How to drive radio-controlled car with Nokia N97 or Nokia 6260 Slide devices

Introduction

05 Jul 2009

Several projects have featured a radio-controlled (RC) car that was controlled with a mobile phone rather than a traditional radio controller. Commonly, communication between the car and the mobile device was done using a Bluetooth connection, and the commands sent to the car were based on an accelerator sensor in the mobile device. The radio part in the car was replaced with a specifically made Bluetooth module that received the commands from the mobile device.

This document introduces two different solutions for controlling an RC car with a mobile device. The first one uses the Nokia N97 device, which supports the standard Mobile Sensor API (JSR-256). The second uses the Nokia 6260 slide device, where a joystick key is used to control the car. In both cases the car is controlled by using a small Mobile Java™ application that creates and sends the controlling messages to the car.

The main purpose of this document is to describe the software portion of the project—how the controller MIDlets were created, tested, and used. Readers should have at least a basic knowledge of Java programming and how to create MIDlets.

About the MiniController project

This project was started at the end of March 2009, roughly two months before the planned demonstration at the first-ever Nokia Developer Summit event. The event took place the 28th-29th of May in Monaco. The intention was to have a demo stand to demonstrate control of the RC car with a mobile device and also to share the latest information about S60 and Series 40 Mobile Java-related features.

The project consisted of building the car, and writing, testing, and adjusting the controller MIDlets. The total time spent was roughly one week but the work was spread out over two months.

About the car

The car that was used was a standard model made by Tamiya: the RC MINI Cooper S 2006. No changes to the car were made, except the radio module was replaced with the additional Bluetooth module.

The Bluetooth module was ordered from Upper Austria University of Applied Sciences, from the department of Mobile Computing (MC).

Figure 1: Building the car

(C) Copyright Nokia 2013. All rights reserved.

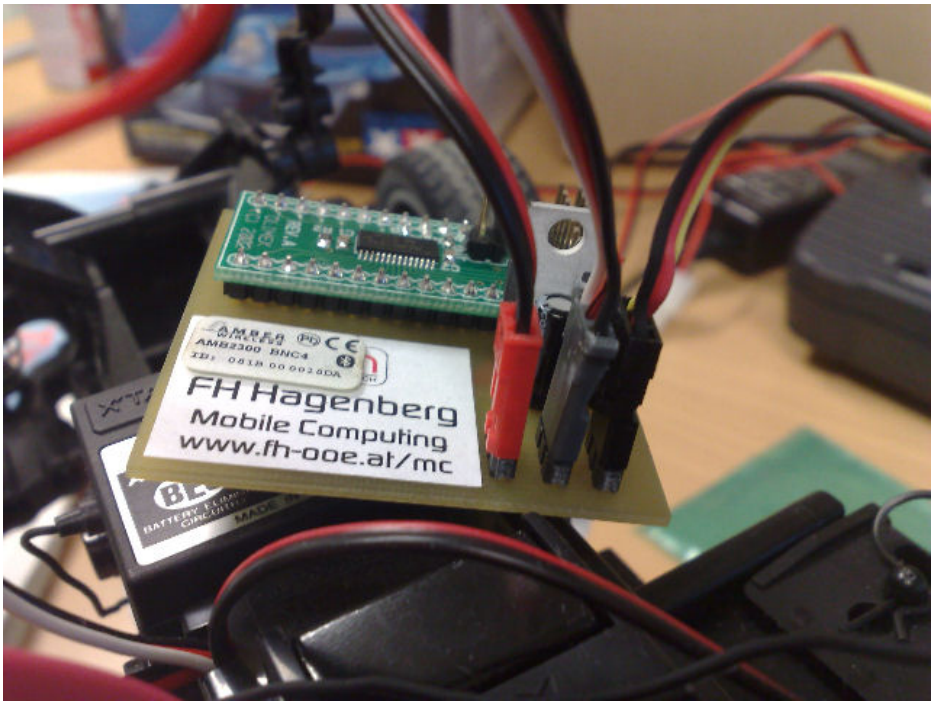


Figure 2: The Bluetooth module connected to the car

About the controller MIDlets

There are two separate MIDlets to control the car, for each of the (two) mobile phones used. These devices share most of the code, but because of the differences in hardware it was justified to write two separate MIDlets. The Nokia N97 device has a Mobile Sensor API (JSR-256), a touchscreen, and an nHD (640 x 360 pixels) screen size, while the Nokia 6260 slide device has no JSR-256 support, no touchscreen, and a QVGA screen. The Nokia 6260 slide device has a force-sensitive joystick[5], which is ideal for controller use.

In both MIDlets, messages to be sent to the car are generated based on user input, and they are sent to the car over Bluetooth connection.

Figure 3: Nokia N97 and Nokia 6260 slide devices

MiniController MIDlet for the Nokia N97 device

The most important issue in the project was to demonstrate the new Mobile Sensor API support in Nokia devices. The Nokia N97 device was the first to have this API as a built-in feature. There are actually four sensors supported by the API in the Nokia N97 device: accelerometer, battery charge level sensor, charger state sensor, and network signal level sensor.

The MiniController MIDlet uses the accelerometer for tracking the device position, and based on that information creates the controller messages, which are then sent to the car.

The MIDlet consists of six classes:

Class	Main purpose
MiniController.java	The main MIDlet class.
LogCanvas.java	Canvas screen for showing output, for example from Bluetooth device discovery and service discovery.
SensorCanvas.java	Actual Canvas screen for showing the sensor values, creating the Bluetooth connection, creating the messages, and sending the messages to the car. This screen also has some adjustment buttons for setting the offsets for controlling values.
DeviceDiscoverer.java	Executes the device inquiry as specified in the Bluetooth API (JSR-82). Note: Currently not used because this only needs to be done once.
ServiceDiscoverer.java	Executes the service search as specified in the Bluetooth API (JSR-82). Note: Currently not used because this only needs to be done once.
SettingsForm.java	A Form for entering different kind of settings. Currently this feature is not used.

Table 1: MiniController MIDlet classes

Main features of the MIDlet

(c) Copyright Nokia 2013. All rights reserved.

In the beginning of the development work, the MIDlet was used to search the Bluetooth devices as well as the services in them. During the course of developing the application it came clear that device inquiry and service search would not be necessary to do again, once the exact URL to the correct service has been found. That also made `LogCanvas` quite useless. However, these features were saved because they are needed if the Bluetooth module is replaced with a new one (or somebody else is using the MIDlet for his/her testing).

In the Nokia N97 device it is possible to get the accelerometer values, which indicate the position of the mobile phone. Based on information gathered from similar projects, controlling the car by tilting the mobile device is very intuitive. Also, the data received from the sensor is quite accurate, and it is received quickly enough for the controlling to work very well.

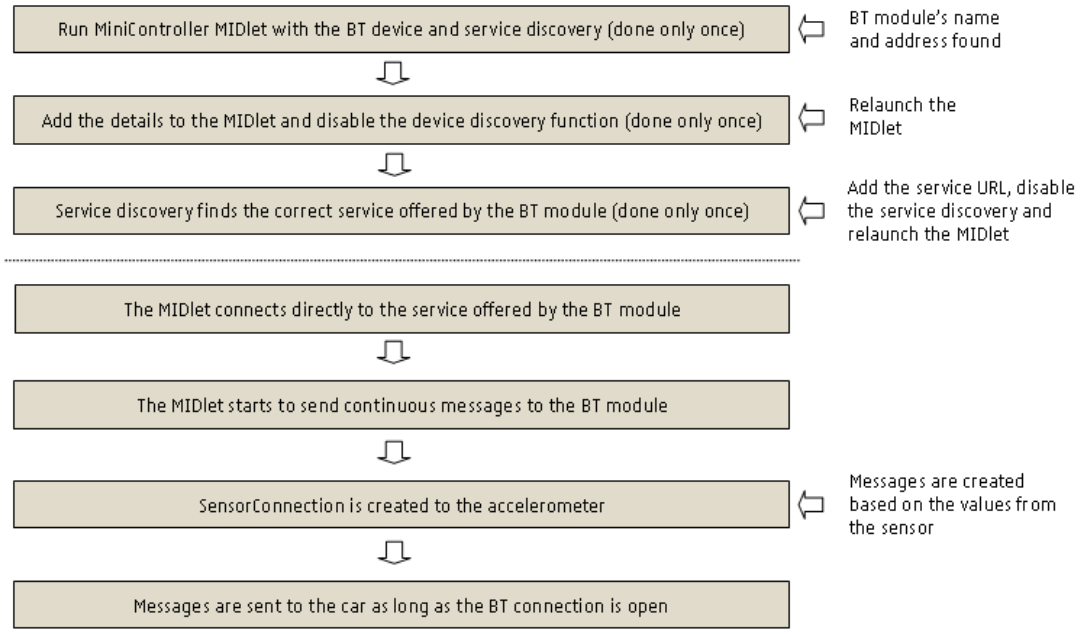


Figure 4: Simplified process flow of the MiniController MIDlet

Using the Mobile Sensor API

The Mobile Sensor API (JSR-256) is an optional API that offers a unified way of managing sensors connected to the mobile devices, and easy access to the sensor data. The first device from Nokia to support this API is the Nokia N97 mobile phone. Currently (April 2009) there is one Nokia SDK available that supports this API: the Nokia N97 SDK 0.5.

Using the API makes it possible to find information about the sensors supported by the Mobile Java implementation. The Nokia N97 device shows information about two different accelerometers, one returning sensor values as integers and another returning double values. In this case, integer values were used. The method below shows how the sensor selection is done. The `findSensors()` method returns `SensorInfo` objects of all the sensors of type 'acceleration'. Then the sensor's data type is checked and the sensor with TYPE_INT data type is selected.

Generally, if the sensor is measuring different properties or dimensions simultaneously, those values are considered as separate channels. The data types are actually features of the channels. So, in the case of the accelerometer, the sensor is returning three values, one from each channel (axis_x, axis_y, and axis_z). In our case we want to use the accelerometer, which returns integer values (Channel's data type is TYPE_INT).

```
private SensorConnection openSensor() {
 infos = SensorManager.findSensors("acceleration", null);
 if (infos.length==0) return null;
 int datatypes[] = new int[infos.length];
 int i = 0;
 String sensor_url = "";
 while (!sensor_found) {
  datatypes[i] = infos[i].getChannelInfos()[0].getDataType();
  if (datatypes[i] == 2) { //ChannelType.TYPE_INT = 2
   sensor_url = infos[i].getUrl();
   sensor_found = true;
  }
  else i++;
 }
 try {
  return (SensorConnection)Connector.open(sensor_url);
 }catch (IOException ioe) {
 ioe.printStackTrace();
  return null;
 }
}
```

Example 1: The openSensor() method in the SensorCanvas class, which is used for finding the correct sensor and for creating the SensorConnection

When the correct sensor is found and the connection to it is established, `DataListener` is set for listening to the sensor values. Note the BUFFER_SIZE as a parameter for the `setDataListener()` method. In this case, a size of 4 was used. The registered `DataListener` receives data within sequential `dataReceived()` notifications. The notification is sent when the number of collected data values equals a set size of the buffer. The buffer size was based on testing; the idea was to limit the amount of data to be sent to the Bluetooth module. The buffer values also made it possible to easily calculate an average of the buffer values, which were then used as actual values for the controlling.

The `DataListener`'s state is controlled by using a boolean variable isStopped. When this variable gets value 'true', the `DataListener` is removed and `SensorConnection` is closed.

```
private synchronized void initSensor() {
 sensor = openSensor();
```


(C) Copyright Nokia 2013. All rights reserved.

```
 if (sensor == null) return;
 try {
  sensor.setDataListener(this, BUFFER_SIZE);
  while(!isStopped){
   try {
    wait();
   } catch(InterruptedException ie){}
  }
  sensor.removeDataListener();
 }catch (IllegalMonitorStateException imse) {
  imse.printStackTrace();
 }catch (IllegalArgumentException iae) {
  iae.printStackTrace();
 }
 try {
  sensor.close();
 } catch(IOException ioe){
  ioe.printStackTrace();
 }
 if (isStopped) {
  sensor = null;
 }
}
```

Example 2: The initSensor() method in the SensorCanvas class, which calls the openSensor() method for opening SensorConnection, sets the DataListener, and takes care of closing the connection

When all the preparations have been made, `dataReceived()` notifications are coming continuously. The data received from the `dataReceived()` method consists of the `SensorConnection`, where the data is coming from, actual data as a Data object array, and a boolean value, which tells us if some data is lost. In our case, only the actual data is used.

An average value for each direction is calculated and then the messages are created.

```
/**
 * Notification of the received sensor data.
 * @param sensor - SensorConnection, the origin of the received data
 * @param data - the received sensor data
 * @param isDataLost - true if some data has been lost
 */
public void dataReceived(SensorConnection sensor, Data[] data, boolean isDataLost) {
 int[] directions = new int[3];
 for (int i = 0; i < data.length; i++) {
  int values[] = data[i].getIntValues();
  int temp = 0;
  for (int j = 0; j < values.length; j++) {
   temp = temp + values[j];
  }
  temp = temp / BUFFER_SIZE;
  directions[i] = temp;
 }
 x_int = directions[0];
 y_int = directions[1];
 z_int = directions[2];
 createMessageData();
 repaint();
}
```

Example 3: The dataReceived() method in the SensorCanvas class, which is called when the number of collected data values equals a set size of the buffer (BUFFER_SIZE)

Using the Bluetooth API

Currently the MIDlet creates the `StreamConnection` directly to the service available in the car's Bluetooth module. The correct device was searched by using the standard `DiscoveryAgent.startInquiry()` method:

```
try {
 LocalDevice localDevice = LocalDevice.getLocalDevice();
 DiscoveryAgent agent = localDevice.getDiscoveryAgent();
 agent.startInquiry(DiscoveryAgent.GIAC, this);

catch(BluetoothStateException bse) {
 bse.printStackTrace();
}
```

Example 4: Code demonstrating how the device inquiry can be done

The device names and addresses were printed out in `deviceDiscovered()` method. The `RemoteDevice` found was added to a `Vector` (remoteDevices) for later use.

```
public void deviceDiscovered(RemoteDevice btDevice, DeviceClass cod) {
 String address = btDevice.getBluetoothAddress();
 String name = "";
 try {
  name = btDevice.getFriendlyName(false);
 } catch (IOException ex) {
  ex.printStackTrace();
 }
 midlet.logCanvas.addString(name + "(" + address + ")");
 if (address.equals("0018DA00081B")) {
  midlet.logCanvas.addString("0018DA00081B found!");
  agent.cancelInquiry(this);
 }
 remoteDevices.addElement(btDevice);
}
```

Example 5: The deviceDiscovered() method in the DeviceDiscoverer class, which is called when a device is found during an inquiry

Similarly, the `LocalDevice.searchServices()` method was used for finding the services. The correct URL was found by using the `ServiceDiscoverer` class (implementing `DiscoveryListener`) and its `serviceSearchCompleted()` method. This method is called when the service search is completed. In this method, the MiniController class' method searchCompleted() is called, with the `ServiceRecord` as its parameter. The `searchCompleted()` method goes

(c) Copyright Nokia 2013. All rights reserved.

through the `ServiceRecord` array, which in this case consists of only one service. The URL is found by using the `ServiceRecord.getConnectionURL()` method:

```
for (int i = 0; i < servRecord.length; i++) {
 url = servRecord[i].getConnectionURL(ServiceRecord.AUTHENTICATE_NOENCRYPT, false);
 logCanvas.addString("URL: " + url);
}
```

Example 6: Code demonstrating how the service URL is found

Note: As mentioned earlier, the DeviceDiscoverer and ServiceDiscoverer classes are not used in the MiniController application. They can be taken in to use for searching devices and services by simply removing '//' from the beginning of the commented lines in two locations:

1. In the LogCanvas class, paint() method, one line:
   //midlet.startDeviceDiscovery(); // NOT NEEDED, IF THE BT SERVICE URL IS KNOWN.
2. In the MiniController class, inquiryCompleted() method, lines:
   //serviceDiscoverer = new ServiceDiscoverer(this); // NOT NEEDED, IF THE
   //serviceDiscoverer.startServiceSearch(devices[index]); // URL IS KNOWN

Also, in the MiniController class, the following variable needs to be changed when the correct device name and service URL have been found:
protected String device = "BNC4-081B000018DA";
protected String url = "btspp://0018DA00081B:1;authenticate=true;encrypt=false;master=false";

Creating the messages

After the connection has been created, the MIDlet starts to send messages to the car. There are separate messages for accelerating and steering, and they are sent continuously in a separate thread. Between each sending, a short delay of 50 ms is held.

The details of the messages are explained in Figure 11 in [1]. In brief, the message consists of six bytes, as shown below:

- Preamble: 1 byte, always 0xFF, used to identify the start of a frame. DeviceID, MessageID, and DataLength must not be 0xFF.
- DeviceID: 1 byte, identifies the device, in this case either 3 (steering) or 4 (acceleration).
- MessageID: 1 byte, identifies the message type a device sends or receives; either 0 for receiving from the car or 1 for sending to the car.
- DataLength: 1 byte, indicates how many data bytes will follow (0 – 255).
- Data: 0 – 255 bytes.
- Checksum: 1 byte, calculated by performing a modulo 256 sum over the entire packet (preamble, DeviceID, MessageID, DataLength, and data).

There is a known feature in Java programming language, that its Bytes are signed values from -128 to 127. As shown above, the preamble byte must be 0xFF! After some testing with byte and integer arrays, it seemed that using the `DataOutputStream.write(int b)` method works nicely. As listed in the MIDP 2.0 javadoc the method 'writes the specified byte (the low eight bits of the argument b) to the underlying output stream', and the parameter b is the byte to be written, even if it is given as an integer.

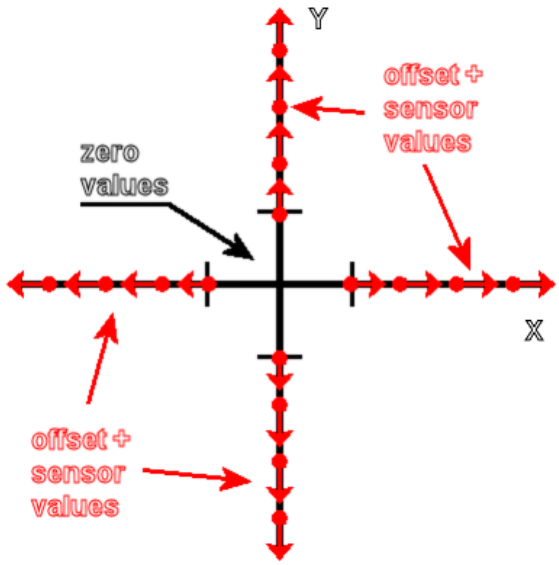
A message (array of six integers) is created simply in one method as shown below:

```
private int[] createIntMessage(int device, int data) {
    int[] message = new int[6];
    message[0] = preamble; // preamble, always 0xFF
    message[1] = device; // deviceID, steering (=4) or accelerator (=3)
    message[2] = 1;// messageID, message type: 0 = receives, 1 = sends
    message[3] = 1;    // data length, in this case always 1
    message[4] = data; // actual data
    int checksum = (int)((message[0] + message[1] + message[2] + message[3] + message[4])%256);
    message[5] = checksum;     // checksum
    return message;
}
```

Example 7: The createIntMessage() method in the SensorCanvas class, which creates the message of six integers

The `createIntMessage()` method is called in the `createMessageData()` method, which controls and adjusts the accelerating and steering values to be within the 'correct' limits. For this project it was chosen that there is a certain area around zero, where only zero values are sent to the car. If the values are outside this 'zero area', they are handled properly. It is also checked in the `createMessageData()` method if the car is moving forward or backward.

Figure 4 shows the principle of the value generation. Note that the origo values are not actually zero. It was clarified by adjusting the offset values with touchscreen buttons, which are the 'zero values', that is, when the tires are not going forward or backward and when they are pointing exactly straight ahead. These zero values are sent as 'zero messages', when the controller device is, for example, lying on the table screen facing upwards.



(C) Copyright Nokia 2013. All rights reserved.

Figure 5: The principle of generating the accelerating and steering values

Below is the code of the `createMessageData()` method, which is used for creating the data messages. The variables acc_offset and dir_offset are the zero values, when the car is not moving and the tires are steered exactly straight ahead. The offset value is defining the area limits, where the zero messages are sent. Speed for going backward is set to be constant.

```
/**
 * Creates the message data for both accelerating and steering. If z_int
 * or y_int (from sensor) is close to the "zero value" (-offset <->
 * offset), "zero message" is created. Otherwise the value is based on
 * combination of offset and the sensor value.
 */
private void createMessageData() {
 // Code for using x values for accelerating
 if (x_int < 0) forward = true;
 else forward = false;
 if (x_int > -(2*offset) && x_int < (2*offset)) {
  if (!forward && back) {
   accelerating = acc_offset + 15;
  }
  else accelerating = acc_offset;
 }
 else {
  if (forward) accelerating = acc_offset + x_int/3; // Limited speed
  else accelerating = acc_offset + 15; // Backwards by tilting
  if (accelerating > 120) accelerating = 120;
  if (accelerating < -120) accelerating = -120;
 }
 if (y_int > -offset && y_int < offset) {
  steering = dir_offset;
 }
 else {
  steering = dir_offset + y_int;
  if (steering > 120) steering = 120;
  if (steering < -120) steering = -120;
 }
 dir_int_message = createIntMessage(STEERING, steering);
 acc_int_message = createIntMessage(ACCELERATOR, accelerating);
}
```

Example 8: The createMessageData() method in the SensorCanvas class, which creates the message data and calls the createIntMessage() method for actually creating the messages

There are future plans for improving the speed and steering control. Currently it seems that trying to go forward and steer simultaneously doesn't work too well. Also it would be nice to have better control of adjusting the maximum speed. One improvement idea is to use nonlinear values for steering (and maybe for accelerating), which could replace the current solution of using offsets. (Nonlinear values were also used in the ShakerRacer project [1].)

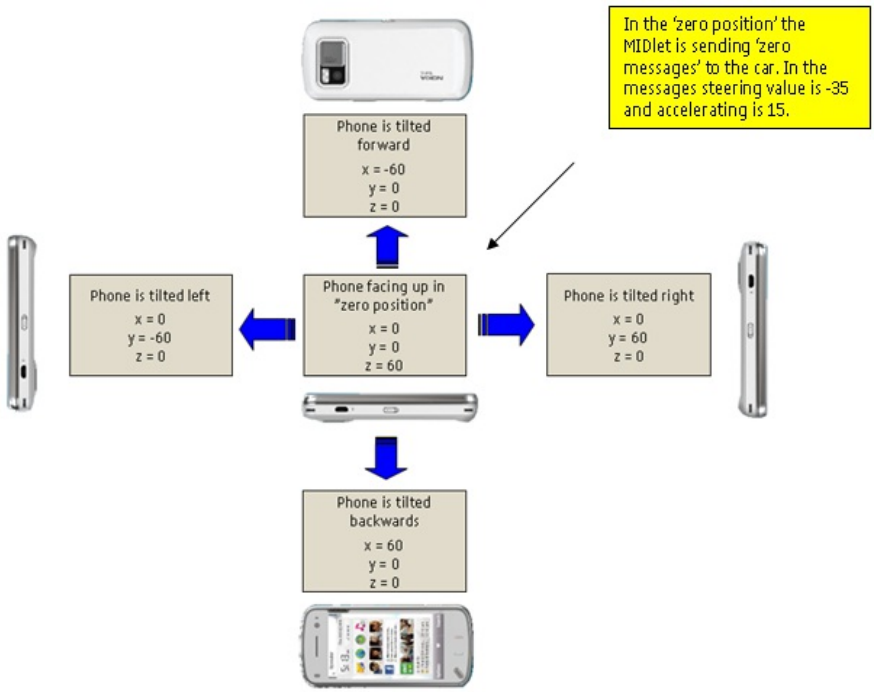


Figure 6: Roughly the maximum values of the accelerometer sensor, when the Nokia N97 device is tilted

Figure 6 shows the rough maximum values of the sensor when the Nokia N97 device is tilted. Note that the values are varying all the time, and the values are not exactly the ones shown in the figure. By testing and adjusting the zero message values it was found that when the car was not moving and the tires were pointing directly straight ahead, the steering value should be roughly -35 and the accelerating value should be roughly 15. It is not known why these values are so far away from zero. The similar maximum values for the car vary case by case and can be determined only by testing. For this purpose there are four buttons in the MiniController MIDlet screen. The offsets for steering and accelerating can be adjusted by using the buttons.

When the zero values are found, it is time to search for suitable maximum values for steering and accelerating forward and backward.

The MIDlet main screen (`SensorCanvas`) is shown below. The screen view consists of the following:

- The actual sensor values (INT x, INT y, INT z values);
- The steering and accelerating values sent to the car;

(C) Copyright Nokia 2013. All rights reserved.

- The offset values for steering and accelerating;
- Red/yellow indicator showing graphically the sensor movement;
- Touchscreen buttons for starting and stopping (= creating the BT connection and closing it);
- Touchscreen buttons for adjusting the offsets (Acc-, Acc+, Dir-, Dir+);
- Touchscreen button for exiting the MIDlet.

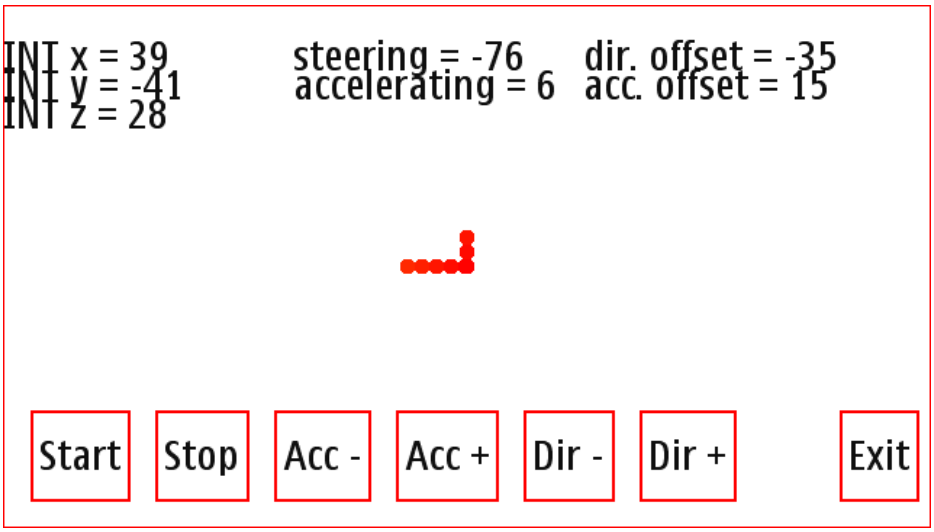


Figure 7: SensorCanvas screenshot

In the `SensorCanvas` class (which implements the Runnable interface) the messages are written to the output stream:

```
public void run() {
 while (!isStopped) {
  if (acc_int_message == null) acc_int_message =
          createIntMessage(ACCELERATOR, acc_offset);
  if (dir_int_message == null) dir_int_message =
          createIntMessage(STEERING, dir_offset);
  try {
   for (int i = 0; i < acc_int_message.length; i++) {
    dataOutputStream.write(acc_int_message[i]);
   }
   dataOutputStream.flush();
   for (int i = 0; i < dir_int_message.length; i++) {
    dataOutputStream.write(dir_int_message[i]);
   }
   dataOutputStream.flush();
  } catch (IOException ioe) {
   ioe.printStackTrace();
  }
  try {
   Thread.sleep(DELAY); // Delay of 50 ms
  } catch (InterruptedException ie){}
 }
}
```

Example 9: The run() method in the SensorCanvas class, which takes care of sending the controlling messages to the car

S40Controller MIDlet for Nokia 6260 slide

Another interesting way of controlling the RC car is to use the Nokia 6260 slide device, because it has a force-sensitive joystick. It is possible to get x and y values by pressing the key edges, and even in such a way that the value depends on the pressing power. Getting the values is very easy—simply import one additional class from the Nokia UI API: `com.nokia.mid.ui.JoystickEventListener`. Note that this class is an optional feature in the Nokia UI API, and depending on the device capabilities there may be implementations that do not support joystick events. Currently (May 2009) only the Nokia 6260 slide device supports this feature. The `System.getProperty("com.nokia.mid.ui.joystick_event")` will return 'true' if this feature is available in the device.

The `JoystickEventListener` class has only one method, `joystickEvent(int x, int y)`, which is called when the Joystick key sends an event. The key events are relative to its original position (0, 0), and the maximum and minimum values of x and y are -127 and 127. As an example, suppose that the (5, 5) event was received. It means that the joystick is held in the left and up diagonal. Now, if the (-25, 25) event is sent, it means that the joystick is still in the same direction of (-5, 5), however more force was applied.

(C) Copyright Nokia 2013. All rights reserved.

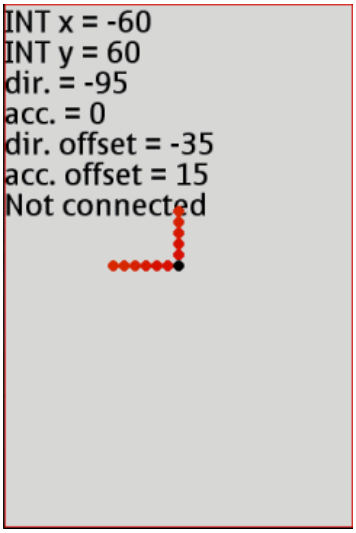


Figure 8: A screenshot of the S40Controller MIDlet in the S40 6th Edition SDK

The S40Controller MIDlet shares a lot of code with the MiniController MIDlet. It is actually much smaller, because using the `JoystickEventListener` was so simple. It also has similar Bluetooth-related classes, which are normally not used.

The most important parts of the code are listed below:

```
    public void joystickEvent(int x, int y) {
        x_int = x;
        y_int = y;
        createMessageData();
        repaint();
    }
```

Example 10: The joystickEvent() method in the ControlCanvas class, which handles the joystick key events

```
    private void createMessageData() {
        if (y_int > -offset && y_int < offset) {
            accelerating = acc_offset;
        }
        else {
            accelerating = acc_offset - y_int/4;
            if (accelerating > 35) accelerating = 35;
            if (accelerating < 0) accelerating = 0;
        }
        if (x_int > -offset && x_int < offset) {
            steering = dir_offset;
        }
        else {
            steering = dir_offset + x_int;
            if (steering > 30) steering = 30;
            if (steering < -95) steering = -95;
        }
        dir_int_message = createIntMessage(STEERING, steering);
        acc_int_message = createIntMessage(ACCELERATOR, accelerating);
    }
```

Example 11: The createMessageData() method in the ControlCanvas class, which creates the steering and accelerator messages

The `createMessageData()` method works in a similar way as in the MiniController MIDlet. There is a certain offset, when 'zero messages' are sent to the car and values outside this area are really handled as controller values. Maximum values to be sent to the car have been searched by testing. They vary in other hardware combinations (other cars, other mobile phones, and other BT modules), so thorough testing is always mandatory. The variables acc_offset and dir_offset are the zero values, when the car is not moving and the tires are steered exactly straight forward.

Future plans for the project

The MiniController project was undertaken in a limited timeframe. There were some issues that were not addressed at the time that might be useful and interesting to take a look at. For example, in the ShakerRacer project, nonlinear steering was implemented to make it easier to drive directly straight ahead. A similar kind of nonlinear accelerating might be good for this project.

Currently the MiniController MIDlet has simply a certain area around zero, when only 'zero values' are sent to the car in the steering and accelerating messages.

It should also be possible to get messages from the car—values of engine revolutions per minute and current speed, at least. Receiving messages from the car would require some additional Bluetooth-related code, but otherwise should be easy to implement.

Another interesting idea would be to record the sent messages and try to redrive the same route again without anyone controlling the car. Also, it should be possible to use the WLAN network instead of Bluetooth for lengthening the maximum distance between the controlling mobile phone and the car.

Contacting the project creator

The project was undertaken mainly by one person, Jarmo Lahtinen, chief engineer in the Nokia Devices R&D unit, in S60 Java organisation. His work focuses on different kinds of projects related to MIDlet development tools, developer documentation, and cooperation with other organisations inside Nokia,

(C) Copyright Nokia 2013. All rights reserved.

for example Nokia Developer.

To send feedback, comments, or questions about the project, use the following e-mail address: jarmlaht@ovi.com.

References

- [1] ShakerRacer: Drive Your Car with Your Phone (<link broken "Technical_Background_of_ShakerRacer.pdf")
- [2] JSR 256: Mobile Sensor API Specification (http://www.jcp.org/en/jsr/detail?id=256)
- [3] How To Get Information about Sensors in Java ME (How_to_get_information_about_sensors_in_Java_ME)
- [4] Video of Using Mobile Sensor API for Controlling RC Car in Nokia Developer Summit 2009 in Monaco (http://share.ovi.com/media/jarmlaht.public/jarmlaht.10005)
- [5] Developing Java ME Applications with Force-Sensitive Joystick and Force-Sensitive Joystick API (Series_40_6th_Edition:_Developing_Java_ME_applications_with_force-sensitive_joystick_and_Force_Sensitive_Joystick_API)

See also

- MiniController_project_1_0.pdf This wiki article as downloadable pdf file
- MiniController.zip The MiniController MIDlet as NetBeans project (For Nokia N97 and other S60 5th Ed. devices supporting JSR-256)
- S40Controller.zip The S40Controller MIDlet as NetBeans project (For Nokia 6260 Slide)
- Mobile Sensor API (JSR-256) javadoc documentation and RI Binary
- How to get information about sensors in Java ME
- How to get accelerometer sensor values in Java ME
- How to use sensors in Java ME
- Mobile Sensor API (JSR-256) beta add-on for Nokia 5800 XpressMusic
- Nokia N97 SDK 0.5

(C) Copyright Nokia 2013. All rights reserved.

(C) Copyright Nokia 2013. All rights reserved.