Using LeJOS

This text is intended as a short guide for students using LeJOS in classroom projects. It does not replace the documentation coming with LeJOS and it does not teach Java programming. If you are seriously interested in Java on the RCX look for books on the topic. §

I have tested the examples with LeJOS 1.0.5, Sun Java2 SDK 1.4.0, IBM Jikes 1.15, and Linux 2.4. Things may or may not work on Windows or with other versions.

Here is a single file with all the pages. There is also a pdf version.

Installation

Getting LeJOS

LeJOS is available from the Sourceforge project homepage. Download two files:

`lejos_1_0_5.tar.gz`	The compiler sources, JVM, and API classes
`lejos_1_0_5.doc.tar.gz`	Javadoc generated API docs

You can create the contents of the second package from the first.

Building

Unpack `lejos_1_0_5.tar.gz` to an installation directory:

```
$ tar xvzf lejos_1_0_5.tar.gz
```

This creates the subdirectory `lejos_1_0_5`.

LeJOS comes partially built. Change to the new directory and run `make` to finish building LeJOS:

```
$ cd lejos_1_0_5
$ make
```

`make` takes some time and creates various subdirectories. The most important ones are `./bin` and `./lib`. Here are the essential files:

`bin/lejosfirmdl`	Download utility to transfer the JVM to the RCX	Linux binary
`bin/lejos.srec`	JVM image file for the RCX	RCX firmware image
`bin/lejosc`	Java compiler driver	Linux binary
`bin/lejos`	Static linker and download utility for Java programs	Linux binary
`bin/lejosrun`	Internal programm called by `lejos`	Linux binary
`lib/classes.jar`	LeJOS API classes	Java bytecode
`lib/jtools.jar`	Internal classes used by `lejos`	Java bytecode

To "install" LeJOS simply add `./bin` to the program search path. For a bash:

```
$ PATH=`pwd`/bin:$PATH
```

Edit one of the startup files to make this permanent. § The binaries (`lejos`, `lejosc`, ...) look for other LeJOS files in directories relativ to their own location (for example *where_lejos_is*`/../lib/`*some_file*).

If you prefer a minimum installation, copy the files listed above from `./bin` to a directory in the search path (an element of `$PATH`, usually `/usr/bin` or `/usr/local/bin`). Copy the Jar files to the corresponding

../lib directory (/usr/lib or /usr/local/lib).

Making LeJOS use Jikes

By default LeJOS uses the standard Java compiler javac from the Sun SDK. javac is a Java program itself, and therefore not exactly a speed demon. IBM offers another Java compiler, jikes. § Jikes is written in C++ and several times faster than javac. Here is how to make LeJOS use Jikes instead of Javac.

1. Edit the main Makefile and change:

   ```
   #JAVAC=jikes -bootclasspath c:\\some_windows_path\\rt.jar
   JAVAC=javac -target 1.1
   ```

   to

   ```
   JAVAC=jikes -bootclasspath /usr/jre/lib/rt.jar
   #JAVAC=javac -target 1.1
   ```

   The path to rt.jar will probably be wrong. Locate rt.jar on your system and enter the absolute pathname here. §

2. Edit tools/Makefile and change two lines:

   ```
   TVMCDEFS=-DTOOL_NAME=\"javac\" -DTOOL_ALT_VAR=\"JAVAC\" -DJAVA2
   TVMC1DEFS=-DTOOL_NAME=\"javac\" -DTOOL_ALT_VAR=\"JAVAC\"
   ```

   to

   ```
   TVMCDEFS=-DTOOL_NAME=\"jikes\" -DTOOL_ALT_VAR=\"JAVAC\"
   TVMC1DEFS=-DTOOL_NAME=\"jikes\" -DTOOL_ALT_VAR=\"JAVAC\"
   ```

3. Recompile LeJOS:

   ```
   $ make clean
   $ make
   ```

4. If you chose the minimum installation, copy bin/lejosc to the destination directory.

On my system, the compile time for Hello.java drops from 2.2 seconds (Javac) to 0.2 seconds (Jikes).

API Docs

Unpack the API documentation:

```
$ tar xvzf lejos_1_0_5.doc.tar.gz
```

This creates the subdirectory lejos_1_0_5.doc. Point your browser to lejos_1_0_5.doc/apidocs/index.html to view the documentation.

You can build the API docs from the Java source files on your own. In the installation directory run:

```
$ make javadoc
```

This generates a more complete and up–to–date version in ./apidocs.

Basic Steps

Firmware Download

The RCX needs some ***firmware*** to run user programs. Firmware is for the RCX like an operating system for a PC. The LEGO RIS and NQC both use the same firmware ("LEGO firmware"), which is supplied on the

CD–ROM in the Mindstorms Box. [§] LeJOS uses a different firmware, which essentially implements a JVM.

A new RCX has no firmware. Firmware survives for 20–30 seconds, when you remove the batteries from an RCX. Any firmware is ***immediately gone***, if you switch the RCX on with batteries removed. An RCX with no firmware shows the little man standing and a `1` in its display, nothing else:

All firmware files have a specific file format ("motorola S–record"), no matter what they contain. There are many different ***firmware download utilities,*** and LeJOS comes with its own. You may use any firmware download utility with any firmware.

LeJOS needs to know the device file of the IR tower. Set the ***environment variable*** `RCXTTY` to the device file: `/dev/ttyS0` for the first serial port, `/dev/ttyS1` for the second port, and so on. If your system uses the newer device filesystem, the names are `/dev/tts/0`, `/dev/tts/1`, ...

To ***download*** the LeJOS ***firmware*** run

```
$ lejosfirmdl
```

`lejosfirmdl` looks for the file `../bin/lejos.srec` and transfers it to the RCX. Any previously loaded firmware is silently overwritten. While downloading, a counter runs up in the display. It counts in units of 10 bytes. Since the firmware is about 16 kB, the download is complete at around 1600.

There is a ***fast download mode*** that transfers (in theory) 4x the normal rate.

```
$ lejosfirmdl -f
```

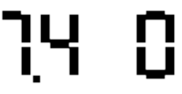
Timings on my machine give approximately:

normal download	170 sec	(100 byte/sec)
fast mode	50 sec	(325 byte/sec)

Fast mode is not documented by LEGO and does not work in all configurations. Newer RCXs have problems, and the USB IR tower has no fast mode at all.

Once the firmware is successfully downloaded, it takes control of the RCX. The LEGO firmware shows the uptime (in minutes) and the active program slot:

The LeJOS JVM shows the voltage level and a zero:



If there is a user program in the RCX, you see the little man:

Compiling

Here is the first program `Hello`. A discussion of the various statements follows below. Compile it with `lejosc`:

```
$ lejosc Hello.java
```

`lejosc` is just a driver that calls a real Java compiler for you, supplying appropriate arguments. You can call the Java compiler directly: §

```
$ javac -classpath lib/classes.jar -target 1.1 Hello.java
```

Or for Jikes:

```
$ jikes -classpath lib/classes.jar Hello.java
```

If you don't want to give the "–classpath" argument with every call, set the `CLASSPATH` environment variable:

```
$ export CLASSPATH=.:lib/classes.jar
```

Don't forget the dot, or `lejos` won't find your program (`Hello.class`) in the working directory for downloading to the RCX.

Downloading

The compiler generates a standard bytecode file, for example `Hello.class`.

A normal Java program loads additional bytecode as needed. Obviously this cannot work for the RCX: Once a program is downloaded and operating, the RCX has not necessarily a connection back to the host, and cannot request more bytecode at runtime.

Therefore all the bytecode that a LeJOS program might eventually ever use has to be located and downloaded at once. For other programming languages this is called ***"static linking"***. § The utility program `lejos` does static linking for Java: It collects all bytecode that a program might use and packs it into a temporary image. This image is transmitted to the RCX immediately. The image itself is pretty useless, except for downloading to the RCX. §

Run `lejos` to perform both steps (static linking and downloading) in one call:

```
$ lejos Hello
```

The argument "–verbose" makes `lejos` print the content of the static image:

```
$ lejos -verbose Hello
```

In the list you will find `Hello`, along with a lot of other classes used explicitly (`TextLCD`, `Thread`) or indirectly. This list is necessary to locate the source of an exception, if the program bombs out.

While downloading the LCD display shows a ***counter*** running up. It counts units of 100 bytes. The static image of `Hello` is around 3 kB, so the counter goes up to 30–something.

Fast mode doesn't work here. The ***download rate*** is about 75 bytes/sec. Downloading `Hello` takes 40+ seconds. When the download completes successfully, the RCX double beeps, and returns to voltage display.

Start the program by pressing the green "Run" button. The display shows

until you press and release the green "Run" button, then returns to voltage display. The little man is shown walking, as long as a program executes.

The RCX keeps a LeJOS program through power cycles.

There is no way of interrupting a running program, if it doesn't explicitly care. Of course, you can always switch the RCX off (red "On–Off" button).

Most of the time you compile, then download, then compile again, download again, and so on.. This is part of the development cycle: edit, compile & download, run & test. I find it convenient to pack both calls in a short shell script, for example "`lj`":

```
#!/bin/bash
if [ -f $1 ]; then
        lejosc $1 &lejos ${1%.java}
else
        lejosc $1.java &lejos $1
fi
```

Place the script somewhere in the PATH, make it executable ("chmod +x lj"), then call it with either a Java source file or a class name.

Anatomy of `Hello.java`

This is a short discussion of the parts of the first Java program. Here is the source code:

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

class Hello
{
    public static void main(String[] args) throws InterruptedException
    {
        TextLCD.print("HELLO");
        Button.RUN.waitForPressAndRelease();
    }
}
```

`import josx.platform.rcx.*;`
: LeJOS does not use the standard API library. It supplies it's own version of `java.lang`, which is imported by default. LeJOS' package `java.lang` has most of the standard classes, but not all.
The package `josx.platform.rcx` has classes that represent the various hardware components of the RCX. A program that works in memory only (computes a sum of numbers, for example) could do without this import clause. §

`class Hello {...}`
: LeJOS programs consist of classes, as all Java programs do. You are free to choose class names.

`public static void main(String[] args)...`
: The `main` method is a program's entry point. It is called first.
The `String[]` parameter is a dummy. It does not receive a value.

...`throws InterruptedException`
: Exceptions must be caught or stated in the exception signature. Since `Button.RUN.waitForPressAndRelease();` could throw an `InterruptedException`,

we must deal with it. Here we simply throw it. See below on exceptions.

`TextLCD.print("HELLO");`

`TextLCD` is a class in package `josx.platform.rcx`. There is only one instance of class `TextLCD` (it is a singleton). This object represents the LCD display. It would be useless to create more `TextLCD` objects.
Class `TextLCD` defines static methods to control the LCD display. `print(String s)` is one of them. It displays the first 5 characters of `s` on the LCD display. Not all characters appear readable, some are just approximations.

`Button.RUN.waitForPressAndRelease();`

The program blocks here, until you press and release the green "Run" button. This gives us time to read the display. Without it, the program would terminate immediately and return control of the RCX to the firmware. The firmware immediately overwrites the LCD display with the current battery level.

Getting information in and out

The RCX has several sources of information:

- Sensor input ports (A, B, C)
- Rubber buttons
- Internal Timers
- Receiving messages from the IR port
- Battery level

It can produce information in various ways:

- Motor output ports (1, 2, 3)
- Internal beeper
- LCD display
- Sending messages through the IR port

LeJOS can make use of all of them. They will be discussed in turn.

Motors

Class `Motor` contains three static members, each representing one RCX output (black connectors).

`Motor.A`

`Motor.B`

`Motor.C`

The member names match the labels (A, B, C) printed on the RCX. It is meaningless to create new Motor objects.

`Motor` defines some methods to ***control outputs.*** The most important ones are:

`forward()`	switch the motor on and run it in forward direction.

`backward()`	switch the motor on and run it in reverse direction.
`stop()`	switch the motor off and block it actively.
`flt()`	let the motor float free.

Here is program `GoStopBack` to turn motor A forward for 1 second, then stop for 1 second, then turn it backward for 1 second, then release it.

Forward and backward

The program or the RCX have ***no idea*** of ***what direction*** "forward" (or "backward") is. The method names "forward" and "backward" mean just opposite voltage levels. What direction a motor turns, or in what direction a robot moves, depends entirely on the construction. It is usually a good idea to connect wires in such a way, that calling `forward()` in the program makes the robot move foward as well. If it doesn't, just turn the wire connectors on the RCX by 180°.

Power levels

There are methods to deal with 8 different ***power levels*** (0 = no power = off ... 7 = full power). In reality using the power level makes little difference, if the motor isn't heavy loaded. If you want run a motor slower than normal, try switching it rapidly on and off under program control. Program `GoSlow` shows this.

Motor speed can be adjusted by varying the two sleep times. The motor makes a funny noise, but I haven't had any damage yet.

Power consumption

The RCX lives from batteries, [§] therfore power consumption is an issue. Dave Baum has written a page about LEGO motors and their properties. Here is a short summary:

Motor type	*Item–#*	*running free*	*stalled*	*rpm unloaded*
Standard motor	5114			3000+
Geared Mindstorms Motor	5225			300
Red micro motor				25

(More on rotation speed see below.)

The geared LEGO Mindstorms motors work very efficiently. You can see this, if you connect two motors with a wire, and turn the axle of one motor by hand. The other motor's axle follows immediately.

Lamps

Outputs can control ***other effectors,*** not just motors. For example, the *LEGO Ultimate Accessory Set* (Item #3801) comes with a lamp. This is not a LED. It lights no matter what "direction" the output is "running".

You can easily ***build your own LED lights*** from standard LEDs. Cut a LEGO wire and solder a 100–150 Ohm resistor and a LED in series to the open end:

Not beautiful, but cheap. The LED will light only in one output "direction".

Very bright ***LED light bricks*** are available from Hitechnic Stuff in different colors (red, green, yellow, blue, bi–color).

Sensors

Class `Sensor` defines three static members, one for each RCX input port (gray connectors):

`Sensor.S1`

`Sensor.S2`

`Sensor.S3`

The member names match the labels (1, 2, 3) printed on the RCX. It is meaningless to create new Sensor objects.

`Sensor.SENSORS` is an array with 3 elements, holding the same objects as above. [§] This way you can access all sensors by index, for example in a short loop.

Sensor Configuration

Three steps are required to use a sensor properly ("sensor configuration"):

1. Set sensor type.
2. Set sensor mode.
3. Activate the sensor (for some sensors).

Method `Sensor.setTypeAndMode` performs steps 1 and 2. Method `Sensor.activate` performs step 3.

Sensor type

Sensor of different types (touch, light, rotation, ...) require specific electrial signals for proper operation. The RCX cannot find out what sensor type is attached to a port, therefore you have to tell the ***sensor type*** explicitly. Interface `SensorConstants` defines constants for the different sensor types:

Sensor type
`SensorConstants.SENSOR_TYPE_TOUCH`
`SensorConstants.SENSOR_TYPE_LIGHT`
`SensorConstants.SENSOR_TYPE_ROT`
`SensorConstants.SENSOR_TYPE_TEMP`
`SensorConstants.SENSOR_TYPE_RAW`

Sensor mode

Sensors return a voltage level or appear as an electrical resistor. The RCX sees the response as an unsigned 10–bit value in the range 0–1023 (= $2^{10} - 1$). This is the sensor's ***raw value***.

The ***sensor mode*** tells the RCX how to interpret the raw value. Interface `SensorConstants` defines constants for different sensor modes:

Sensor mode	*Useful for sensor type*	*Remark*	*Listener reading*
`SENSOR_MODE_BOOL`	Touch	High raw values are mapped to `false`, low values to `true`.	0, 1
`SENSOR_MODE_RAW`	All	Digital voltage level, no interpretation,	0...1023
`SENSOR_MODE_PCT`	All	Scales raw values to 0–100 with 0 = high raw value, 100 = low raw value.	0...100
`SENSOR_MODE_ANGLE`	Rotation	Accumulate signals from a rotation sensor and return the sum of all signals received up to now. Signals have a sign and may add to or subtract from the sum.	
`SENSOR_MODE_DEGC`	Temperature	Map to degrees centigrade.	
`SENSOR_MODE_DEGF`	Temperature	Map to degrees fahrenheit.	
`SENSOR_MODE_EDGE`	Touch	Interpret as `SENSOR_MODE_BOOL`; Returns the total number of value changes received up to now. This is the number of times a touch sensor was pressed or released.	0...
`SENSOR_MODE_PULSE`	Touch	Interpret as `SENSOR_MODE_BOOL`; Return the total number of changes from 1 to 0 received up to now. This is the number of times a touch sensor was released.	0...

You can mix any type and mode, but most combinations are useless.

Active and passive sensors

Light and rotation sensors ("active sensors") require power for operation, touch and temperature sensors ("passive sensors") do not. Method `Sensor.activate` supplies power to an active sensor, `Sensor.passivate` cuts it. Passive sensors don't care.

Reading a sensor

Once a sensor is configured, the method `Sensor.readValue()` returns its current value, interpreted as requested by the mode. Attach a touch sensor to input 1 and run program `ShowTouch.` The LCD display shows `1` or `0` as the touch sensor is pressed or released.

Touch Sensor

Touch sensors are basically switches. They are open (sensor released) or closed (sensor pressed).

Boolean mode

This is the most useful mode for touch sensors. Values 0 and 1 correspond to released and pressed state. See program `LCDShowTouch` for an example.

Raw mode

Change `SENSOR_MODE_BOOL` in program `ShowTouch` to `SENSOR_MODE_RAW`:

```
Sensor.S1.setTypeAndMode(SensorConstants.SENSOR_TYPE_TOUCH,
                         SensorConstants.SENSOR_MODE_RAW);
```

Values are above 1000, when the sensor is released, and below 200, when it is pressed.

The sensor reading jumps around an average value. This is one reason why "cooked" modes (for example boolean or percent mode) are easier to handle than raw mode.

The lower value depends even on the specific sensor exemplar. Attach a different touch sensor to the input, and you will probably see other low values.

?? two touch sensors on one input

Percent mode

Change program `ShowTouch` again to `SENSOR_MODE_PCT`:

```
Sensor.S1.setTypeAndMode(SensorConstants.SENSOR_TYPE_TOUCH,
                         SensorConstants.SENSOR_MODE_PCT);
```

You will see values 0 (released) or 100 (pressed).

In contrast to raw mode, percent mode gives a stable reading and does not depend on the specific sensor exemplar.

Pulse mode

Pulse mode makes sense for touch sensors. It is based on boolean mode. Instead of giving the current value, it counts transitions from 1 to 0. This corresponds to touch sensor releases.

Pulse mode (as well as edge mode and angle mode for rotation sensors) return an accumulated value, not the current state of a sensor. Method `setPreviousValue` sets a sensor to a desired value, usually 0. Without this method the sensor value would accumulate for ever.

Example program `ShowPulses` counts touch sensor pulses up to 10, then resets to 0, and so on.

?? delay

Edge mode

Almost the same as pulse mode, but counts *all* transitions from 0 to 1 and back.

Light Sensor

Light sensors have a red LED and a light–sensitive diode, that responds to infra–red light. It needs `activate` to work.

Percent mode is the most useful mode for light sensors. Raw mode works, but jumps around an average value. Program `LCDShowLight` displays the light sensor reading. Values are about 80 for daylight, 20 in a dark room.

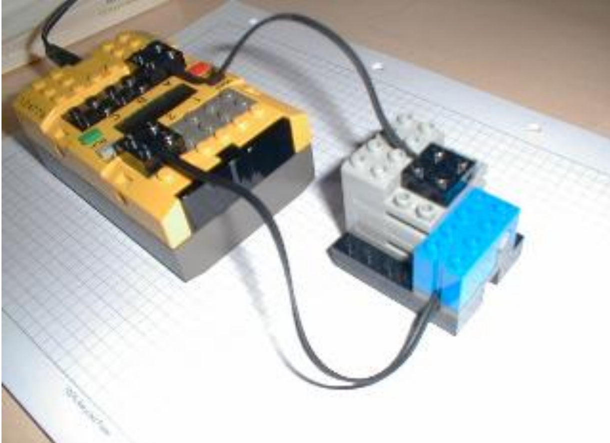
?? sensitive to signals from IR tower, LEGO remote control, RCX IR sender

Rotation Sensor

Rotation sensors count units of 1/16 of a full circle (steps of 22.5°). They require `activate` to work.

The only useful mode is angle mode, as in program `ShowRotation`.

A rotation sensor counts fairly reliable at slow rotation speeds (1 rpm). It also works at the rotation speed of an unloaded motor (#5225 gear motor, about 300 rpm). Connect the axle of a motor directly to a rotation sensor:



Program `RotationSpeed` displays the rotation speed in rpm. Here are some results:

Power supply	*voltage level*	*motor*	*rpm*
6 rechargeables	7.2	Mindstorms motor #5225	275
		Micro motor	20
		Standard motor #5114	cannot follow
1.0–RCX external	10.0	Mindstorms motor #5225	365
		Micro motor	25
		Standard motor #5114	cannot follow

Listeners

Polling vs. Callback

The examples above use calls to `Sensor.readValue()` to explicitly request sensor values ("polling"). An alternative programming scheme uses listeners to react to sensors (or asynchronous events in general).

A listener is an object with a specific method (`stateChanged`). This method ("callback method") is never called directly from within the user program. Instead, it is called by the VM, whenever a sensor changes its state (hence the name `stateChanged`). In a short sentence, the idea is

"Don't call us, we call you."

This is comparable to a mail service. When you order a package by mail, you will not run to the post office every hour and check if your package has arrived. (Even if, depending on the package, sometimes might want to do.) Instead, you rely on the post man to come to you and deliver the package, as soon as it arrives.

The idea of callbacks is somewhat suspicious, if you are not used to it. Many people feel they loose control, and don't like something else to call their methods, especially if "something else" is not visible.

Listeners are generally ***preferable*** to polling, because they don't waste CPU cycles with method calls, that return no new information. They are appropriate, whenever you need to react to asynchronous events. Asynchronous events happen at points of time, that your program cannot control or reliably predict.

For example, look at program `LCDShowTouch`, that continuously shows a boolean touch sensor value in the LCD display. The loop runs fast and reads the sensor very often. Most of the time the sensor value will not have changed. The loop will eat lots of CPU cycles, putting the same value in the LCD display over and over again.

We might add a delay to the loop to reduce CPU load, as in program `ShowTouchDelay`:

```
while(true)
{
     ...
     Thread.sleep(100);
}
```

But we can only guess about a appropriate delay time. Here we use 100 millis. This may be too fast, or too slow, or accidentally ok. In this simple program it does not matter anyway. In a more complex program it will certainly matter.

A listener updates the display whenever the sensor state changes, and not at any other time. There is no need to guess a delay. In fact, there is not even a loop (at least not in the user code, and not for the purpose of watching a sensor).

The following pages show you how to use listeners.

Using Listeners

A listener is an object of a ***listener class.*** Listener classes are defined in the user program, and must implement the interface `SensorListener`:

```
class TouchViewer implements SensorListener
{...}
```

The interface makes sure, that all listener classes define the proper ***callback method*** `stateChanged`:

```
class TouchViewer implements SensorListener
{
    public void stateChanged(Sensor s, int old, int nu)
    {...}
```

```
}
```

When the callback method is called later, it receives ***three arguments:***

`Sensor s`
: The Sensor object that changed state. This is one of `Sensor.S1`, `Sensor.S2`, and `Sensor.S3`.

`int old`
: The previous sensor value.

`int nu`
: The current sensor value, different from `old`.

The callback method does whatever you want. For example, show the new sensor value in the LCD display:

```
class TouchViewer implements SensorListener
{
    public void stateChanged(Sensor s, int old, int nu)
    {
        LCD.showNumber(nu);
    }
}
```

The VM does not know about your listener, if you don't tell it. Therefore a listener must be ***registered*** to a sensor with `addSensorListener`, as in program `ShowTouchCallback`.

```
...
Sensor.S1.addSensorListener(new TouchViewer());
...
```

There are two different classes:

- The ***main class*** defines method `main`. It creates and registers a listener, then it is done. `main` does *not* contain a loop or a delay.
- The other class is the ***listener class***. It is just a container for the callback method. The object itself is less important.

One object of the listener class is registered, then not used any more by `main`. Therefore, the listener may be defined in an anonymous class:

```
...
Sensor.S1.addSensorListener(new SensorListener()
{
    public void stateChanged(Sensor s, int old, int nu)
    {
        LCD.showNumber(nu);
    }
});
...
```

The class has no name (hence "*anonymous*" class). It is used just once to create one single object.

Multiple Listeners

The same listener object can be registered to different sensors. The first parameter (`Sensor s`) tells the callback method which sensor's state changed.
?? useful example

LeJOS allows ***up to 4 listeners*** registered to one sensor. Here is program `ShowAndBeep`, that beeps when the light sensor value drops faster than a certain rate, and displays the current light sensor value in the LCD display. Two independent listeners are registered, one for each task:

DropWatcher
: watches for the value to drop fast.

ValueDisplay
: shows the value in the display.

The main program registers both.

```
Sensor.S1.addSensorListener(new LightViewer());
Sensor.S1.addSensorListener(new DropWatcher());
```

Sensor threads

Sensor inputs are monitored by a thread in the JVM. When this thread observes a sensor state change, it invokes the registered callback methods. However, it is just a single thread. As a consequence, while one callback method executes, other callback methods are blocked.

Try program `SensorThread`. Connect two touch sensors to inputs 1 and 3, two motors to outputs A and C. Touch sensor 1 runs motor A for 5 seconds, touch sensor 3 motor C. The callback method waits for 5 seconds, then returns. This is intentionally a rather long time.

For the remaining examples we define a simple helper method `nap()` for sleeping. `nap()` returns `true`, if the given time has passed without an interrupt. It returns `false`, if it was interrupted.

```
private final static boolean nap(long millis)
{
    try
    {
        Thread.sleep(millis);
    }
    catch(InterruptedException ex)
    {
        return false;
    }
    return true;
}
```

Here is program `SensorThreadNap`, using `nap()`:

When touch sensor 1 is pressed, motor A starts running. Now press touch sensor 3, while A is still running. Motor C does not start, until A stops, that is, until the first callback is finished. Even worse, the second listener call is buffered and starts executing, when the event has long passed. This probably is not what you wanted.

As a rule, make ***callbacks return as soon as possible.*** They should just signal that a certain condition has arrived, but not perform activities on their own. It is up to somebody else to react to the new condition. This calls for threads; see the next page.

Listeners and threads

In the example above we watch two touch sensors. Whenever one is touched, an action of some duration follows (in the example a motor runs for a couple of seconds). The problem is, there is only *one* thread monitoring all sensors and running *all* callbacks. Therefore, two callbacks cannot execute in parallel.

Action threads

Threads solve this problem: In a callback, a thread is started that performs an action. Now the callback method returns immediately, it does not have to wait for the action to complete. See program `ActionThreads` for

an example.

Lost memory

The solution above still has a problem. Every invocation of a callback method creates and starts a new thread. As the event is repeated, many threads are created and started. Since there is no garbage collection, each new thread eats memory that is permanently lost.

Instead of creating new action threads, we use a single action thread. The action thread sits waiting in an endless loop. When something happens, it runs an action, then resumes waiting again.

Interrupts are a simple Java signalling facility. [§] Two `Thread` methods are useful to use interrupts:

`interrupt()`
: Interrupts a thread.

`sleep()`
: Blocks the current thread for a given time. Throws an `InterruptedException`, when an interrupt is raised within this time.

Here is program `ActionThread` using interrupts. Basically we create another level of listeners.

Overlapping actions

Yet a problem remains. When you press a touch sensor, the motor starts running for 5 seconds, then stops. If you press the same touch sensor again *while the motor is running,* the motor stops immediately. This is because the action thread is interrupted in the second sleep, while performing the action and *not* while waiting for an interrupt.

It depends on the specific action, how to deal with interrupts within that action. Alternatives are:

Abort the current action** and **wait for the next interrupt.
: This is the implementation above.

***Abort the current action** and immediately **start it again** from the beginning.*
: In program `RepeatAction` we use a flag (`completed`) to remember if the action has completed normally (`completed == true`) or was interrupted (`completed == false`) and needs to be restarted.

```
public void run()
{
    while(true)
    {
        // wait for an interrupt
        nap(Integer.MAX_VALUE);

        // perform some action
        // repeat until completed is true
        boolean completed = false;
        while(!completed)
        {
            myMotor.forward();
            completed = nap(5000);
            myMotor.stop();
        }
    }
}
```

***Ignore the interrupt** and continue with the action.*

This is more complicated, because in Java interrupts cannot be blocked. In program `UninterruptibleAction` we calculate and store the time (`doneAt`), when the action is completed. When interrupted within the action, we resume speeling for the rest of the time until completion.

```
public void run()
{
    while(true)
    {
        // wait for an interrupt
        nap(Integer.MAX_VALUE);

        // perform some action
        // continue sleeping until the time is over
        int doneAt = (int)System.currentTimeMillis() + 5000;
        myMotor.forward();
        while((int)System.currentTimeMillis() < doneAt)
            nap(doneAt - (int)System.currentTimeMillis());
        myMotor.stop();
    }
}
```

Scheme for using listeners

The last example shows a general construction scheme:

1. Define one ***callback*** for each source of asynchronous ***events.*** The callbacks interrupt other threads, but do not take action on their own.
2. Define one ***thread*** for each ***activity.*** An action thread waits for some condition to occur, then starts a response to the event.
3. The ***main program*** only starts the action threads, registers callbacks, nothing else.

Other listeners

Sensors are just one source of asynchronous events. Other hardware components of the RCX may generate events as well. LeJOS defines listener interfaces for these, too:

`SensorListener`
: Discussed above.

`ButtonListener`
: Called when the rubber buttons are used.

`SerialListener`
: Called when IR messages arrive.

The remaining two information sources, battery level and timers, are treated differently. There are no listener interfaces for these.

Battery level

Buttons

There are four rubber buttons on top if the RCX. They are under control of LeJOS, except of the red "On–Off" button. When no program is running, the green "Run" button starts it.

Class `Button` defines constants corresponding to the buttons:

`Button.VIEW`	black	"View"
`Button.PRGM`	gray	"Prgm"
`Button.RUN`	green	"Run"

The array `Button.BUTTONS` contains the three buttons as elements in the same order.

A call to `waitForPressAndRelease()` blocks, until the button is pressed and released. This is useful if a program wants to wait for the user. Many of the example programs in this text use it.

Another way to handle buttons is to register a `ButtonListener` with a call to `addButtonListener`. A listener object defines two callbacks, `buttonPressed` and `buttonReleased`. Here is example program `ButtonClick`.

Time

Sound

The RCX has a beeper to generate sound. A program can use ***predefined sounds*** ("system sounds"), or it can generate single ***tones.*** Class `Sound` defines methods for this.

Playing tones

Method `playTone` plays a tone, given its frequency and duration. Frequency is audible from about 31 Hz to above 10000 kHz. § The duration argument is in $^1/_{100}$ of a second (centiseconds, not milliseconds) and is truncated at 256, so the maximum duration of a tone is 2.56 seconds.

Here is program `AudibleSounds` to test frequencies.

Sound buffer

After each `playTone` there is a delay of the same duration. If you remove the delay (`Thread.sleep`), the `playTone` calls are executed rapidly in the loop. Actually *playing* the tones takes much longer. The RCX ***buffers up to 8 tones.*** Further `playTone` calls are discarded. Run program `SoundBuffer`. No delay here!

You will hear 9 tones. The first tone starts playing immediately, as soon as `playTone` is called. No need to buffer the tone. The next 8 calls are buffered and played later, because the beeper is still busy playing the first tone. The 10th call and all others are ignored, because the buffer is full.

System sounds

Method `systemSound` selects one of 6 predefined system sounds (table from the API documentation):

code	*sound*	*wrapper*
0	short beep	`beep()`
1	double beep	`twoBeeps()`
2	descending arpeggio	`beepSequence()`
3	ascending arpeggio	
4	long, low beep	`buzz()`

5	quick ascending arpeggio

There are convenience calls for some of the system sounds, see right column. They are just buffering wrappers for corresponding `systemSound` calls.

`systemSound` accepts two arguments: The first is a boolean:

`true`
: If the beeper is busy, place the sound in the buffer and play it as soon as the beeper is available. The buffer has a capacity of 8, see above. All other calls are ignored.

`false`
: Play the sound immediately without buffering. If the beeper is busy, interrupt the currently playing sound.

The second argument is the system sound code, see the table above. Try program `SystemSound`. It plays 9 descending arpeggios (1 immediately plus 8 from the buffer). The rest of the calls is lost. If you change `true` to `false`, the program starts with a short, funny noise: 20 descending arpeggios, rapidly starting to play and getting interrupted. Only the last arpeggio plays normally.

LCD Display

There are two classes for accessing the LCD display, `LCD` and `TextLCD`.

`LCD` has methods for displaying numbers and controlling the various display segments individually.

`TextLCD` lets you display short strings of up to 5 characters at a time. Of course, character display is only an approximation. Some characters are hard to read.

Displaying Numbers

The RCX display has 5 digits, 4 digits to the left of the little man, 1 digit to the right (the "program slot" for the LEGO firmware). Display an unsigned value (0...9999) on the left with `showNumber(int n)`, an unsigned value (0...9) on the right with `showProgramNumber(int n)`. Here is example program `DisplayNumbers`.

The additional method `setNumber()` gives finer control over the values displayed. It accepts three arguments:

1. Type of value
   `LCDConstants.LCD_UNSIGNED`
   : Display values 0...9999 on the left, without a sign. Same as `showNumber(int n)`.
   `LCDConstants.LCD_SIGNED`
   : Display values –9999...9999 on the left, including a negative sign.
   `LCDConstants.LCD_PROGRAM`
   : Display values 0...9 on the right, without a sign. Same as `showProgramNumber(int n)`.
2. Value to be shown.
3. Location of decimal point:
   `LCDConstants.LCD_DECIMAL_0`
   : No decimal point.
   `LCDConstants.LCD_DECIMAL_1`
   : As in `999.9`
   `LCDConstants.LCD_DECIMAL_2`
   : As in `99.99`
   `LCDConstants.LCD_DECIMAL_3`
   : As in `9.999`

`setNumber()` requires a call to `LCD.refresh()` to actually display the number. This is different from `showNumber()` and `showProgramNumber()`, where the `refresh()` is already included. Here is example program `DisplaySignedNumbers`. Another example programm `DisplayDecimalPoint` for the decimal point.

Controlling Segments

Interface `Segments` defines constants corresponding to the various display segments. There are three methods to control the display:

`setSegment(int code)`
    Set a segment.
`clearSegment(int code)`
    Clear a single segment.
`clear()`
    Clear all segments.

All of them require `refresh()` to actually take effect.

This is what you see when all segments are set:

Here is program `Segments` to demonstrate segment display. [§] The "datalog" segment has 4 pieces (quarter circles), the "uploading" segment 5 pieces (little squares). There seems to be no way to control the pieces individually.

Displaying Text

Class `TextLCD` defines methods to display text approximations. At most 5 charactes fit into the display. Character codes range from 32 (ASCII blank) to 127.

`print(char[] text)`
    Displays up to the first 5 characters of the array. No need to refresh.
`print(String text)`
    Displays up to the first 5 characters of the string. No need to refresh.
    Please note: As of LeJOS 1.0.5, this methods ***eats memory.*** The string argument is internally converted into a character array. A *new array* is allocated on every call.

`printChar(char ch, int pos)`
    Prints a single character at the given display position. `pos` is an integer in 0...4, where 0 corresponds to the "program slot" (rightmost) and 4 to the leftmost position:

```
4321 0
```

Requires `refresh()` to take effect. Here is a sample program `Chars` to show all character approximations.

Program `Text` uses character arrays and strings to display text. In the first loop both output sequences consume memory. In subsequent loops `print(String)` eats memory, `print(char[])` does not.

IR Communication

Support for IR communication in LeJOS evolves rapidly. This text only uses the (relatively old) class `josx.platform.rcx.Serial`.

Different LEGO Mindstorms devices use IR signals:

- IR port of the RCX
- IR Tower (serial or USB)
- LEGO remote control (sender only)
- Light sensor (receiver only)

The RCX IR port and the IR tower are the most interesting for IR communication.

Message buffer

The RCX implements a protocol based on packets of bytes. Each packets begins with an opcode byte, followed by parameter bytes. For the meaning of opcode see Kekoa Proudfoot's "RCX Internals". Most of the opcodes only make sense to the LEGO firmware, not LeJOS.

Opcode 0xF7 ("set message buffer") is useful here. It is followed by one parameter byte, the message value. The RCX has a 1–byte message buffer. If is receives a packet with opcode 0xF7, it stores the parameter byte (= message value) in the message buffer, overwriting the previous content.

The message buffer remains unchanged, until it is read (and consumed) by the user program, or overwritten by a new message.

RCX to RCX

The most basic operation transmits bytes from one RCX to another. Two methods send and receive data:

`boolean sendPacket(byte[] buffer, int offset, int count)`
: Sends a apcket to the receiver. You could send any opcode and data bytes, but opcode 0xF7 is easy to handle (see above).

`int readPacket(byte[] buffer)`
: Reads the packet last received and returns the length of the packet (number of bytes). If nothing was received, 0 is returned.
The packet is consumed by this call. The next call to `readPacket` returns 0, unless a new packet arrived.

Two more methods are useful:

`boolean isPacketAvailable()`
: Checks if a packet was received and is available for reading with `readPacket`.

`waitTillSent()`
: Blocks after a call to `sendPacket`, until the message is really out.

Here are two programs that exchange one message. This is the sender `SendBytes`. Here is the receiver `ReadBytes`.

When a message arrives while the receiver is busy doing something else, it is stored in the message buffer. It does not get lost.

The receiver consumes a message with `readPacket`. `isPacketAvailable` will return false, and another call of `readPacket` will return 0.

Message rate

These sender and receiver programs transmit messages in a tight loop. They transmit at about 30 messages/second.

Listeners

The IR port receives messages at unknown points of time. Messages are asynchronous events, like sensor changes. As for sensors, polling is a poor way to wait for messages. Again, you can register a listener. Its callback method is triggered when a message arrives.

The interface `SerialListener` defines a single method prototype, `packetAvailable`. A serial listener implements this interface and defines the required method. Here is example program `ReadListener`.

Range

The RCX and the tower can send IR signals at two different power levels, short range and long range.

The old serial IR tower has a mechanical switch to set the range. The newer USB tower has no switch.
?? how to set IR range on the USB tower?

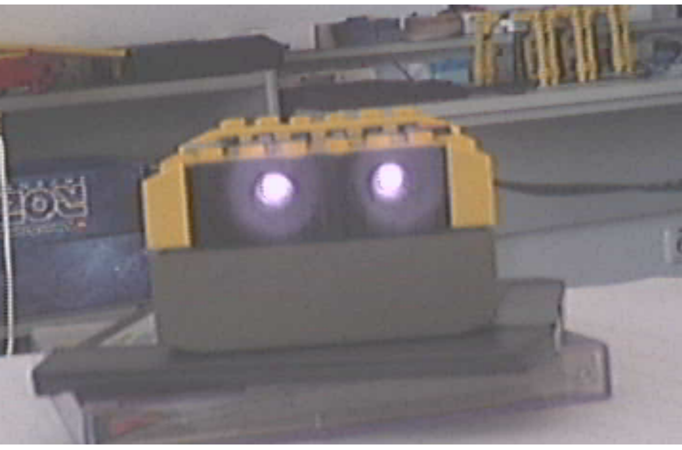
The RCX's range setting is software controlled. Use methods `setRangeShort` and `setRangeLong` to set the level.

Note that the power level only affects the sender. The receiver always receives at the same sensitivity. To get a feeling for the different ranges, use the programs from above.

The sender ist set to short range (default), and placed face to face to the receiver in a room lit by daylight. Then the RCXs are slowly moved apart. Signals are transmitted reliably over a distance of 1 meter; at 1.5 meters many signals are lost, at 2 meters no signals are received.

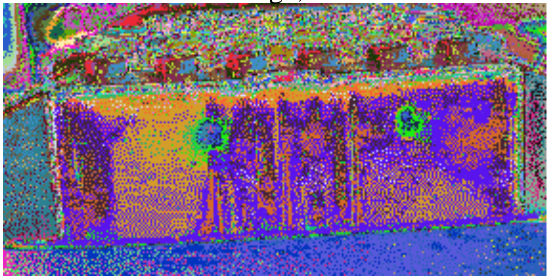
At long range, signals are transmitted reliably over a distance of 7+ meters. I had no more space to test longer distances.

Another way to compares ranges is to look at the IR emitting diodes. The human eye cannot see IR light, but my webcam can. Here is an image of an RCX sending messages at long range:



Here is the same RCX sending at short range. You probably see nothing.

Below is the same image, converted to false colors. Now you can see the IR diodes:



Host to RCX

RCX to Host

IR Messages from the IR tower and the Remote Control

IR Tower

As stated above, messages have an opcode of 0xF7 (= 247). This is true for messages from another RCX or from the IR tower. The IR tower sends several copies of the same packet.

Here is program `MsgReceiver` to demonstrate this. Use NQC for a simple way to send messages from the IR tower:

```
$ nqc -msg 23
```

You will see the counter go up from 1 to 5 steps at a time.

LEGO Remote Control

The LEGO Remote Control uses a different packet format. The opcode is 0xD2 (= 210), followed by two parameter bytes.

byte 1	byte 2	meaning
0	0	button released
0	1	message 1
0	2	message 2
0	4	message 3

It continuously emits messages, as long as a button is pressed. When a butten is released, a message with two 0 parameter bytes is sent.

Here is program `AnyMsgReceiver` to receive IR messages from another RCX, an IR tower, or the LEGO remote control.

Debugging

Errors happen at different levels.

Syntax errors and ***static semantics errors*** are not a problem. They are detected by the compiler.

Dynamic semantics errors happen at runtime. In Java they raise

Logical errors are hardest to detect and correct. The programs runs, but just doesn't do what it is supposed to do. Your friends are the beeper and the LCD display.

?? lejos–emu

Exceptions

If a Java program throws an unhandled excpetion in the RCX, the JVM buzzes, then halts the program and dumps some information in the LCD display.

Run this program. Of course, it bombs out immediately:

```
class BombOut
{
    public static void main(String[] args)
    {
        int i = 1;
        int j = 0;
        int k = i/j;
    }
}
```

The LCD display shows:

The first four digits ("`0000`") are the method index. The last digit ("`0`") is the class index mod 10. To find out what these numbers mean, make `lejos` dump the contents of the downloaded image:

```
$ lejos -o /dev/null -verbose BombOut
Class 0: java/lang/Object
Class 1: java/lang/Thread
Class 2: java/lang/String
Class 3: java/lang/Throwable
Class 4: java/lang/Error
Class 5: java/lang/OutOfMemoryError
Class 6: java/lang/NoSuchMethodError
Class 7: java/lang/StackOverflowError
Class 8: java/lang/NullPointerException
Class 9: java/lang/ClassCastException
Class 10: java/lang/ArithmeticException
Class 11: java/lang/ArrayIndexOutOfBoundsException
Class 12: java/lang/IllegalArgumentException
Class 13: java/lang/InterruptedException
Class 14: java/lang/IllegalStateException
Class 15: java/lang/IllegalMonitorStateException
Class 16: java/lang/ThreadDeath
Class 17: BombOut
Class 18: java/lang/System
Class 19: java/lang/RuntimeException
Class 20: java/lang/Exception
Class 21: java/lang/Runtime
Signature 0: main([Ljava/lang/String;)V
Signature 1: run()V
Signature 2: ()V
Signature 3: ()V
Signature 4: notify()V
    ...lines omitted here...
Signature 49: valueOf(Ljava/lang/Object;)Ljava/lang/String;
Signature 50: ([CII)V
Signature 51: arraycopy([CI[CII)V
Signature 52: getRuntime()Ljava/lang/Runtime;
```

First, find method number `0000` in the list. This is `main([Ljava/lang/String;)V;`. So the exception was thrown from method `main`. (surprise, surprise)

Next, look at all classes with numbers % 10 == 0. Candidates are

0 = `java/lang/Object`,

10 = `java/lang/ArithmeticException`, and

20 = `java/lang/Exception`.

We know that `ArithmeticException` is it. You need a little creative guessing here.

List of example programs

program	*purpose*
`ActionThread`	Listeners start actions by interrupting a single action thread.
`ActionThreads`	Listeners start actions by creating and starting a new thread on every invocation. Memory leak.
`AnyMsgReceiver`	Receive IR messages from another RCX, the IR tower or the LEGO Remote control.
`AudibleSounds`	Play tones of all frequencies to test audible sounds.
`BombOut`	Throw exception.
`ButtonClick`	Register a button listener.

Chars	Display character approximations in the LCD display.
DisplayDecimalPoint	Set the decimal point in the LCD display.
DisplayNumbers	Show numbers in the LCD display.
DisplaySignedNumbers	Show signed numbers in the LCD display.
GoSlow	Reduce motor speed by rapidly switching on and off.
GoStopBack	Control motor.
Hello	Display "Hello" in the LCD display.
LCDShowLight	Continuously display light sensor reading in the LCD display.
MsgReceiver	Receive IR messages from another RCX.
ReadBytes	Read one IR message.
ReadBytesFast	Repeatedly read IR messages.
ReadListener	Register IR message listener.
RepeatAction	Listeners start actions by interrupting an action thread. Interupted actions are repeated.
RotationSpeed	Display rotation speed as rpm value.
Segments	Control all LCD display segments.
SendBytes	Send one IR message.
SendBytesFast	Repeatedly send IR messages.
SensorThread	Sleep in a sensor listener, blocking subsequent calls.
SensorThreadNap	Same as `SensorThread`, using helper method to sleep.
ShowAndBeep	Registers two listeners at one sensor.
ShowPulses	Display touch sensor count in LCD display.
ShowRotation	Continuously display rotation sensor reading in the LCD display.
ShowTouch	Show touch sensor reading in the LCD display using a tight polling loop.
ShowTouchAnonymous	Use anonymous listener class to show touch sensor reading in the LCD display.
ShowTouchCallback	Use a listener class to show touch sensor reading in the LCD display.
ShowTouchDelay	Use delay to reduce CPU load in sensor polling loop.
SoundBuffer	Make limited size of sound buffer audible.
SystemSound	Play system sound without buffering.
Text	Show text in the LCD display.
UninterruptibleAction	Listeners start actions by interrupting an action thread. Ignore interrupts while performing an action.

Document history

2002–04–19 Initial version, incomplete

2002–04–26 Added page "Sound"

2002–07–25	Added page "LCD Display" Linked zip, pdf files
2002–04–XX	Cleaned up examples
2002–07–20	Added page "IR Messages from the IR tower and the Remote Control"
2002–07–25	Added page "LCD Display"
2002–07–25	Modified page "Listeners and threads", according to ideas from Paul Andrews
2002–07–26	Moved examples from text to links, added page "Examples list"

Seitenverzeichnis

Using LeJOS

- Installation
  - Getting LeJOS
  - Building
  - Making LeJOS use Jikes
  - API Docs
- Basic Steps
  - Firmware Download
  - Compiling
  - Downloading
  - Anatomy of `Hello.java`
- Getting information in and out
- Motors
- Sensors
  - Sensor Configuration
  - Touch Sensor
  - Light Sensor
  - Rotation Sensor
- Listeners
  - Polling vs. Callback
  - Using Listeners
  - Multiple Listeners
  - Sensor threads
  - Listeners and threads
  - Other listeners
- Battery level
- Buttons
- Time
- Sound
- LCD Display
- IR Communication
  - Message buffer
  - RCX to RCX
  - Host to RCX
  - RCX to Host
  - IR Messages from the IR tower and the Remote Control
- Debugging
  - Exceptions
- List of example programs
- Document history