Programming Lego Robots with LeJOS

Table of Contents

1. Introduction

This document describes a few experiments which I did to get myself acquainted with LeJOS - the Java Operating System for the Mindstorm Robotic Invention System. I hope that somebody might find it useful when they start playing with LeJOS. The document is written with the assumption that the reader is comfortable with Java.

2. What is LeJOS

The ‘brain’ of the Lego Mindstorm Robotic Invention system is a tiny Hitachi microcontroller (H8300) with 32 kilo byte of RAM. Such systems are normally programmed in C or Assembly Language. Programs written in Java are converted by the compiler into something called ‘bytecode’. An interpreter program (a ‘virtual machine’) reads this byte code and executes it. If you implement a Java Virtual Machine for the Lego Robotic Controller, it becomes possible to execute compiled Java Bytecode on the Lego RCX; you can now say goodbye to ‘C’ and program your Lego Robots in Java. That’s exactly what LeJOS helps you do.

Your Lego RCX might come preloaded with the ‘firmware’ provided by the manufacturer - your first step is to replace it with LeJOS. After that is done, you can write programs in Java, convert them to byte code, download to the RCX and see them in action!

3. Downloading and installing LeJOS

This is in fact a very simple process. Grab the source distribution from *lejos.sourceforge.net* and unpack it under, say `/usr/local/src` of your Linux machine (I use a fairly old Debian GNU/Linux box with GCC 2.95.2 and J2SDK 1.3.1). A simple ‘make’ will build the package without any problem. You can change over to the ‘bin’ subdirectory (of the source distribution) and type ./lejosfirmdl to download the firmware (if your IR tower is not connected to COM1, you will have to set an environment variable called RCXTTY to /dev/ttyS1 [for COM2 - if that’s what you use]) by giving the command:

```
export RCXTTY=/dev/ttyS1
```

The download will take some time. Once the firmware download is over, you are ready to compile and run programs.

4. Hello, World

A program called ‘lejosc’ (which is just a wrapper for the actual Java compiler, ‘javac’) should be used for compiling your Java code (you will have to include the path to the ‘bin’ directory of the LeJOS source distribution in your ‘PATH’ variable). Another program called ‘lejos’ lets us download the code to the RCX. We will first try out the classic ‘hello,world’.

```
import josx.platform.rcx.*;

class hello{
 static void main(String args[])
 {
  LCD.clear();
  TextLCD.print("hello");
  try{
```

```
   Thread.sleep(3000);
  }catch(InterruptedException e){}
 }
}
```

The import statement in the beginning is essential - we will not be able to access classes such as ‘LCD’ without it. It will be good if you go to the subdirectory `classes/josx/platform/rcx` of the LeJOS source distribution and read the class definitions. You will see that ‘class LCD’ (defined in LCD.java) has a static member function ‘clear’.

We compile the program:

```
lejosc hello.java
```

And download it:

```
lejos hello
```

Press the ‘run’ button of the RCX and our ‘hello, world’ ritual is over!

The static member function ‘showNumber’ of the class ‘LCD’ helps us to display numbers on the RCX’s LCD screen, instead of text messages. Almost all the information you need to program with LeJOS could be obtained by browsing through the class definition files in `classes/josx/platform/rcx`.

5. Manipulating the motors

Here is a small program which helps us manipulate the motors attached to the RCX (Motor A and Motor C).

```

import josx.platform.rcx.*;

class motor3 {
 static void main(String args[])
 {
  Motor.A.setPower(1); /* Very slow. Max 7 */
  Motor.C.setPower(1); /* Very slow. Max 7 */
  Motor.A.forward();
  Motor.C.forward();
  try{
   Thread.sleep(1000);
  }catch(InterruptedException e){}
  Motor.A.stop();
  Motor.C.stop();
 }
}
```

The ‘Motor’ class contains three static fields (A, B, C) which are references to three objects of type ‘Motor’ having id’s (the Motor class has a character field called ‘iId’) ‘A’, ‘B’ and ‘C’. The Motor class itself can’t be instantiated (private constructor). This is a common theme which you observe in LeJOS. We set the power level to be delivered to 1 (maximum is 7 and default is 3) and start the motors by calling ‘forward’. Once the motors start, they keep running until you stop them by calling ‘stop’ after a delay of 1 second.

6. Using the touch sensors

The basic Lego Mindstorm kit comes with two touch sensors and one light sensor. The ‘Sensor’ class models these sensors. It contains three static fields S1, S2 and S3 which are references to objects of type ‘Sensor’. A sensor has both a ‘type’ and a ‘mode’. For example, the type can be ‘touch’ or ‘light’. If the type is touch and the mode is say ‘edge’, the sensor ‘readValue’ method will return a value which gets incremented on every switch closure and release (both a closure and release generate an electrical ‘edge’). The types and modes are defined by constants in the file `classes/josx/platform/rcx/SensorConstants.java` For example, SensorConstants.SENSOR_TYPE_TOUCH defines a touch sensor and SensorConstants.SENSOR_MODE_EDGE defines the edge counting mode. An example program will make the usage clear.

```
import josx.platform.rcx.*;

/* Both a press and a release are edges - counts edges.
/* SensorConstants.SENSOR_MODE_BOOL lets us check whether
 * the sensor is pressed or released  (pressed = 1,
 * released = 0).
 */

class touch2{
 static void main(String args[])
 {
  Sensor.S1.setTypeAndMode
  (SensorConstants.SENSOR_TYPE_TOUCH,
   SensorConstants.SENSOR_MODE_EDGE);
  while(true) {
   LCD.clear();
   LCD.showNumber(Sensor.S1.readValue());
   try{
    Thread.sleep(100);
   }catch(InterruptedException e){}
  }
 }
}
```

Here is a program which lets a Lego Rover move forward until touch sensor 1 is pressed:

```
import josx.platform.rcx.*;

class touchmot1{
 static void main(String args[])
 {
  Sensor.S1.setTypeAndMode(1, 0x20); /* TOUCH, BOOL */
  Motor.A.forward();
  Motor.C.forward();
  while(Sensor.S1.readValue() == 0);
  Motor.A.stop();
  Motor.C.stop();
 }
```

```
}
```

‘readValue’ returns 1 when the sensor is pressed. What does the program below do?

```
import josx.platform.rcx.*;

class touchmot2{
 static void main(String args[])
 {
  Sensor.S1.setTypeAndMode(1, 0x20); /* TOUCH, BOOL */
  while(true) {
   Motor.A.forward();
   Motor.C.forward();
   while(Sensor.S1.readValue() == 1); /* pressed? */
   while(Sensor.S1.readValue() == 0);
   Motor.A.backward();
   Motor.C.backward();
   while(Sensor.S1.readValue() == 1);
   while(Sensor.S1.readValue() == 0);
  }
 }
}
```

We start both motors in the forward direction. The touch sensor is not pressed, so we come out of the first loop. Now we wait till the touch sensor gets pressed (the motors keep on rotating all the time). Then we reverse the motion and wait for the touch sensor to get released. The motors keep on running till the next switch closure and we repeat the process...

7. Listeners

A Sensor object can have a ‘listener’ attached to it. Whenever the state of the sensor changes, the Sensor object invokes the ‘stateChanged’ method of the listener. Our listener class should implement an interface called ‘SensorListener’ (which declares the single abstract method ‘stateChanged’). The method is invoked with three arguments - the first one being a reference to the sensor object whose state has changed and the others, the new and the old values of the sensor.

Here is a program which changes the direction of the motors every time touch sensor 1 is pressed and stops the motors when touch sensor 3 is pressed.

```
import josx.platform.rcx.*;

class touchListener implements SensorListener {
 void reverse()
 {
  if (Motor.A.isForward()) {
   Motor.A.backward();
   Motor.C.backward();
  } else {
   Motor.A.forward();
   Motor.C.forward();
  }
 }

 void stop()
```

```
 {
  Motor.A.stop();
  Motor.C.stop();
 }

 public void stateChanged(Sensor src, int oldValue, int newValue)
 {
  if(src.getId() == Sensor.S1.getId()) {
   if((oldValue == 0) && (newValue == 1))
    reverse();
  } else stop();
 }
}

class touchmot4{
 static void main(String args[])
 {
  Sensor.S1.setTypeAndMode(1, 0x20); /* TOUCH, BOOL */
  Sensor.S3.setTypeAndMode(1, 0x20); /* TOUCH, BOOL */
  Sensor.S1.addSensorListener(new touchListener());
  Sensor.S3.addSensorListener(new touchListener());
  Motor.A.forward();
  Motor.C.forward();
  while(true);
 }
}
```

We attach listener objects to both sensors. The listener object attached to sensor 1 is notified every time the sensor state changes from 0 to 1 or 1 to 0 - only when the state changes from 0 to 1 (we have a sensor ‘press’) is the reversal action done. The stateChanged method calls ‘reverse’ or ‘stop’ depending on whether it was called by sensor 1 or sensor 3.

8. Using the light sensor

If ‘setTypeAndMode’ is not called on a Sensor object, it is assumed that the sensor is a light sensor. Suppose we have attached a light sensor to the sensor 2 input of the RCX. The following program measures the intensity of the light falling on the sensor and displays it as a numerical value on the LCD panel. Note that we ‘activate’ the light sensor (the LED attached to the sensor should come on - we get greater accuracy now) by calling ‘activate’.

```
import josx.platform.rcx.*;

class light1{
 static void main(String args[])
 {
  Sensor.S2.activate();
  while(true) {
   LCD.clear();
   LCD.showNumber(Sensor.S2.readValue());
   try{
    Thread.sleep(100);
   }catch(InterruptedException e){}
  }
 }
}
```

9. A simple strip counting robot

We build a simple robotic rover and make it run (for a short distance) on a surface of uniform color and texture on which thin paper strips (of a different, uniform color) are pasted at short gaps - say we have 5 such strips. The light sensor of the rover is parallel to the surface. Whenever the vehicle ‘enters’ a strip, a counter should get incremented. After 5 such strip encounters, the vehicle should reverse. Now, whenever it ‘enters’ a strip in the reverse journey, the stored count should get decremented. When the count becomes 1, the journey should stop. The robot will know when it encounters a paper strip by the change in intensity of reflected light. Before conducting the experiment, we measure light intensities on both the paper strip as well as the surface (for my experiment, I used white paper and a dark surface)

```

import  josx.platform.rcx.*;

class lightListener implements SensorListener {
 int stripCount = 0;
 boolean alreadyInside = false;
 final int INFLECTION_POINT = 40, MAX_STRIPS = 4;
 char direction = 'f';

 boolean insideStrip(int val)
 {
  if(val > INFLECTION_POINT) return true;
  return false;
 }

 boolean outsideStrip(int val)
 {
  if(val <= INFLECTION_POINT) return true;
  return false;
 }

 boolean enterStrip(int oldval, int newval)
 {
  if(outsideStrip(newval)) {
   alreadyInside = false;
   return false;
  }
  if(insideStrip(newval)) {
   if(alreadyInside == false) {
    alreadyInside = true;
    return true;
   }
  }
  return false;
 }

 void actionOnEnterStrip()
 {
  Sound.beep();
  if (direction == 'f') {
   if(++stripCount == MAX_STRIPS) {
```

```
    direction = 'r';
    Motor.A.backward();
    Motor.C.backward();
   }
  } else if (direction == 'r') {
   if (--stripCount == 1) {
    Motor.A.stop();
    Motor.C.stop();
    Sound.beepSequence();
   }
  }
  LCD.showNumber(stripCount);
 }

 public void stateChanged(Sensor src, int oldval, int newval)
 {
  if(enterStrip(oldval, newval))
   actionOnEnterStrip();
 }
}

class strip2 {
 public static void main(String args[])
 {
  Sensor.S2.activate();
  Sensor.S2.addSensorListener(new lightListener());
  Motor.A.forward();
  Motor.C.forward();

  while(true);
 }
}
```

10. Multithreading

Normal Java thread syntax works with LeJOS. Here is an example program:

```
import josx.platform.rcx.*;

class abc extends Thread {
 public void run()
 {
  int i = 0;
  while(true){
   LCD.showNumber(i++);
   try{
    Thread.sleep(50);
   }catch(InterruptedException e){}
  }
 }
}

class def extends Thread {
 public void run()
```

```
 {
  Sensor.S3.setTypeAndMode(1, 0x20); /*TOUCH, BOOL*/
  while(Sensor.S3.readValue() == 0);
  Motor.A.forward();
  Motor.C.forward();
  while(Sensor.S3.readValue() == 1);
  while(Sensor.S3.readValue() == 0);
  Motor.A.stop();
  Motor.C.stop();
 }
}

class thread3 {
 static void main(String args[])
 {
  abc a = new abc();
  def d = new def();
  a.start();
  d.start();
 }
}
```

The first thread keeps on displaying a counter which gets incremented every 50ms. The second thread starts the motors when touch sensor 3 is pressed and stops them when sensor 3 is pressed once again.

Threads have priorities - minimum is 1, maximum 10 and default is 5. Priority can be altered by the ‘setPriority’ method. You might like to browse through the thread class implementation in `classes/java/lang`. After you do so, you might like to think about why the program below behaves the way it does:

```
import josx.platform.rcx.*;
class abc extends Thread {
 int id;
 abc(int n)
 {
  id = n;
 }
 static void delay(int n)
 {
  int i;
  for(i = 0; i < n; i++);
 }
 public void run()
 {
  int i, j = id + 10;
  for(i = id; i < j; i++) {
   LCD.showNumber(i);
   delay(7000);
  }
 }
}

class thread4 {
 static void main(String args[])
 {
  int i;
```

```
  abc a1, a2;
  a1 = new abc(0);
  a2 = new abc(20);
  a1.setPriority(7);
  a2.setPriority(3);
  a1.start();
  try {
   TextLCD.print("hello");
   Thread.sleep(1000);
  }catch(InterruptedException e){}
  a2.start();
  for(i = 40; i < 50; i++) {
   LCD.showNumber(i);
   abc.delay(7000);
  }
 }
}
```

11. Button, sound, battery

Here is a program which starts motor A when the VIEW button is pressed and stops it when PRGM is pressed.

```
import josx.platform.rcx.*;

class myButtonListener implements ButtonListener {
 public void buttonPressed(Button b)
 {
  if (b.getId() == Button.VIEW.getId())
   Motor.A.forward();
  else if (b.getId() == Button.PRGM.getId())
   Motor.A.stop();
 }
 public void buttonReleased(Button b)
 {
 }
}

class button2 {
 static void main(String args[])
 {
  Button.VIEW.addButtonListener(new myButtonListener());
  Button.PRGM.addButtonListener(new myButtonListener());
  while(true);
 }
}
```

A program which generates a nice beep sequence when the VIEW button is pressed and released:

```
import josx.platform.rcx.*;

class button1 {
```

```
 static void main(String args[])
 {
  try{
   Button.VIEW.waitForPressAndRelease();
  }catch (InterruptedException e){}
  Sound.beepSequence();
  try{
   Thread.sleep(1000);
  }catch(InterruptedException e){}
 }
}
```

A program which displays the current battery level in millivolts:

```
import josx.platform.rcx.*;

class battery {
 static void main(String args[])
 {
  LCD.showNumber(Battery.getVoltageMilliVolt());
  try{
   Button.VIEW.waitForPressAndRelease();
  }catch(InterruptedException e){}
 }
}
```

12. Persistent Memory

It is possible for a program to store some data in a 'persistent' memory area and use it during its next execution. There are several restrictions - please read the comments in `classes/josx/platform/rcx/PersistentMemoryArea.java`. Here is an example program:

```

import josx.platform.rcx.*;

class persistent {
 static void main(String args[])
 {
  int magic = 0x7632, size = 10, i;
  byte b;
  PersistentMemoryArea p;
  try {
   p = PersistentMemoryArea.get(magic, size);
  }catch(OutOfMemoryError e){
   Sound.beepSequence();
   return;
  }
  b = p.readByte(0);
  for(i = 0; i < 5; i++) {
   LCD.showNumber(b);
   if(b == 127) b = 0;
   else b++;
```

```
   try{
    Thread.sleep(500);
   }catch (InterruptedException e){}
  }
  p.writeByte(0, b);

 }
}
```

The persistent area is identified by a ‘magic number’.