examples of defining sensors with quantity and context type 10
Table 5: Permissions..... 19

Introduction

JSR 256 Mobile Sensor API allows Java ME application to fetch data easily and uniformly from sensors. Sensor API does not provide any methods for controlling the sensors. The API offers unified way of managing sensors, connected to the mobile devices, and easy access to the sensor data.

The API is targeted to the resource-constrained devices and needs to be memory-efficient to run.

The API is designed as an Optional Package that can be used with many Java Micro Edition Profiles. The minimum platform required by the API is, Connected Limited Device Configuration (CLDC), version 1.1.

Overview

A sensor is any measurement data source. Sensors are of different types:

- Physical sensors such as magnetometers and accelerometers.
- Virtual sensors such as battery level sensor combine and manipulate the data they receive from other kinds of physical sensors. Battery level sensor indicates the left over charge in a battery.

Use Cases of Sensor

Sensors can be widely used for the following scenarios:

- **Gaming:** Device itself acts as a game control providing motion related input modalities.
- **Better usability of a device:** Improves accessibility and usability of device such as change in display backlight, display orientation.
- **Monitor health condition:** Monitors the user heart rate, body temperature etc.
- **Outdoor activities:** Supports step counter etc.
- **Monitor external conditions:** Air pressure, temperature, wind and humidity.

API Description

The JSR 256 API is split into two parts:

- javax.microedition.sensor
- javax.microedition.sensor.control

javax.microedition.sensor

The package *javax.microedition.sensor* is a mandatory package and provides API for receiving the information from a sensor.

It contains 6 classes and 10 interfaces.

The following table shows the interface and classes.

Table 1: Sensor Package Interface Information

Interface	Description
Channel	Provides information about the channel and maintain conditions for monitoring the data.
ChannelInfo	Provides information about the data properties of the channel.
Condition	Sets condition to monitor data and receives notification when data meets defined condition
ConditionListener	Provides notifications when monitored data meets the condition defined by the application.
Data	Represents data values retrieved from one channel of a sensor.
DataAndErrorListener	Implemented by the application to receive data and errors from the sensor.
DataListener	Implemented by the application to receive data from the sensor.
SensorConnection	Abstraction of an actual sensor
SensorInfo	Contains a variety of information about the physical sensor.
SensorListener	Represents a listener that receives notifications when the availability of the sensor changes.

Table 2: Sensor Package Class Information

Class	Description
LimitCondition	Condition intended for numeric data to set various kind of conditions.
MeasurementRange	Represents the measurement range of one channel of the sensor.
ObjectCondition	Checks the equality of the set limit to the measured data value.
RangeCondition	Checks if the measured data value is within the defined range.
SensorManager	Used to find sensors and monitor their availability.
Unit	Represents the unit of the measured data values.

javax.microedition.sensor.control

The package *javax.microedition.sensor.control* is an optional package that represents examples of possible controls like starting, stopping, calibrating, and setting a measurement range or accuracy, and provides an interface for implementing new controls. It contains 6 interfaces. Table 3 shows the interfaces.

Table 3: Control Package Interface Information

Interface	Description
Control	Represents a sensor control enabling the sensor's control.
Controllable	Represents a sensor which can be controlled with the controls provided.
MeasurementRangeControl	Example of a general control used to set the measurement range of the sensor.
SampleRateControl	Example of a general control used to set the sample rate of the sensor.
StartControl	Example of a general control used to start the sensor by calling its *execute()* method.
StopControl	StopControl object is an example of a general control used to stop the sensor by calling its *execute()* method.

Sensor

The application that wishes to utilize sensor data typically performs following operations with a sensor:

- Sensor detection (Discovery)
- Sensor activation (Connection, Monitoring)
- Data capture (Listener, Condition)
- Sensor deactivation

Sensor Detection

The main functionality of Mobile Sensor API is to fetch the sensor data and monitor it based on a set of conditions. The appropriate sensor has to be found or known beforehand in order to use. An application can search for a desired sensor based on **quantity** and **context type.**

The quantity is the property the sensor is measuring.

The context type represents the environment where the measurement is taken, such as "*ambient*" or "*device*".

Context type categorizes sensors into four groups:

- **Ambient**: sensors measuring some ambient property of the environment.
- **Device**: sensors measuring properties related to the device.
- **User**: sensors measuring some function of the user.
- **Vehicle**: sensors measuring properties related to a vehicle.

The application can use either one independently or specify both.

To help the user to find the sensor, some readymade combinations and the sensor following from the definition, are presented in table 4.

Table 4: Examples of defining sensors with quantity and context type

Sensor	Quantity	Context type
accelerometer	acceleration	user
alcometer	alcohol	user
altimeter	altitude	user
ambient light sensor	ambient_light	ambient
amperemeter	electric_current	ambient
anemometer	wind_speed	ambient
barometer	pressure	ambient
battery charge level sensor	battery_charge	device
blood pressure meter	blood_pressure	user
clinical thermometer	temperature	user
clock, time gauge	time	ambient
compass	direction	ambient
fingerprint sensor	fingerprint	user
flip state sensor	flip_state	device
GPS	location	user
heartrate sensor/RR interval	heart_rate/RR_inteval	user
humiditymeter	humidity	ambient
illuminance sensor	illuminance	ambient
“is charged" sensor	is_charged	device
joystick	direction_of_motion	user
keyboard	character	device
luminance sensor	luminance	ambient
microphone	sound_intensity	ambient
milometer	length	ambient
mouse	direction_of_motion	user
network field intensity indicator	network_field_intensity	device
orientation sensor	orientation	device
power sensor	power	ambient
proximity sensor	proximity	ambient
radiation sensor	absorbed_dose	ambient
respiration sensor	respiration	user
rotation sensor	rotation	device

scales	mass	user
sensor measuring user's blood glucose level	blood_glucose_level	user
sensor measuring user's blood oxygen saturation	blood_oxygen_saturation	user
sensor measuring user's fat percentage	body_fat_percentage	user
sensor measuring user's gestures	gesture	user
skin conductance sensor	skin_conductance	user
step counter	step_count	user
teslameter	magnetic_flux_density	ambient
thermometer for air	temperature	ambient
tilt angle sensor	angle	device
velocimeter	velocity	user

SensorManager provides two static methods to find sensors.

findSensors(quantity, contextType)

```
public static SensorInfo[] findSensors(java.lang.String quantity, java.lang.String contextType)
```

where quantity can be one of those present in Table 4 and contextType can be among the following values:

- SensorInfo.CONTEXT_TYPE_AMBIENT
- SensorInfo.CONTEXT_TYPE_DEVICE
- SensorInfo.CONTEXT_TYPE_USER
- SensorInfo.CONTEXT_TYPE_VEHICLE.

If both parameters are null, all supported sensors are returned. If only one of the parameters is null then only the other is used as search criteria.

findSensors(URL)

```
public static SensorInfo[] findSensors(java.lang.String url)
```

Specific Sensor can also be found by providing the URL. For example URL *"sensor:acceleration;contextType=device;model=acc01"* specifies acceleration sensor.

The above two methods return an array of *SensorInfo* objects listing the found sensors. *SensorInfo* contains the information of sensor properties such as the model, the vendor and maximum sampling rate.

SensorInfo contains the information of sensor properties listed below:

- Connection Type- *embedded, remote, wireless, wired*
- Context Type- *user, device, ambient, vehicle*
- Description- description of sensor
- Model vendor- specific model information
- Quantity qualifier- to help finding the proper sensor
- URL- URL needed for sensor connection
- ChannelInfo- channels of sensor
- Property sensor- specific properties
- MaxBufferSize- maximum data buffer size

The following code snippet shows how to find sensors.

```
/* get sensors specified by quantity and context type*/
SensorInfo[] sensorInfos = SensorManager.findSensors("acceleration",
                                                    SensorInfo.CONTEXT_TYPE_USER);
/* Find specific sensor by providing the sensor URL.*/
SensorInfo[] sensorInfos =
            SensorManager.findSensors("sensor:acceleration;contextType=device;model=acc01”);

/*get all sensors available even if their access is restricted*/
SensorInfo[] inf = SensorManager.findSensors(null, null);
```

If there are several sensors measuring the same quantity, the application developer may want to select a specific sensor based on criteria such as accuracy, or a sampling rate. This is done by examining and comparing the information provided by the *SensorInfo* instances.

Some sensors are intended for restricted use only, to be used in the manufacturer, operator, or trusted party domain applications only, or if the user permits. When the application does not have the required permissions, all the found sensors are still returned but they cannot necessary be opened. The *Connector.open()* and *PushRegistry.registerConnection()* methods throw *SecurityException* if the application does not have the required permission to use the sensor.

Monitoring Sensor

SensorManager is also responsible for registering and unregistering *SensorListener* objects. A *SensorListener* will get *sensorAvailable()*/ *sensorUnavailable()* notifications. Only one notification for each matching *SensorListener* is sent per change in availability.

```
//register listener to receive status changed for particular sensor
…
 SensorManager.addSensorListener( new SensorListener() {
          public void sensorAvailable(SensorInfo s) {
             form.append("Sensor available :) " + s.getUrl());
          }
          public void sensorUnavailable(SensorInfo s) {
             form("Sensor unavailable :( " + s.getUrl());
          }
       }, " acceleration ");
…
```

Creating Connection

After selecting the desired sensor based on information in *SensorInfo* objects, the sensor URL can be retrieved from the *SensorInfo* object with the *getUrl()* method.

The sensor URL is used to create the *SensorConnection* via the Connector class. The *SensorConnection* is a Generic Connection Framework (GCF) Connection. The *Connector.open()* method returns a *SensorConnection* instance providing an active connection to the sensor. If the given URL is mapping to multiple sensors then Connector may freely decide which matching sensor to use for the returned *SensorConnection*.

The following code snippet shows how to obtain a SensorConnection.

```
…
try {
//Creating a sensor Connection by providing URL of the selected sensor
SensorConnection sensorConn = (SensorConnection)
                                        Connector.open(sensorInfos[selectedSensor].getUrl());
} catch (IOException e) { e.printStackTrace();  }
…
```

Data Capture

The SensorConnection represents a connection to the real sensor device. SensorConnection is used to get the data from the sensor.

Data can be retrieved by two means:

- Synchronously
- Asynchronously

Synchronous Data

Synchronous data retrieval is done by calling the *SensorConnection.getData(int bufferszie)* method. The *bufferSize* parameter defines the number of samples inside each Data object.

The following code snippet illustrates how to start synchronous data fetching.

```
...
  SensorConnection acceralation =
     (SensorConnection)Connector.open("sensor:accelaration");
  final int bufferSize = 1;
  Data[] data = acceralation.getData(bufferSize);
  for (int i=0, l=data.length; i<l; i++){
      double value = data[i].getDoubleValues()[0];
      form.append("\n channel = "+
      data[i].getChannelInfo().getName()+", value=" + value);
  }
...
```

The measured data is returned as an array of Data objects, where each Data object encapsulates the data of one channel. The channel is a way to structure the data.

Channels represent different dimensions of the measurement. If the sensor measures several values simultaneously, values from each channel are stored to separate Data objects.

If the sensor is measuring just one property, there is only one channel e.g. thermometer. On the other hand, Accelerometer has three channel namely *"axis_x"*, *"axis_y"* and *"axis_z"*.

The Data object contains the information about the channel of its origin, and it can be retrieved with the *Data.getChannelInfo()* method.

The following information is mandatory for all *ChannelInfo* objects:

- **Name**- *unique name of the channel*
- **Accuracy**- *express between <0 - 1>*
- **Data Type**- *TYPE_DOUBLE, TYPE_INT, TYPE_OBJECT*
- **Measurement ranges**- *defined with smallest and the largest possible value and resolution*
- **Scale**- *expressed as exponent of ten*
- **Unit**- *unit in which data value is represented*

Asynchronous Data

The asynchronous mode requires applications to register as *DataListener* objects to *SensorConnection* using *setDataListener(DataListener listener,int bufferSize)* in order to receive *dataReceived()* notifications of collected data. To quit getting notifications, the application must call the method *removeDataListener()*.

When registering a *DataListener* to a *SensorConnection*, you must also provide the size of the buffer. This buffer indicates how many data-values should be collected before sending an event to the *DataListener*. Setting the buffer size to a low value would give you faster updates. However, using a large buffer size you can easily filter out freak values from the accelerometer by calculating the average.

```
public class DataCollecter implements DataListener
{
….
public void connect()
{
 …
 sensor = (SensorConnection) Connector.open(sensorInfos[selectedSensor].getUrl());
 sensor.setDataListener(this, 1);
 ….
}

public void dataReceived(SensorConnection sensor, Data[] data,boolean isDataLost) {
 final string xchannel = "axis_x";
 final string ychannel = "axis_y";
 final string zchannel = "axis_z";
      for (int i = 0; i < data.length; i++) {
         if (xchannel.equals(data[i].getChannelInfo().getName())) {
            double xvalue = data[i].getDoubleValues()[0];
             form.append("channel x value "+xvalue);
         } else if (ychannel.equals(data[i].getChannelInfo() .getName())) {
```

```
            double yvalue = data[i].getDoubleValues()[0];
            form.append("channel y value "+yvalue);
          } else if (channelNames[2].equals(data[i].getChannelInfo().getName())) {
            double zvalue = data[i].getDoubleValues()[0];
            form.append("channel z value "+zvalue);
          }
        }
}
…
}
```

Adding Conditions

Conditions can also be set to sensor. Conditions are used for monitoring the sensor data. The Channel maintains Condition objects attached to the channel of the sensor. There are three conditions provided in Sensor API: *LimitCondition*, *RangeCondition*, and *ObjectCondition*.

ConditionListener should be set to receive condition met notification, when any one of the conditions is met.

LimitCondition

LimitCondition is intended for numeric data to set various kinds of conditions. LimitCondition is set through the use of Operator. The operator has an important role in defining the LimitCondition.

One of the following operators must be used in defining the LimitCondition:

- Condition.OP_EQUALS - equals
- Condition.OP_LESS_THAN_OR_EQUALS - less than or equals
- Condition.OP_LESS_THAN - less than
- Condition.OP_GREATER_THAN - greater than
- Condition.OP_GREATER_THAN_OR_EQUALS - greater than or equals

Examples to set LimitCondition objects to Channel objects

```
channelObj.addCondition(listener, new LimitCondition(50,Condition.GREATER_THAN));
```

The ConditionListener of channelObj will be notified when value is > 50.

```
channelObj.addCondition(listener, new LimitCondition(-45, Condition.LESS_THAN));
```

```
channelObj.addCondition(listener, new LimitCondition(45, Condition.GREATER_THAN));
```

The ConditionListener of channelObj will be notified when value is < -45 or > 45.

RangeCondition

RangeCondition checks if the measured data value is within the defined range. Four types of ranges can be defined with the operators:

- open range, (lowerLimit, upperLimit)
- closed range, [lowerLimit, upperLimit]
- half-closed half-open range, [lowerLimit, upperLimit)
- half-open half-closed range, (lowerLimit, upperLimit]

Examples to set RangeCondition objects to Channel objects

```
channelObj.addCondition(listener, new RangeCondition(lowerLimit,
                    Condition.OP_GREATER_THAN,   upperLimit, Condition.OP_LESS_THAN);
```

ConditionListener will be notified when lowerLimit < value < upperLimit

```
channelObj.addCondition(listener, new RangeCondition(lowerLimit,
                                    Condition.OP_GREATER_THAN_OR_EQUALS,  upperLimit,
                                    Condition.OP_LESS_THAN_OR_EQUALS);
```

ConditionListener will be notified when lowerLimit <= value <= upperLimit

ObjectCondition

The ObjectCondition checks the equality of the set limit to the measured data value. This Condition is intended for a channel, the data type of which is ChannelInfo.TYPE_OBJECT.

The equality is checked by using the equals() method of the set limit. The ObjectCondition MUST be immutable.

```
…
private void addConditions(){
        private static final ObjectCondition ObjCond =
                new ObjectCondition( new Object(){  public boolean equals(Object o){return true;}});
                ChannelInfo[] cInfos = sensor.getSensorInfo().getChannelInfos();
                for (int i=0,l=cInfos.length; i<l; i++){
                    Channel channel = sensor.getChannel(cInfos[i]);
```

```
                    if (cInfos[i].getDataType() == ChannelInfo.TYPE_OBJECT){
                        channel.addCondition(this, ObjCond);
                        continue;
                    }
                }
}
…
```

Sensor Deactivation

Sensor deactivation should be done once your MIDlet no longer requires sensor. Sensor deactivation is done through *removeDataListener()* method in *SensorConnection* class. *removeDataListener()* removes the *DataListener* registered to this *SensorConnection*. The *DataListener* stops getting dataReceived() notifications.

```
public void disconnectSensor() {
…
 if(sensor!=null)  {
  sensor.removeDataListener();
  try {
      sensor.close();
  }catch(IOException e) {
       e. printStackTrace();
  }
 }
…
notifyDestroyed();
}
```

Detecting Sensor API's presence

To check whether the handset supports Sensor API,

```
System.getProperty("microedition.sensor.version").
```

can be used. If it supports, the sensor version is returned, else null will be returned.

Security & Permissions

Some methods in this API are defined to throw a *SecurityException* if the user does not have the permissions needed to perform the action.

This API specifies three permissions to restrict the usage of some sensors. They are:

Private Sensor

Private sensors are sensors that handle confidential data that the user does not want to expose. Examples include sensors related to health, or wealth like a heart rate sensor, or scales.

Protected Sensor

Protected Sensors are system sensors with restricted access to system. System sensors might handle hidden or protected system resources. Data from these sensors is provided only for trusted applications.

Public Sensor

Sensors that are not private or protected are understood as public sensors.

All these sensor groups, which are private, protected, and public, require the *javax.microedition.io.Connector.sensor* permission to open the connection.

Table 4 defines the names of three sensor related permissions and the methods that are protected by each permission.

Table 5: Permissions

Permissions name	Methods protected by this Permission
javax.microedition.sensor.PrivateSensor	Connector.open() PushRegistry.registerConnection()
javax.microedition.sensor.ProtectedSensor	Connector.open() PushRegistry.registerConnection()
javax.microedition.io.Connector.sensor	Connector.open()

PushRegistry

PushRegistry can be used with Sensor for automatic launch of application. When the application is registered, the sensor URL containing the conditions for launch is specified together with the application to be launched.

The application management software (AMS) monitors the defined sensors. When the sensor becomes available or one of the monitored conditions is met, the AMS finds the application to be woken up and launches it.

The AMS stores the data value that caused the automatic launch of the application when the monitored condition was met, as well as the met condition. The validity, uncertainty and timestamp information is also stored.

PushRegistry can be used in two ways:

- Availability-based push
- Condition-based push.

Availability-based push

Sensor application is launched automatically when the sensor becomes available. Some sensors may not support availability-based push mechanism. For example, some sensors might always be available but may not support availability-based push. Whether the sensor supports availability-based push can be found out with the method *SensorInfo.isAvailabilityPushSupported()*

The MIDlet registered statically in the JAD for availability-based push is shown below.

```
MIDlet-Push-1: sensor:acceleration,sample.PushAcclApp,*
MIDlet-Permissions: javax.microedition.io.PushRegistry
```

Condition-based push

Conditions are set per channel. After any one condition of any channel is met, the registered application is launched.

The sensor URL scheme for the PushRegistry does not define conditions for channels of an Object data type.

Whether a sensor supports condition-based push can be found out with the method *SensorInfo.isConditionPushSupported()*.

The MIDlet registered statically in the JAD for condition-based push is shown below.

```
MIDlet-Push-1: sensor:RR_interval?channel=RR_interval&limit=240&op=le,
sample.PushCondApp,*
MIDlet-Permissions: javax.microedition.io.PushRegistry,
javax.microedition.sensor.PrivateSensor
```

The AMS system checks periodically the availability of supported RR_interval sensors around. If RR_interval sensor is available, then the defined conditions are checked. If any one of the conditions is met, the application is launched. In the above example, the measured value being less than or equal to 240 ms causes the application to be launched.

Sensor Example

This sample example shows how to use Sensor API for getting sensor information.

Class: SensorMIDlet

```
import java.io.IOException;
import java.util.Vector;

import javax.microedition.io.Connector;
import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.CommandListener;
import javax.microedition.lcdui.Display;
import javax.microedition.lcdui.Displayable;
import javax.microedition.lcdui.List;
import javax.microedition.lcdui.Canvas;
import javax.microedition.midlet.MIDlet;

import javax.microedition.sensor.ChannelInfo;
import javax.microedition.sensor.Data;
import javax.microedition.sensor.DataListener;
import javax.microedition.sensor.SensorConnection;
import javax.microedition.sensor.SensorInfo;
import javax.microedition.sensor.SensorManager;

public class SensorMIDlet extends MIDlet implements DataListener, CommandListener {
 private static final Command CMD_SELECT =  new Command("Select",  Command.SCREEN, 1);
 private static final Command CMD_EXIT =    new Command("Exit", Command.EXIT, 1);

 private SensorConnection sensor;
 private String[] channelNames;
 private SensorInfo[] sensorInfos;
 private boolean sensorSelected;

 private List sensorSelector;
 private SensorCanvas sensorCanvas;

 /* Initialize MIDlet and detect accelerometer sensor.  */
 public SensorMIDlet() throws IOException {
       if (System.getProperty("microedition.sensor.version") == null) {
          throw new IllegalArgumentException("JSR256 is not supported!");
       }
       sensorCanvas =  new SensorCanvas(this);
        sensorInfos = getSensorInfos();
```

```
    if (sensorInfos == null) {
        throw new IllegalArgumentException(
                "Valid accelerometer sensor not found");
    }
    sensorSelector = new List("Select accelerometer:", List.EXCLUSIVE);
    for (int i = 0; i < sensorInfos.length; i++) {
        sensorSelector.append(sensorInfos[i].getUrl(), null);
    }
    sensorSelector.addCommand(CMD_SELECT);
    sensorSelector.addCommand(CMD_EXIT);
    sensorSelector.setCommandListener(this);
}

public void startApp() {
    if (!sensorSelected) {
        Display.getDisplay(this).setCurrent(sensorSelector);
    } else {
        Display.getDisplay(this).setCurrent(sensorCanvas);
        sensor.setDataListener(this, 1);
    }
}

public void pauseApp() {
    sensor.removeDataListener();
}

public void destroyApp(boolean unconditional)  {
  if (sensor != null) {
      try {
        sensor.removeDataListener();
         sensor.close();
      } catch (IOException ex) {
         ex.printStackTrace();
      }
    }
    notifyDestroyed();
}

public void commandAction(Command c, Displayable d) {

    if (d.equals(sensorSelector))  {
        if (CMD_SELECT.equals(c)) {
            new Thread()  {
                public void run()  {
```

```
                int idx = sensorSelector.getSelectedIndex();
                channelNames = getSensorInfoChannelsNames(sensorInfos[idx]);
                try {
                  sensor = (SensorConnection) Connector.open(sensorInfos[idx].getUrl());
                  sensorSelected = true;
                  Display.getDisplay(SensorMIDlet.this).setCurrent(sensorCanvas);
                  sensor.setDataListener(SensorMIDlet.this, 1);
                } catch (IOException e) {e.printStackTrace();}
              }
            }.start();
        } else if (CMD_EXIT.equals(c)) {
          destroyApp(false);
          notifyDestroyed();
        }
    }
}

/* pass acceleration data to the sensorCanvas.*/
public void dataReceived(SensorConnection sensor, Data[] data, boolean isDataLost) {
    double accelX = 0;
    double accelY = 0;
    double accelZ = 0;
    for (int i = 0; i < data.length; i++) {
        if(channelNames[0].equals(data[i].getChannelInfo().getName())) {
          accelX = data[i].getDoubleValues()[0];
        }else
        if(channelNames[1].equals(data[i].getChannelInfo().getName())) {
          accelY = data[i].getDoubleValues()[0];
        }else
        if(channelNames[2].equals(data[i].getChannelInfo().getName())) {
          accelZ = data[i].getDoubleValues()[0];
        }
    }
    sensorCanvas.setValues(accelX, accelY, accelZ);
}

/* Detect appropriate sensor info. */
private SensorInfo[] getSensorInfos() {
    /* Find all device accelerometers*/
    SensorInfo[] sensorInfos =
      SensorManager.findSensors("acceleration",SensorInfo.CONTEXT_TYPE_DEVICE);

    Vector validInfos = new Vector();
    for (int i = 0; i < sensorInfos.length; i++) {
```

```
            if (getSensorInfoChannelsNames(sensorInfos[i]) != null) {
                validInfos.addElement(sensorInfos[i]);
            }
        }
        SensorInfo[] ret = null;
        if (validInfos.size() > 0) {
            ret = new SensorInfo[validInfos.size()];
            validInfos.copyInto(ret);
        }
        return ret;
    }

    /* Detect valid channel names for sensor.    */
    private String[] getSensorInfoChannelsNames(SensorInfo sensorInfo) {
        Vector channelNames = new Vector();
        ChannelInfo[] channelInfos = sensorInfo.getChannelInfos();
        if (channelInfos.length > 1) {
            /* Accelerometer must support at least 2 channels*/
            for (int i = 0; i < channelInfos.length; i++) {
                if (ChannelInfo.TYPE_DOUBLE == channelInfos[i].getDataType()) {
                    /* The channel type must be double*/
                    channelNames.addElement(channelInfos[i].getName());
                }
            }
            if (channelNames.size() > 2) {
                String[] names = new String[3];
                names[0] = (String) channelNames.elementAt(0);
                names[1] = (String) channelNames.elementAt(1);
                names[2] = (String) channelNames.elementAt(2);
                return names;
            }
        }
        return null;
    }
}
```

Class: SensorCanvas

```
import java.io.IOException;

import javax.microedition.lcdui.Canvas;
import javax.microedition.lcdui.Graphics;
import javax.microedition.lcdui.Image;

public class SensorCanvas extends Canvas implements Runnable {
```

```
    private double  xChannel, yChannel, zChannel;
    private SensorMIDlet midlet;

    public SensorCanvas(SensorMIDlet midlet)  {
            this.midlet = midlet;
        setFullScreenMode(true);
        new Thread(this).start();
    }

    public void run()  {
        try   {
            while (true)  {
                repaint();
                Thread.sleep(100);
            }
        }catch (InterruptedException e) {
        }
    }

    public void paint(Graphics g)  {
        g.setColor(0xffffff);
        g.fillRect(0, 0, getWidth(), getHeight());
        g.setColor(0);
        g.drawString("Accelorometer Values",getWidth()>>1,10,17);
        g.drawString("X Channel "+xChannel,getWidth()>>1,30, 17);
        g.drawString("Y Channel "+yChannel,getWidth()>>1,50, 17);
        g.drawString("Z Channel "+zChannel,getWidth()>>1,70, 17);
        g.drawString("Exit",2,getHeight()-15,20);
    }

    public void setValues(double x, double y, double z)     {
        xChannel = x;
        yChannel = y;
        zChannel = z;
    }

    public void keyPressed(int keyCode)  {
            midlet.destroyApp(true);
    }
}
```