

# **The leJOS NXJ Tutorial**

## Introduction to leJOS and this Tutorial

This tutorial will teach you how to use leJOS NXJ to run Java programs on your LEGO MINDSTORMS NXT ®.

leJOS NXJ is a firmware replacement for the NXT and you will need to flash the leJOS firmware replacing the existing LEGO Mindstorms firmware. (Don't worry you can put the original firmware back whenever you want).

The tutorial gives step by step instructions for setting up the leJOS NXJ software up on your PC and replacing the firmware on the NXT. It covers all supported PC operating systems: Microsoft Windows, MAC OS X and Linux.



The tutorial will teach you how to write Java programs that control a variety of different types of robots. You will learn how to program all the sensors and motors that come with the NXT.

leJOS is not for beginners. You will need to understand at least the basics of the Java language to use it. However, it is much more powerful than the graphical programming environment that comes with the NXT: you will be able to program robots with more complex and interesting behaviors.

leJOS NXJ gives you all the power of the Java language. You can use a variety of data types including strings, floating point number and arrays, including multi-dimensional arrays. You can do complex mathematical calculations using all the standard Java math functions. You can program concurrent threads, and you can use Java events.

Java is an Object Oriented environment that lets you define your own classes and objects. This makes program simple to understand but capable of any degree of complexity.

leJOS NXJ has been used to program many sophisticated robots for the LEGO Mindstorms NXT. See <http://lejos.sourceforge.net/forum/viewforum.php?f=5> for some examples. It is very popular with universities and university students.

The tutorial will lead you on to more advanced topics such as using third-party sensors communicating with the PC, other NXTs, mobile phones and other Bluetooth devices. It will teach you advanced robotic programming techniques such as behaviour programming

**What is  
leJOS NXJ?**

# What is leJOS NXJ?



leJOS NXJ is a Java programming environment for the LEGO MINDSTORMS NXT ®. It allows you to program LEGO ® robots in Java.

It consists of:

- Replacement firmware for the NXT that includes a Java Virtual Machine.
- A library of Java classes (classes.jar) that implement the leJOS NXJ Application Programming Interface (API).
- A linker for linking user Java classes with classes.jar to form a .binary file that can be uploaded and run on the NXT.
- PC tools for flashing the firmware, uploading programs, debugging, and many other functions.
- A PC API for writing PC programs that communicate with leJOS NXJ programs using Java streams over Bluetooth or USB, or using the LEGO Communications Protocol.
- Many sample programs

As leJOS is a firmware replacement, the new firmware leJOS NXJ firmware must be flashed onto the NXT, and will replace the standard LEGO MINDSTORMS firmware. This wipes out any files currently held on the LEGO firmware. The LEGO firmware can be restored using the LEGO supplied software.

leJOS is an open source project hosted in the sourceforge repository. It was originally created from the TinyVM project that implemented a Java VM for the LEGO Mindstorms RCX system. The RCX version of leJOS proved very popular with the LEGO Mindstorms Robotic Inventions Systems owners, and close to 200,000 downloads of all versions of leJOS have been done. The original author of TinyVM and the RCX version of leJOS was Jose Solorzano.

The NXT has given the leJOS development team the opportunity to greatly expand the capability of leJOS.

## What are the advantages of leJOS NXJ?

There are many advantages of using leJOS NXJ rather than the NXT-G or other programming environments for the NXT. These include:

- It uses the industry-standard Java language.
- It provides object-oriented programming.
- It is an open source project with many contributors.
- It allows you a choice of professional Integrated Development Environment such as Eclipse and Netbeans that support syntax directed editors and many other features.
- It has cross platform support – Windows, Linux, Mac OS X.
- It is much faster than NXT-G.
- It has full support for Bluetooth.
- It provides highly accurate motor control.
- It has advanced navigation support.
- It provides Behavior classes that support the subsumption architecture for ease of programming of complex robot behaviors.
- It supports third party sensors.
- It supports remote monitoring and tracing of your leJOS NXJ program from the PC.
- It provides trigonometry and other Math functions
- It supports the J2ME LCD user Interface including many graphics functions.
- It supports multithreading.
- It supports listeners and events.
- It supports safe memory management with garbage collection
- It has USB support including Java streams over USB and USB debugging.
- It supports standard Java Communication streams
- It has a flash file system accessed by the standard java.io classes.
- It supports data logging and remote capturing of the logs.
- It has sound support including playing 8-bit WAV files
- It provides dozens of sample programs.
- It supports remote execution from the PC using iCommand.
- The Web site has online forums to help solve any problems you might have, to share projects ideas, and to communicate with the development team.
- It has telerobotics support via standard TCP/IP sockets.
- It supports NXT to NXT Bluetooth communications.
- It supports Bluetooth communication with other devices.
- It supports two-way communication with RCX via third party adapters such as the Mindsensors NRLink.
- It provides compatibility with Lego Communications Protocol, so that many tools that work with the standard LEGO firmware, also work with leJOS.
- It has an easy to use menu system.
- It is widely used by universities and other education establishments.
- It has computer Vision and speech support via iCommand.

# **Getting Started on Microsoft Windows**

## Getting started on Microsoft Windows

This section tells you how to get started if your PC runs Microsoft Windows. If you use Linux or MAC OS X, see the sections at the end of this document.

To get started on Microsoft Windows you will need installed on your PC:

- The LEGO Mindstorms software.
- The libusb-win32 filter driver.
- A Java Standard Edition SDK.
- Apache ant (optional)
- The leJOS NXJ software.

You will then need to set up environment variables on your PC and start a command window to type commands into.

When all that is done, your PC is ready. You then need to flash the leJOS NXJ firmware, and after that you can compile and run your first program.

These steps are described in the sections below.

### LEGO Mindstorms software

You will need the LEGO Mindstorms software installed on your PC, as its USB driver is used by leJOS. Follow the LEGO instructions to install it.

### Libusb

You will also need the libusb-win32 filter driver – you can download it from <http://libusb-win32.sourceforge.net/#downloads>.

On Windows XP systems, you can just download it and execute it.

On Windows Vista systems, however, you must install it in Windows XP compatibility mode. To do this:

1. Download libusb-win32-filter-bin-0.1.12.1.exe (the version current @ 27/09/07)
2. Right click on this file. Select Properties | Compatibility Click the "Run this program in compatibility mode" box and select "Windows XP (Service Pack2)" from the drop down list.
3. Right click again and select "Run as Administrator". Follow the installation instructions.

You should install libusb in a folder that does not have spaces in its name – for example, do not install it in "Program Files".

Run the libusb test program and it will list the usb devices plugged into your computer.

## Java SDK

You will need a Java Standard Edition SDK on your PC. You can download the latest from <http://java.sun.com/>. Follow the instructions for installing it. leJOS NXJ works has been tested with versions 1.5 and 1.6, but will not work with earlier version.

## Apache ant

Apache ant is useful for running the leJOS samples. If you have a Java IDE on your system, such as Eclipse, it might already be installed. You can download ant from <http://ant.apache.org/bindownload.cgi>. Installing ant is simple – you just unzip it into a folder of your choice (e.g. c:\ant). leJOS needs ant 1.7 or later.

## Installing leJOS

You can download leJOS NXJ for Windows from [http://lejos.sourceforge.net/p\\_technologies/nxt/nxj/downloads.php](http://lejos.sourceforge.net/p_technologies/nxt/nxj/downloads.php). Unzip it to a folder of your choice (e.g. c:\lejos0.6). Note that it creates a subfolder called lejos\_nxj.

## Setting up environment variables

You need to set:

Env. Var.	Value	Example
NXJ_HOME	the folder you installed leJOS NXJ into	C:\lejos0.6\lejos_nxj
JAVA_HOME	the folder you installed the Java SDK into	C:\Program Files\Java\jdk1.6.0_06
ANT_HOME	the folder you installed ant into	C:\ant
PATH	Add the bin folders for Java, leJOS and ANT	%NXJ_HOME%\bin;%JAVA_HOME%\bin;%ANT_HOME%\bin;%PATH%

You can set these environment variables for all users on the PC by going to Control Panel > System > Advanced > Environment Variables and creating them or editing existing values.

## Using a Command Window

You can start a command window by Start > Run and typing cmd.

Type *set* to list environment variables and check they are all set up correctly.



## Flashing the NXJ firmware

Make sure your NXT is attached to the PC by its USB cable, and switch it on by pressing the orange button.

Then in your command window, type ***nxjflash*** to flash the leJOS NXJ firmware. You will see some messages on your command window, and the NXT should show the leJOS splash screen and then the leJOS menu.

# **Compiling and running your first program**

## Compiling and running your first program

Java programs need to be compiled to class files before they can be run. For leJOS NXJ, all the class files that are to be run on the NXT needed to be linked to produce a binary file (with the extension .nxj) and this must then be uploaded to the NXT.

To run a sample program, such as the View.java sample, follow these steps:

Start a command window, and change directory to the View sample folder:

```
cd %NXJ_HOME%\samples\View
```

Compile the program with the ***nxjc*** command:

```
nxjc View.java
```

Then link, upload and run it with the ***nxj*** command:

```
nxj -r View
```

You should see the menu of the View sample on your NXT.

# **Setting up the Eclipse IDE**

## Setting Up the Eclipse IDE

Programming for leJOS NXJ is best done using an Integrated Development Environment. IDEs have syntax-directed editors that immediately show you any syntax errors in your program, rather than waiting until you compile the program and then showing a list of errors. This, together with color coding of the source, automatic formatting of the code, prompting for method names and signatures, expanding and collapsing parts of your program, and many other editing features, makes creating your program a much faster and more enjoyable experience. But the advantages of the IDE do not end there: they also help you with creating and building projects, debugging, generating documentation, and creating user interfaces. Java IDEs put all the Sun Java tools and a variety of third-party tools at your fingertips. They make supporting new tools simple, either by use of plug-ins or by integration of external tools.

IDEs are easy to set and use and you should use them for all your leJOS programming – even the simplest of projects.

This tutorial concentrates the Eclipse Java Integrated Development Environment, as this is currently the most popular one for leJOS programmers, but there are many other Java IDEs, such as Netbeans and BlueJ, each with their own strengths,

There are three ways of building leJOS programs in Elclipse:

- Using external tools
- Using ant build files
- Using a leJOS plug-in

It is worth understanding all of these techniques as they each have there advantages, and a combination of the techniques may well work best for your projects.

### ***Setting up Eclipse***

### ***Creating a leJOS project***

### ***Creating external tools***

### ***Using ant build files***

### ***Installing and using the leJOS plugin***

# **Programming with leJOS NXJ**

# **Writing Your First Program**

## Writing your first leJOS NXJ program

Let us start with a simple “Hello World” program. We will create a HelloWorld class in the default java package:

```
public class HelloWorld
{
}
```

leJOS requires the standard **main** method for the program entry point:

```
public class HelloWorld
{
    public static void main (String[] args)
    {
    }
}
```

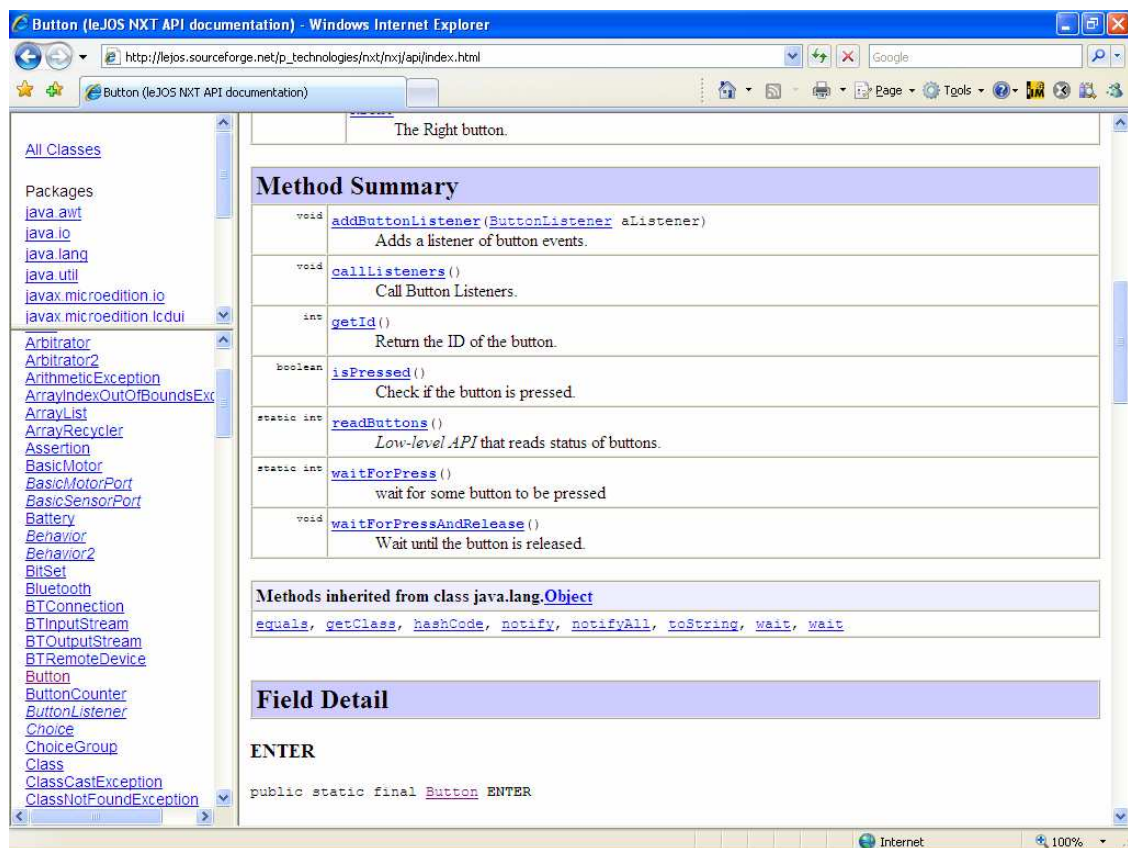
Recent versions of leJOS NXJ support the standard java **System.out.println** method and scroll the output on the NXT LCD screen.

```
public class HelloWorld
{
    public static void main (String[] args)
    {
        System.out.println("Hello World");
    }
}
```

If you run this program as it is, it will display Hello World” and then immediately return to the menu, so you will not be able to see what is displayed (unless you are very quick).

We either need the program to sleep for a while to allow the text to be read, or to wait for a button to be pressed. Let us wait for a button to be pressed. To do this we need to include the leJOS NXJ Button class in the program. Button is in the lejos.nxt package. We can either include lejos.nxt.Button or lejos.nxt.\* to allow any of the standard lejos.nxt classes to be used in the program. The Button class has a method waitForPress() that waits for any button to be pressed. You can find out what methods a class supports by looking at the API documentation:





The API documentation is on the leJOS wseb site and included in the leJOS download in `lejos_nxt/docs/apidocs/index.html`.

The complete HelloWorld program is:

```
import lejos.nxt.*;

public class HelloWorld
{
    public static void main (String[] args)
    {
        System.out.println("Hello World");
        Button.waitForPress();
    }
}
```

# **Controlling the Hardware**

# **LCD, Buttons and Sound**

## LCD, Sound and Buttons

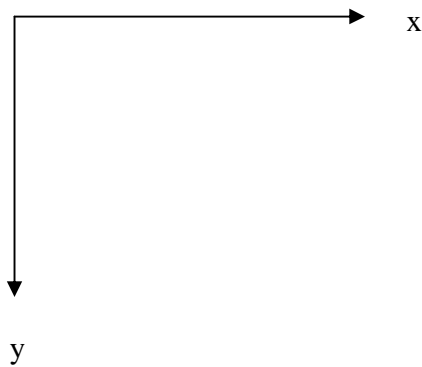
The NXT can be used as a simple computer with an LCD display, 4 buttons and sound capabilities. These functions are supported by the LCD, Button, and Sound classes.

### **LCD**

The LCD can be used in text mode and graphics mode.

#### **Text LCD methods**

As a text display, the NXT LCD screen is 16 characters wide and eight characters deep. It is addressed using (x, y) co-ordinates as follows:



x ranges from 0 to 15, and y from 0 to 7.

The methods to write to the LCD in text mode are:-

- `public static drawString(String str, int x, int y);`

This draws a string of text to the LCD screen starting at text co-ordinate (x, y).

- `drawString(String str, int x, int y, boolean invert);`

This variant of `drawString` supports inverting the text drawing white characters on a black background.

- `public static drawInt(int i, int x, int y);`

This draws an integer starting at co-ordinate (x,y). The integer is left aligned and takes up as many characters as are necessary.

- `public static drawInt(int i, int places, int x, int y);`

This variant of `drawInt` right-aligns the integer and always uses the number of characters indicated by *places*.

- `drawChar(char c, int x, int y, boolean invert);`

This draws a character at text co-ordinate (x, y) with optional inversion of the character.

- `public static clear();`

Clears the display.

Example:

```
import lejos.nxt.*;
import java.io.*;

public class LCDTest
{
    public static void main (String[] args)
        throws Exception
    {
        LCD.drawString("Free memory:", 0, 0);
        LCD.drawString("RAM:", 1, 1, true);
        LCD.drawInt((int) System.getRuntime().freeMemory(), 6, 8, 1);
        LCD.drawString("Flash:", 1, 2, true);
        LCD.drawInt(File.freeMemory(), 6, 8, 2);
        Thread.sleep(2000);
    }
}
```

## Buttons

The NXT buttons are accessed by static fields:

- `Button.ENTER`
- `Button.ESCAPE`
- `Button.LEFT`
- `Button.RIGHT`

To test if a button is pressed, you use:

- `public final boolean isPressed();`

Example:

```

import lejos.nxt.*;

public class ButtonPresses
{
    public static void main (String[] args)
        throws Exception
    {
        while (true) {
            LCD.clear();
            If (Button.ENTER.isPressed()) LCD.drawString("ENTER",0,0);
            if (Button.ESCAPE.isPressed()) LCD.drawString("ESCAPE",0,0);
            if (Button.LEFT.isPressed()) LCD.drawString("LEFT",0,0);
            if (Button.RIGHT.isPressed()) LCD.drawString("RIGHT",0,0);
        }
    }
}

```

To wait for a specific button to be pressed and released, you use:

- `public final void waitForPressAndRelease() throws InterruptedException;`

Example:

```

import lejos.nxt.*;

public class ButtonTest
{
    public static void main (String[] args)
        throws Exception
    {
        Button.ENTER.waitForPressAndRelease();
        LCD.drawString("Finished", 3, 4);
        Thread.sleep(2000);
    }
}

```

To wait for any button to be pressed, you do:

- `public static int waitForPress();`

The returns the id of the button that is pressed.

To specify a listener to listen for button events for this button, you:

- `public synchronized void addButtonListener (ButtonListener aListener)`

See “Listeners and Events” below for how button listeners work.

To read the current state of all the buttons, you do:

- `public static native int readButtons();`

## Sound

There is a static method to play a tone:

```
public static playTone(int aFrequency, int aDuration);
```

Example:

```
import lejos.nxt.*;

public class Tune {

    // NOTE: This tune was generated from a midi using Guy
    // Truffelli's Brick Music Studio www.aga.it/~guy/lego
    private static final short [] note = {
        2349,115, 0,5, 1760,165, 0,35, 1760,28, 0,13, 1976,23,
        0,18, 1760,18, 0,23, 1568,15, 0,25, 1480,103, 0,18, 1175,180, 0,20, 1760,18,
        0,23, 1976,20, 0,20, 1760,15, 0,25, 1568,15, 0,25, 2217,98, 0,23, 1760,88,
        0,33, 1760,75, 0,5, 1760,20, 0,20, 1760,20, 0,20, 1976,18, 0,23, 1760,18,
        0,23, 2217,225, 0,15, 2217,218};

    public static void main(String [] args) {
        for(int i=0;i<note.length; i+=2) {
            final short w = note[i+1];
            final int n = note[i];
            if (n != 0) Sound.playTone(n, w*10);
            try { Thread.sleep(w*10); } catch (InterruptedException e) {}
        }
    }
}
```

There are methods that use playTone to play a variety of sounds. These are compatible with the RCX version of leJOS:

- public static void systemSound (boolean aQueued, int aCode);

The aQueued parameter is ignored on the NXT.

The values of code are:

code = 0	Short beep
code = 1	Double beep
code = 2	Descending arpeggio
code = 3	Ascending arpeggio
code = 4	Long, low buzz

There are also methods for each of the system sounds:

- public static void beep();
- public static void twoBeeps();
- public static void beepSequence();
- public static void beepSequenceUp();
- public static void buzz();

This is also a method to produce a rest when playing a tune:

- `public static void pause(int t);`

leJOS NXJ can also play 8-bit WAV files. To play these you do:



# **Controlling the Sensors**

# Controlling the Sensors

The NXT comes with four sensors; the touch sensor, the sound sensor, the light sensor and the ultrasonic sensor.

## Touch Sensor

To use a touch sensor, you create an instance of it attached to a sensor port, using the constructor:

- `public TouchSensor(ADSensorPort port);`

The port is usually `SensorPort.S1`, `S2`, `S3` or `S4`, but it could be a remote sensor port (see the `RemoteNXT` class) or a port attached to a third-party port expander. It can be an instance of any class that implements the `ADSensorPort` interface. (“AD” stands for “Analog/Digital” – the touch sensor is an example of an Analog to Digital sensor).

To test if the touch sensor is pressed, you use the `isPressed()` method:

- `public boolean isPressed();`

Example:

```
import lejos.nxt.*;

public class TouchTest {
    public static void main (String[] args) throws Exception
    {
        TouchSensor touch = new TouchSensor(SensorPort.S1);

        while (!touch.isPressed());
        LCD.drawString("Finished", 3, 4);
        Thread.sleep(2000);
    }
}
```

## Light Sensor

To use a light sensor, you create an instance of it attached to a sensor port, using the constructor:

- `public LightSensor(ADSensorPort port);`

The port is usually `SensorPort.S1`, `S2`, `S3` or `S4`, but it could be a remote sensor port (see the `RemoteNXT` class) or a port attached to a third-party port expander. It can be an instance of any class that implements the `ADSensorPort` interface. (“AD” stands for “Analog/Digital” – the light sensor is an example of an Analog to Digital sensor).

Example:

```
import lejos.nxt.*;

public class LightTest {
    public static void main (String[] args)
        throws Exception
    {
        LightSensor light = new LightSensor(SensorPort.S1);

        while (true) {
            LCD.drawInt(light.readValue(), 4, 0, 0);
            LCD.drawInt(light.readNormalizedValue(), 4, 0, 1);
            LCD.drawInt(SensorPort.S1.readRawValue(), 4, 0, 2);
            LCD.drawInt(SensorPort.S1.readValue(), 4, 0, 3);
            Thread.sleep(2000);
        }
    }
}
```

## Sound Sensor

The sound sensor is an Analog/Digital sensor. It supports two modes:

To use a sound sensor, you create an instance of it attached to a sensor port, using the constructor:

- `public SoundSensor(ADSensorPort port);`

The port is usually `SensorPort.S1`, `S2`, `S3` or `S4`, but it could be an instance of any class that implements the `ADSensorPort` interface.

Example:

```

import lejos.nxt.*;

public class SoundScope {

    public static void main (String[] args)    throws Exception {
        SoundSensor sound = new SoundSensor(SensorPort.S1);

        while (!Button.ESCAPE.isPressed()) {
            LCD.clear();
            for(int i=0;i<100;i++) {
                LCD.setPixel(1,i,60 - (sound.readValue()/2));
                Thread.sleep(20);
            }
        }
    }
}

```

The above example gives a graphical display of the way the sound reading varies over a two-second period.

## Ultrasonic Sensor

The Ultrasonic Sensor is an I2C sensor. The classes that support I2C sensors all extend the I2CSensor class.

Example:

```

import lejos.nxt.*;

public class SonicTest {
    public static void main(String[] args) throws Exception {
        UltrasonicSensor sonic = new
            UltrasonicSensor(SensorPort.S1);

        while(!Button.ESCAPE.isPressed()) {
            LCD.clear();
            LCD.drawString(sonic.getVersion(), 0, 0);
            LCD.drawString(sonic.getProductID(), 0, 1);
            LCD.drawString(sonic.getSensorType(), 0, 2);
            LCD.drawInt(sonic.getDistance(), 0, 3);
        }
    }
}

```

# **Controlling the Motors**

## Controlling the Motors

Motors connected directly to the NXT motor ports can be accessed using static variables:

- Motor.A
- Motor.B
- Motor.C

The most basic methods for controlling motors are:

- public setSpeed(int speed);
- public forward();
- public backward();
- public stop();
- public flt();

*flt()* puts the motor in float mode and it will gracefully come to rest. *stop()* stops the motor immediately.

By default, the speed of a motor is regulated by the motor's regulation thread. This keeps the motor running at the desired speed, by changing the power applied to the motors.

You can determine how well this is working by getting the actual speed of the motor:

- public int getActualSpeed()

If you do not want speed regulation, you can call:

- public void regulateSpeed(boolean yes)

and switch regulation off. When motors are not being regulated, you may wish to set the power directly, rather than to set the speed. To do this you can call:

- public synchronized void setPower(int power)

The direction of the motor can be changed with:

- public void reverseDirection();

The built-in tachometer in the NXT motor can be read and reset by:

- public void resetTachoCount()
- public int getTachoCount()

The tachometer can be used to rotate a motor by an angle, in degrees. The angle can be negative.

- `public void rotate(int angle);`
- `public void rotate(int angle, boolean immediateReturn);`

An absolute rather than relative angle can also be specified:

- `public void rotateTo(int limitAngle);`
- `public void rotateTo(int limitAngle,boolean immediateReturn);`

You can test if a motor is moving by:

- `public boolean isMoving();`

This is useful to test if the motor has finished rotating. `isMoving()` returns true when the Motor is moving for any reason (e.g. `forward()` or `backward()` have been called), and not only if a rotate operation is in place. To specifically test to see if a rotate operation is currently in progress, call:

- `public boolean isRotating()`

The angle that you are currently rotating to can be determined by calling:

- `public int getLimitAngle()`

The regulation thread also implements smooth acceleration to prevent jerking motion when speed is increased or decreased. This can be switched on and off, by:

- `public void smoothAcceleration(boolean yes)`

There are set of methods for getting information about the current state of the motor:

- `public int getSpeed()`
- `public int getMode()`
- `public int getPower()`
- `public boolean isRegulating()`
- `public boolean isForward()`
- `public boolean isBackward()`
- `public boolean isFloating()`

As well as controlling speed regulation and smooth acceleration, the regulation thread also controls rotation and bringing the motors to a smooth stop at the limit angle. If a motor is being used in a very simple way that does not require any of these regulation functions, you can shut down the regulation thread by calling:

- `public void shutdown()`

# **Other**

## ***Hardware***



## ***Reading the Battery***

There are two static methods to get the battery voltage:

- `public static native int getVoltageMilliVolt();`
- `public static float getVoltage();`

Example:

```
import lejos.nxt.*;

public class BatteryTest {
    public static void main (String[] args) throws Exception {
        LCD.drawString("Battery: " + Battery.getVoltage(),0,0);
        Thread.sleep(2000);
    }
}
```

# **Controlling wheeled vehicles**

# Controlling Wheeled Vehicles

One of the most common form of robots created using the Mindstorms NXT are wheeled vehicles, and leJOS contains several classes that specifically support wheeled vehicles.

## Pilot

The Pilot class is used to control vehicles with two independently driven wheels, that can turn on the spot.

It steers such a vehicle using two regulated motors. A pilot does not keep track on the position or bearing of a robot – that is the job of a Navigator.

The Pilot constructor needs to know the diameter of the wheels of the vehicle and the width of the track, i.e. the distance between the centers of the tracks of the two wheels. These measurements can be in any units as it is the ratio between them that is important, not the measurement units.

The Pilot constructor also needs to know what Motors are used, and whether driving them forwards drives the robot forward or backward.

The constructors are:

- `public Pilot(float wheelDiameter,float trackWidth,Motor leftMotor, Motor rightMotor)`
- `public Pilot(float wheelDiameter,float trackWidth,Motor leftMotor, Motor rightMotor, boolean reverse)`

Use the second constructor if you need to set the *reverse* Boolean.

The main methods of a pilot are:

- `public void setSpeed(int speed);`
- `public void forward()`
- `public void backward()`
- `public void stop()`
- `public void rotate(int angle)`
- `public void rotate(int angle, boolean immediateReturn )`
- `public boolean isMoving()`
- `public void travel(float distance)`
- `public void travel(float distance,boolean immediateReturn)`
- `public void steer(int turnRate)`
- `public void steer(int turnRate, int angle)`
- `public void steer(int turnRate, int angle, boolean immediateReturn)`

*setSpeed* sets the speed of the robot in degrees per second on wheel rotation.

*forward* moves the robot forward until it is stopped or another movement method is called. *backward* moves the robot backwards. *stop* stops the robot.

**rotate** rotates the robot on its axis by the number of degrees specified. If you want the thread to do other things while the robot is rotating, use the second variant of the method and set `immediateReturn = true`.

## **CompassPilot**

The CompassPilot is an extension of the Pilot class that implements the same methods, but uses a compass sensor to ensure that the pilot does not deviate from the correct angle.

It needs a HiTechnic or Mindsensors compass sensor plugged in to one of the sensor ports.

The constructors are:

- `public CompassPilot(SensorPort compassPort, float wheelDiameter, float trackWidth, Motor leftMotor, Motor rightMotor)`
- `public CompassPilot(SensorPort compassPort, float wheelDiameter, float trackWidth, Motor leftMotor, Motor rightMotor, boolean reverse)`
- `public CompassPilot(CompassSensor compass, float wheelDiameter, float trackWidth, Motor leftMotor, Motor rightMotor)`
- `public CompassPilot(CompassSensor compass, float wheelDiameter, float trackWidth, Motor leftMotor, Motor rightMotor, boolean reverse)`

## **Navigator**

### **TachoNavigator**

### **CompassNavigator**

# ***Debugging leJOS NXJ Programs***

## Error handling and debugging

leJOS NXJ provides several features for error handling and debugging, including:

- Exceptions
- Data Aborts
- USB Debugging

The Remote Monitoring and Tracing facility, which is described in its own section below, can also be used for debugging.

### ***Exceptions***

Most of the standard Java language exception classes are supported by leJOS, and user can create their own exception classes.

Example:

The following simplified version of the ExceptionTest example demonstrates what happens for an exception that is not caught – in this case an `ArrayIndexOutOfBoundsException` exception.

```
import lejos.nxt.*;

public class ExceptionTest {
    public static void main (String[] args)
    {
        SensorPort p = SensorPort.PORTS[5];
    }
}
```

### ***Data Aborts***

If the leJOS firmware crashes you will normally a Data Abort. The screen shows the PC value where the failure occurred, and other details of the failure.

The screen is something like:

The most common reason for data aborts is executing a file that is not a leJOS NXJ binary, or executing an incomplete leJOS NXJ file.

If you get a data abort in any other case, you should report the error to the leJOS development team by posting the details on the leJOS NXJ forums.

## ***Remote USB Debugging***

Tracing statements can be inserted into leJOS NXJ programs using the `lejos.nxt.Debug` class.

To start debugging, you use one of the static `open()` methods:

- `public static void open(int timeout)`
- `public static void open()`

This waits for the PC based monitor to connect.

To run the debug monitor on the PC, you execute ***nxjdebug***.

This connects to the debugger on the NXT over USB – the USB cable must be connected.

If you use the variant of `open` with a timeout, it waits the specified number of seconds and if the debug monitor has not connected, proceeds without debugging.

Debug statements can be output using the method:

- `public static void out(String s)`

If no successful `open` statement has been executed, the debug output is discarded. If there was a successful output, the string appears on standard out in the window or terminal that ***nxjdebug*** was run from, on the PC.

When debugging is completed, you should call the static `close()` method:

- `public static void close()`

# ***Bluetooth Communications***



# Bluetooth Communications

leJOS supports a multitude of methods of communicating NXT to NXT, PC to NXT, Mobile phone to NXT, NXT to remote Bluetooth device, etc.

## ***Java Data Streams over Bluetooth***

leJOS NXJ supports data streams over Bluetooth and USB.

The initiator program for a Java data stream over Bluetooth can be on a NXT, a PC, a mobile phone or another device that supports the Bluetooth Serial Port Profile (SPP).

### **Receiver**

The receiver program on the NXT wait for a connection by calling one of the `waitForConnection` methods in the `Bluetooth` class.:

- `public static BTConnection waitForConnection(byte[] pin);`
- `public static BTConnection waitForConnection();`

The second version of `waitForConnection` assumes the default pin: "1234".

You need to ensure that Bluetooth power and visibility are on. The leJOS NXJ start-up menu can be used to do this.

Example:

```
BTConnection btc = Bluetooth.waitForConnection();
```

Input and output streams or data input and output streams can then be opened by calling the following methods from the `BTConnection` class:

- `public InputStream openInputStream() throws IOException;`
- `public OutputStream openOutputStream() throws IOException;`
- `public DataInputStream openDataInputStream() throws IOException;`
- `public DataOutputStream openDataOutputStream() throws IOException;`

Example:

```
DataInputStream dis = btc.openDataInputStream();  
DataOutputStream dos = btc.openDataOutputStream();
```

Data items can then be read from the `DataInputStream` by:

- `public final int read(byte b[]) throws IOException`
- `public final int read(byte b[], int off, int len) throws IOException`
- `public final boolean readBoolean() throws IOException`
- `public final byte readByte() throws IOException`
- `public final short readShort() throws IOException`
- `public final int readInt() throws IOException`
- `public final char readChar() throws IOException`
- `public final float readFloat() throws IOException`
- `public String readLine() throws IOException`

Data can be written to the `DataOutputStream` by:

- `public synchronized void write(byte b[], int off, int len) throws IOException`
- `public final void writeBoolean(boolean v) throws IOException`
- `public final void writeByte(int v) throws IOException`
- `public final void writeShort(int v) throws IOException`
- `public final void writeChar(int v) throws IOException`
- `public final void writeInt(int v) throws IOException`
- `public final void writeFloat(float v) throws IOException;`
- `public final void writeChars (String value) throws IOException`

Example:

```
for(int i=0;i<100;i++) {  
    int n = dis.readInt();  
    LCD.drawInt(n,7,0,1);  
    LCD.refresh();  
    dos.writeInt(-n);  
    dos.flush();  
}
```

The `DataInputStream`, `DataOutputStream` and `BTConnection` can then be close using the `close()` method.

The full `BTRceive` example is:

```

public class BTReceive {

    public static void main(String [] args) throws Exception
    {
        String connected = "Connected";
        String waiting = "Waiting...";
        String closing = "Closing...";

        while (true)
        {
            LCD.drawString(waiting,0,0);
            BTConnection btc = Bluetooth.waitForConnection();

            LCD.clear();
            LCD.drawString(connected,0,0);

            DataInputStream dis = btc.openDataInputStream();
            DataOutputStream dos = btc.openDataOutputStream();

            for(int i=0;i<100;i++) {
                int n = dis.readInt();
                LCD.drawInt(n,7,0,1);
                dos.writeInt(-n);
                dos.flush();
            }

            dis.close();
            dos.close();
            Thread.sleep(100); // wait for data to drain
            LCD.clear();
            LCD.drawString(closing,0,0);
            btc.close();
            LCD.clear();
        }
    }
}

```

## NXT Initiator

To initiate a Bluetooth connection from one NXT to another NXT, you first need to add the receiver NXT to the initiator NXT's Bluetooth devices.

To do this, you go to the Bluetooth menu in the leJOS NXJ start-up menu and select "Search". Providing the Bluetooth power is on and visibility is on for the receiving NXT, it will be found and you can select "Add" to add it to the initiator's Bluetooth devices.

To check it is in the Devices list, you can select "Devices" from the Bluetooth menu of the initiator NXT.

You can then create a BTRemoteDevice class on the initiator NXT:

Example:

```

BTRemoteDevice btrd = Bluetooth.getKnownDevice(name);

```

You can connect to the remote device by its address, which you can get by:

- `public byte[] getDeviceAddr()`

You can then connect to the remote device by calling one of the `connect()` methods in the Bluetooth class:

- `public static BTConnection connect(BTRemoteDevice remoteDevice)`
- `public static BTConnection connect(byte[] device_addr)`
- `public static BTConnection connect(byte[] device_addr, byte[] pin)`

Example:

```
BTRemoteDevice btrd = Bluetooth.getKnownDevice(name);

if (btrd == null) {
    LCD.clear();
    LCD.drawString("No such device", 0, 0);
    Thread.sleep(2000);
    System.exit(1);
}

BTConnection btc = Bluetooth.connect(btrd);

if (btc == null) {
    LCD.clear();
    LCD.drawString("Connect fail", 0, 0);
    Thread.sleep(2000);
    System.exit(1);
}
```

Having got a `BTRemoteDevice` object you can open the data input and output streams and read data as in the receiver example above.

The complete `BTConnectTest` example, which works as the initiator program for the `BTReceive` receiver program, is:

```

public class BTConnectTest {
    public static void main(String[] args) throws Exception {
        String name = "NXT";

        LCD.drawString("Connecting...", 0, 0);
        LCD.refresh();

        BTRemoteDevice btrd = Bluetooth.getKnownDevice(name);

        if (btrd == null) {
            LCD.clear();
            LCD.drawString("No such device", 0, 0);
            LCD.refresh();
            Thread.sleep(2000);
            System.exit(1);
        }

        BTConnection btc = Bluetooth.connect(btrd);

        if (btc == null) {
            LCD.clear();
            LCD.drawString("Connect fail", 0, 0);
            LCD.refresh();
            Thread.sleep(2000);
            System.exit(1);
        }

        LCD.clear();
        LCD.drawString("Connected", 0, 0);
        LCD.refresh();

        DataInputStream dis = btc.openDataInputStream();
        DataOutputStream dos = btc.openDataOutputStream();

        for(int i=0;i<100;i++) {
            try {
                LCD.drawInt(i*30000, 8, 0, 2);
                LCD.refresh();
                dos.writeInt(i*30000);
                dos.flush();
            } catch (IOException ioe) {
                LCD.drawString("Write Exception", 0, 0);
                LCD.refresh();
            }

            try {
                LCD.drawInt(dis.readInt(), 8, 0, 3);
                LCD.refresh();
            } catch (IOException ioe) {
                LCD.drawString("Read Exception ", 0, 0);
                LCD.refresh();
            }
        }

        try {
            LCD.drawString("Closing... ", 0, 0);
            LCD.refresh();
            dis.close();
            dos.close();
            btc.close();
        } catch (IOException ioe) {
            LCD.drawString("Close Exception", 0, 0);
            LCD.refresh();
        }

        LCD.clear();
        LCD.drawString("Finished", 3, 4);
        LCD.refresh();
        Thread.sleep(2000);
    }
}

```

## PC Initiator

A PC program can initiate a connection to a NXT and open a Java data stream.

The API on the PC is different to the NXT API. See pcapidocs.

To connect to the NXT, you need a NXTComm object that can be obtained using the NXTCommFactory class:

- `public static NXTComm createNXTComm(int protocol)`

Example:

```
NXTComm nxtComm = NXTCommFactory.createNXTComm(NXTCommFactory.BLUEETOOTH);
```

The reason for using a factory method is that there are several implementations of comms drivers for Bluetooth and USB on the PC and the one that is used depends on what operating system you are using and the contents of the nxj.properties file.

You can connect to the NXT by address or by do a Bluetooth inquiry:

To connect by address, you create a NXTInfo object using the constructor:

- `public NXTInfo(String name, String address)`

Example:

```
NXTInfo nxtInfo = new NXTInfo("NXT", "00:16:53:00:78:48");
```

To find the available NXTs doing a Bluetooth inquiry, you do:

```
NXTInfo[] nxtInfo = nxtComm.search("NXT", NXTCommFactory.BLUEETOOTH);  
  
if (nxtInfo.length == 0) {  
    System.out.println("No NXT Found");  
    System.exit(1);  
}
```

Once you have a NXTInfo object, you can call the open() method of the NXTComm object to connect to the NXT:

- `public boolean open(NXTInfo nxt) throws NXTCommException;`

Once the NXT is open, you can obtain an InputStream and an OutputStream, by calling the getInputStream() and getOutputStream() methods of the NXTComm object:

- `public OutputStream getOutputStream();`
- `public InputStream getInputStream();`

From these you can construct a `DataInputStream` and a `DataOutputStream` and send data to the receiving NXT.

The complete `BTSend` sample is in the `samples` folder.

# **Programming Listeners, Events and Threads**



# Listeners and Events

leJOS implements a listener thread that listens for particular events.

The listener thread supports:

- Button Listeners
- Sensor Port Listeners

Button listeners are used to detect when a button is pressed, whatever your program is doing at the time.

To listen for a press of a specific button, you must call:

Example:

```
import lejos.nxt.*;

public class ListenForButtons
{
    public static void main (String[] args)
    {
        Button.ENTER.addButtonListener(new ButtonListener() {
            public void buttonPressed(Button b) {
                LCD.drawString("ENTER pressed",0,0);
            }

            public void buttonReleased(Button b) {
                LCD.clear(); () ;
            }
        });
        while (true);
    }
}
```

# Java Threads

When a Java program starts, there is a single thread running – the main thread.

Many of the leJOS classes start extra threads running for various purposes, for example:

- Button and SensorPort start a listener thread if listeners are used
- Each Motor has a regulator thread
- The Bluetooth class starts a thread to talk to the separate Bluetooth chip
- Each Timer object starts a timer thread

User program can create their own threads by subclassing Thread, and then using the start method to start the thread. Note that the Runnable interface is not implemented by leJOS, and new threads cannot be created by implementing Runnable classes. User threads in leJOS must subclass Thread and implement the run() method.

Background threads that do not need to terminate in order for the user program to terminate, should be marked as daemon threads by calling setDaemon(true).

When using threads, care should be taken with concurrency issues. When data items are accessed by multiple threads, synchronization is necessary to ensure that data is not read when it is in an inconsistent state.

leJOS supports the standard Java synchronization mechanisms: synchronized methods and synchronized statements using a monitor object. There are some restrictions in the way leJOS handles concurrency and synchronization – see the leKOS NXJ README.html file.

As an example of a leJOS thread, consider the Indicators thread in the leJOS StartUpText menu. This is used to keep the display of the battery level up to date, by reading its value every second, and to indicate when the menu is uploading files or doing other communication from the PC:

```

class Indicators extends Thread
{
    private boolean io = false;

    public void ioActive()
    {
        io = true;
    }

    public void run()
    {
        String dot = ".";
        String [] ioProgress = {".  ", " . ", "  ."};
        int ioIndex = 0;
        int millis;
        while(true)
        {
            try
            {
                if (io)
                {
                    ioIndex = (ioIndex + 1) % ioProgress.length;
                    LCD.drawString(ioProgress[ioIndex], 13, 0);
                    io = false;
                }
                else
                {
                    millis = Battery.getVoltageMilliVolt() + 50;
                    LCD.drawInt((millis - millis%1000)/1000,13,0);
                    LCD.drawString(dot, 14, 0);
                    LCD.drawInt((millis% 1000)/100,15,0);
                }
                Thread.sleep(1000);
            } catch (InterruptedException ie) {}
        }
    }
}

```

The main method starts this thread by:

```

Indicators ind = new Indicators();
ind.setDaemon(true);
ind.start();

```

# **leJOS Utility Classes**

## Utilities

Timer

# **Behavior Programming**

# Behavior programming

## *Programming Behavior with leJOS NXJ*

When most people start programming a robot, they think of the program flow as a series of if-thens, which is reminiscent of structured programming (Figure 1). This type of programming is very easy to get started in and hardly requires any thought or design beforehand. A programmer can just sit at the computer and start typing. The problem is, the code ends up as spaghetti code; all tangled up and difficult to expand. The behavior control model, in contrast, requires a little more planning before coding begins, but the payoff is that each behavior is nicely encapsulated within an easy to understand structure. This will theoretically make your code easier to understand by other programmers familiar with the behavior control model, but more importantly it becomes very easy to add or remove specific behaviors from the overall structure, without negative repercussions to the rest of the code. Let's examine how to do this in leJOS NXJ.

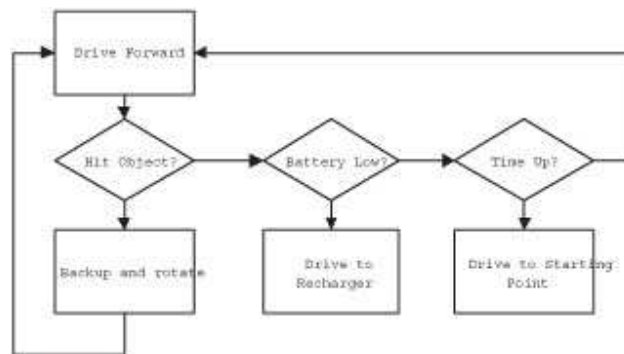


Figure 1: Structured programming visualized.

## *The Behavior API*

The Behavior API is very simple, and is composed of only one interface and one class. The Behavior interface is used to define behaviors. The Behavior interface is very general, so it works quite well because the individual implementations of a behavior vary widely. Once all the Behaviors are defined, they are given to an Arbitrator to regulate which behaviors should be activated. All classes and interfaces for Behavior control are located in the `lejos.subsumption` package. The API for the Behavior interface is as follows.

### **lejos.subsumption.Behavior**

- `boolean takeControl()`

Returns a boolean value to indicate if this behavior should become active. For example, if a touch sensor indicates the robot has bumped into an object, this method should return true.

- void action()

The code in this method initiates an action when the behavior becomes active. For example, if takeControl() detects the robot has collided with an object, the action() code could make the robot back up and turn away from the object.

- void suppress()

The code in the suppress() method should immediately terminate the code running in the action() method. The suppress() method can also be used to update any data before this behavior completes.

As you can see, the three methods in the Behavior interface are quite simple. If a robot has three discreet behaviors, then the programmer will need to create three classes, with each class implementing the Behavior interface. Once these classes are complete, the code should hand the Behavior objects off to the Arbitrator to deal with.

## **lejos.subsumption.Arbitrator**

- public Arbitrator(Behavior [] behaviors)

Creates an Arbitrator object that regulates when each of the behaviors will become active. The higher the index array number for a Behavior, the higher the priority level.

Parameter: an array of Behaviors

- public void start()

Starts the arbitration system.

The Arbitrator class is even easier to understand than Behavior. When an Arbitrator object is instantiated, it is given an array of Behavior objects. Once it has these, the start() method is called and it begins arbitrating; deciding which behaviors should become active. The Arbitrator calls the takeControl() method on each Behavior object, starting with the object with the highest index number in the array. It works its way through each of the behavior objects until it encounters a behavior that wants to take control. When it encounters one, it executes the action() method of that behavior once and only once. If two behaviors both want to take control, then only the higher level behavior will be allowed (Figure 2).



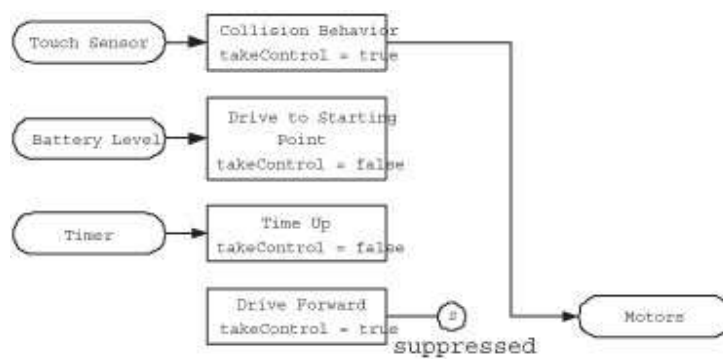


Figure 2: Higher level behaviors suppress lower level behaviors.

Now that we are familiar with the Behavior API under leJOS, let's look at a simple example using three behaviors. For this example, we will program some behavior for a simple robot with differential steering. This robot will drive forward as it's primary low-level behavior. When it hits an object, a high priority behavior will become active to back the robot up and turn it 90 degrees. There will also be a third behavior which we will insert into the program after the first two have been completed. Let's start with the first behavior.

As we saw in the Behavior interface, we must implement the methods `action()`, `suppress()`, and `takeControl()`. The behavior for driving forward will take place in the `action()` method. It simply needs to make motors A and C rotate forward:

```

public void action() {
    Motor.A.forward();
    Motor.C.forward();
}

```

That was easy enough! Now the `suppress()` method will need to stop this action when it is called, as follows:

```

public void suppress() {
    Motor.A.stop();
    Motor.C.stop();
}

```

So far, so good. Now we need to implement a method to tell Arbitrator when this Behavior should become active. As we outlined earlier, this robot will drive forward always, unless something else suppresses it, so this Behavior should always want to take control (it's a bit of a control freak). The `takeControl()` method should return true, no matter what is happening. This may seem counter intuitive, but rest assured that higher level behaviors will be able to cut in on this behavior when the need arises. The method appears as follows:

```
public boolean takeControl() {  
    return true;  
}
```

That's all it takes to define our first Behavior to drive the robot forward. The complete code listing for this class is as follows:

```
import lejos.subsumption.*;  
import lejos.nxt.*;  
  
public class DriveForward implements Behavior {  
  
    public boolean takeControl() {  
        return true;  
    }  
  
    public void suppress() {  
        Motor.A.stop();  
        Motor.C.stop();  
    }  
  
    public void action() {  
        Motor.A.forward();  
        Motor.C.forward();  
    }  
}
```

The second behavior is a little more complicated than the first, but still very similar. The main action of this behavior is to reverse and turn when the robot strikes an object. In this example, we would like the behavior to take control only when the touch sensor strikes an object, so the takeControl() method will be defined as follows:

```
public boolean takeControl() {  
    return Sensor.S2.readBooleanValue();  
}
```

For the action, we want the robot to back up and rotate when it strikes an object, so we will define the action() method as follows:

```
public void action() {  
    // Back up:  
    Motor.A.backward();  
    Motor.C.backward();  
    try{Thread.sleep(1000);}catch(Exception e) {}  
    // Rotate by causing one wheel to stop:  
    Motor.A.stop();  
    try{Thread.sleep(300);}catch(Exception e) {}  
    Motor.C.stop();  
}
```

Defining the suppress() method for this behavior is quite easy in this example. The action() method above is the sort of method that runs very quickly (1.3 seconds) and is

usually high priority. We can either stop it dead by stopping motor movement, or we could wait for it to complete the backing up maneuver. To keep things simple, let's just stop the motors from rotating:

```
public void suppress() {  
    Motor.A.stop();  
    Motor.C.stop();  
}
```

The complete listing for this behavior is as follows:

```
import lejos.subsumption.*;  
  
import lejos.nxt.*;  
  
public class HitWall implements Behavior {  
    public boolean takeControl() {  
        return Sensor.S2.readBooleanValue();  
    }  
  
    public void suppress() {  
        Motor.A.stop();  
        Motor.C.stop();  
    }  
  
    public void action() {  
        // Back up:  
        Motor.A.backward();  
        Motor.C.backward();  
        try{Thread.sleep(1000);}catch(Exception e) {}  
        // Rotate by causing only one wheel to stop:  
        Motor.A.stop();  
        try{Thread.sleep(300);}catch(Exception e) {}  
        Motor.C.stop();  
    }  
}
```

We now have our two behaviors defined, and it's a simple matter to make a class with a `main()` method to get things started. All we need to do is create an array of our Behavior objects, and instantiate and start the Arbitrator as shown in the following code listing:

```
import lejos.subsumption.*;  
  
public class BumperCar {  
    public static void main(String [] args) {  
        Behavior b1 = new DriveForward();  
        Behavior b2 = new HitWall();  
        Behavior [] bArray = {b1, b2};  
        Arbitrator arby = new Arbitrator(bArray);  
        arby.start();  
    }  
}
```

The above code is fairly easy to understand. The first two lines in the `main()` method create instances of our Behaviors. The third line places them into an array, with the lowest priority behavior taking the lowest array index. The fourth line creates the Arbitrator, and the fifth line starts the Arbitration process. When this program is started the robot will scurry forwards until it bangs into an object, then it will retreat, rotate, and continue with its forward movement until the power is shut off.

This seems like a lot of extra work for two simple behaviors, but now let's see how easy it is to insert a third behavior without altering any code in the other classes. This is the part that makes behavior control systems very appealing for robotics programming. Our third behavior could be just about anything. We'll have this new behavior monitor the battery level and play a tune when it dips below a certain level. Examine the completed Behavior:

```

import lejos.subsumption.*;
import lejos.nxt.*;

public class BatteryLow implements Behavior {
    private float LOW_LEVEL;

    private static final short [] note = {
        2349,115, 0,5, 1760,165, 0,35, 1760,28, 0,13, 1976,23,
        0,18, 1760,18, 0,23, 1568,15, 0,25, 1480,103, 0,18,
        1175,180, 0,20, 1760,18, 0,23, 1976,20, 0,20, 1760,15,
        0,25, 1568,15, 0,25, 2217,98, 0,23, 1760,88, 0,33, 1760,
        75, 0,5, 1760,20, 0,20, 1760,20, 0,20, 1976,18, 0,23,
        1760,18, 0,23, 2217,225, 0,15, 2217,218};

    public BatteryLow(float volts) {
        LOW_LEVEL = volts;
    }

    public boolean takeControl() {
        float voltLevel = (ROM.getBatteryPower() * 10 / 355);
        int displayNum = (int)(voltLevel * 100);
        LCD.setNumber(0x301f, displayNum, 0x3004);
        LCD.refresh();
        return voltLevel < LOW_LEVEL;
    }

    public void suppress() {
        // Nothing to suppress
    }

    public void action() {
        play();
        try{Thread.sleep(3000);}catch(Exception e) {}
        System.exit(0);
    }

    public static void play() {
        for(int i=0;i<note.length; i+=2) {
            final short w = note[i+1];
            Sound.playTone(note[i], w);
            try {
                Thread.sleep(w*10);
            } catch (InterruptedException e) {}
        }
    }
}

```

The complete tune is stored in the note array at line 6 and the method to play the notes is at line 30. This behavior will take control only if the current battery level is less the voltage specified in the constructor. The takeControl() method looks a little inflated, and that's because it also displays the battery charge to the LCD display. The action() and suppress() methods are comparatively easy. Action makes a bunch of noise, then exits the program as soon as it is called. Since this behavior stops the program, there is no need to create a suppress() method.

To insert this Behavior into our scheme is a trivial task. We simply alter the code of our main class as follows:

```
public class BumperCar {
    public static void main(String [] args) {
        Behavior b1 = new DriveForward();
        Behavior b2 = new BatteryLow(6.5f);
        Behavior b3 = new HitWall();
        Behavior [] bArray = {b1, b2, b3};
        Arbitrator arby = new Arbitrator(bArray);
        arby.start();
    }
}
```

Note: The voltage level of the NXT at rest is different from the voltage when in action. The voltage level at rest might be 7.8 V, but when motors are activated they naturally cause a drop in the voltage reading. Make sure the voltage threshold used in the BatteryLow constructor is low enough.

This example beautifully demonstrates the real benefit of behavior control coding. Inserting a new behavior, no matter what the rest of the code looks like, is simple. The reason for this is grounded in object oriented design; each behavior is a self contained, independent object.

**TIP:** When creating a behavior control system, it is best to program each behavior one at a time and test them individually. If you code all the behaviors and then upload them all at once to the NXT brick, there is a good chance a bug will exist somewhere in the behaviors, making it difficult to locate. By programming and testing them one at a time it makes it easier to identify where the problem was introduced.

Behavior coding is predominantly used for autonomous robots - robots that work independently, on their own free will. A robot arm controlled by a human would likely not use behavior programming, although it would be possible. For example, a robot arm with four joystick movements could have a behavior for each direction of movement. But as you may recall, behaviors are ordered with the highest order taking precedence over lower order behaviors. Who is to say that pushing left on the joystick would take precedence over pushing up? In other words, behavior control in anything other than autonomous robots is largely overkill.

## Advanced Behavior Coding

It would be nice if all behaviors were as simple as the examples given above, but in more complex coding there are some unexpected results that can sometimes be introduced. Threads, for example, can sometimes be difficult to halt from the `suppress()` method, which can lead to two different threads fighting over the same resources - often the same motor! Another problem that can potentially occur in multi-threaded programs is that events go undetected, such as touch sensor hits. These are a few of the pitfalls we will be examining in this section. Let's start by looking at what is generally the least complicated of the three Behavior methods to implement; the `takeControl()` method.

Note: The behavior control API used by leJOS NXJ is a modified version of the model proposed by Rodney Brooks. His model is all done at the lowest level possible - the motors. This prevents higher level classes from being used in behaviors. For example, the Navigator class accesses the motors of the NXT directly, so with the original behavior control model Navigator could not be used. Also, if both motors are moving forward, and a higher level behavior takes command it is not clear if all lower level motor movements should be stopped. What if the higher level behavior only uses one of the motors? Should the other keep moving forward? And will this lead to odd behavior? These are the problems the leJOS Behavior Control API tries to address.

### Coding Foolproof `takeControl()` Methods

It is important for `takeControl()` methods to be responsive in behavior control systems. When a bumper collides with an object, the robot must stop or reverse direction immediately; otherwise it will continue to move forward into the object. Sometimes when an event occurs, such as a touch sensor press, the program misses the event because the RCX is executing another thread. By the time it gets to the `takeControl()` method the sensor has been released and the program misses its opportunity to activate the proper behavior action. In this section we will learn how to make fool-proof `takeControl()` methods.

In the above example we used single indicators of whether or not to take control. For example, it took one reading from the Sensor class to check if the touch sensor was hit. The `takeControl()` method can also make a decision to take control based on a number of different values. It could initiate an action if it is facing east, the light reading is greater than 60, and the temperature is less than 20 degrees:

```
public boolean takeControl() {
    boolean pass = false;
    if(direction == EAST)
        if(Sensor.S1.readValue() > 60)
            if(Sensor.S2.readValue() < 20)
                pass = true;
    return pass;
}
```

Likewise, a different behavior could just as easily check on the same data, only react differently based on different values. For example, another Behavior could initiate a different action if the robot is facing west, the light reading is less than 60, and temperature is greater than 20 degrees. So a robot can initiate an unlimited number of responses with only a few sensors at its disposal. This leads to another point about implementing the `takeControl()` method.

With the Arbitrator cycling through all the `takeControl()` methods, there could be a significant delay in checking a condition, such as whether a touch sensor has been tapped. It's a feature of the imperfect world we live in that, when the robot strikes an object, the touch sensor may not remain pressed. It sometimes bounces off the object into a position where the bumper is no longer pressing on the touch sensor. You may have noticed in the example that it relies on checking touch very often. What if the touch sensor is momentarily activated, but the Arbitrator misses this fact? The solution is to use a `SensorListener`, and have it set a flag to indicate the event has occurred. Let's take the `HitWall` Behavior from the example above and modify it so it uses a `SensorListener`:



```

import lejos.subsumption.*;
import lejos.nxt.*;

public class HitWall implements Behavior, SensorListener {
    boolean hasCollided;

    // Constructor:
    public HitWall() {
        hasCollided = false;
        SensorPort.S2.addSensorListener(this);
    }

    public void stateChanged(Sensor bumper, int oldValue, int newValue) {
        if(bumper.readBooleanValue() == true)
            hasCollided = true;
    }

    public boolean takeControl() {
        if(hasCollided) {
            hasCollided = false; // reset value
            return true;
        } else
            return false;
    }

    public void suppress() {
        Motor.A.stop();
        Motor.C.stop();
    }

    public void action() {
        // Back up:
        Motor.A.backward();
        Motor.C.backward();
        try{Thread.sleep(1000);}catch(Exception e) {}
        // Rotate by causing only one wheel to stop:
        Motor.A.stop();
        try{Thread.sleep(300);}catch(Exception e) {}
        Motor.C.stop();
    }
}

```

The above code implements a `SensorPortListener`, and hence implements the `stateChanged()` method. It is important to add the sensor listener to `Sensor.S2`, as shown in line 10. Notice the `stateChanged()` method does not simply return the value of the bumper Sensor; rather, if the Sensor value is true then it changes the `hasCollided` variable to true. If, on the next pass, the sensor value is false then `hasCollided` will remain true until `takeControl()` has seen the `hasCollided` value. Once `takeControl()` sees there has been a collision, then `hasCollided` is reset back to false (line 20). With this new code, it should be impossible for the robot to miss any collisions with the bumper!

## Coding Solid `action()` and `suppress()` Methods

In order to code functional `action()` and `suppress()` pairs, it is necessary to understand how arbitration works. Arbitrator cycles through each of its Behaviors, checking the `takeControl()` method to see if the `action()` for the Behavior should be executed. It starts with the highest priority method and goes down to the lowest priority Behavior. As soon as it comes across a behavior that wants to take control, it executes `suppress()` for the

previous Behavior (assuming it is not a higher level thread), then runs the action() method for the current Behavior. As soon as the action() method returns, it then starts looping again, checking each behavior. If the takeControl() from the previous Behavior continues to say true, it does not run action() again. This is important; a single Behavior can not be executed twice in a row. If it could, it would constantly be suppressing itself. If Arbitrator moves on to another Behavior, when that behavior completes then it will call action on the lower level behavior again.

Note: If you would like to remove any mystery about what goes on in the Arbitrator class, take a look at the source code located in src/classes/lejos/subsumption/Arbitrator.java.

To program individual behaviors it is important to understand the fundamental differences between types of behaviors. Behavior actions come in two basic varieties:

- Discrete actions which finish quickly (e.g. back up and turn)
- Actions that start running and keep going an indefinite period until they are suppressed (e.g. driving forward, following a wall).

One final word of advice. Discrete actions execute once and return from the action() method call only when it has completed its behavior. These types of Behaviors generally do not need any code in the suppress() method because once the action is done there is nothing to suppress. The second type of action sometimes runs in a separate thread, although not always. For example, the Motor.A.forward() method call acts like a thread because the motor keeps turning after the method returns. In actuality, this is not a thread; the RCX just turns on an internal switch to activate the motor. An example of a true thread would be complex behavior, such as wall following. The action() method could start a thread to begin following a wall until the suppress() method is called. Be careful of never ending loops! If one were to occur within the action() method then the program would become stuck.

## Summary

So why use the Behavior API? The best reason is because in programming we strive to create the simplest, most powerful solution possible, even if it takes slightly more time. The importance of reusable, maintainable code has been demonstrated repeatedly in the workplace, especially on projects involving more than one person. If you leave your code and come back to it several months later, the things that looked so obvious suddenly don't anymore. With Behavior control, you can add and remove behaviors without even looking at the rest of the code, even if there are 10 or more behaviors in the program. Another big plus of behavior control is programmers can exchange Behaviors with each other easily, which fosters code reusability. Hundreds of interesting, generic behaviors could be uploaded to websites, and you could simply pick the behaviors you want to add to your robot (assuming your robot is the correct type of robot). This reusability of code can be taken forward even more by using standard leJOS NXJ classes such as the Navigation API.

# **Third- Party Sensors**

## Third Party sensors and other devices

leJOS NXJ supports many third party sensors. The two main vendors of third party sensors are Mindsensors and HiTechnic.

Most of the third party sensors and I2C sensors extend the I2CSensor class but there are also Analog/Digital sensors such as the iTechnic Gyro sensor and the IR Seeker.

There are also other I2C devices supplied by the third parties, that are not sensors, but are multiplexers or adapters.

The RCX Motor Multiplexer for Mindsensors is an example of a multiplexer. It allows up to 4 RCX motors to be connected to a NXT sensor port and to be independently controlled.

The Mindstorms NRLink-Nx infra-red communications adapter is an example of an adapter. It allows two-way communication between the NXT and RCXs. It also allows control of Power Function motors.

### I2CSensor

The I2CSensor class implements the basic methods for accessing I2C sensors including getData and sendData.

It also includes methods that are implemented by all the I2C sensors, including getVersion, getProductID and getSensorType.

The method signatures are:

- public int getData(int register, byte [] buf, int len)
- public int sendData(int register, byte [] buf, int len)
- public int sendData(int register, byte value)
- public String getVersion()
- public String getProductID()
- public String getSensorType()

leJOS NXJ uses 7-bit I2C addresses, whereas I2C device specification often give them as 8-bit addresses with the least significant bit set to zero. Most I2C sensors use a default address of 0x01, which such specifications give as 0x02. Most I2C devices allow the address to be changed.

The setAddress method can be used to set the address used to talk to the I2C device – it defaults to 0x01.

Individual I2C devices have registers that can be read and written and registers that can be used to execute commands. Each I2C device has a class that extends I2CSensor and has methods to access the registers and execute the commands specific to that sensor.

The I2CSensor class can be used to implement an I2C device explorer that reports what devices are connected to which sensor sing which address – see the I2CDevices sample. This is possible as all the NXT I2C sensors and other devices support the getVersion, getProductID and getSensorType methods.

## **Compass Sensor**

There are compass sensors sold both by HiTechnic and by Mindsensors. A single leJOS class – CompassSensor supports both of these.

The main method of CompassSensor is:

- public float getDegrees()
- public float getDegreesCartesian()

## **Color Sensor**

The HiTechnic color sensor is supported by the ColorSensor class.

## **Acceleration (Tilt) Sensor**

The range of HiTechnic and Mindsensors acceleration sensors are supported by the TiltSensor class.

## **Gyro Sensor**

The HiTechnic Gyro sensor is supported by the GyroSensor class.

## **IRSeeker**

The HiTechnic IR Seeker sensor is supported by the IRSeeker class.

## **RCX Motor multiplexer**

The Mindsensors MTRMX-Nx sensor is supported by the RCXMotorMultiplexer class.

## **Optical Distance Sensor**

The Mindsensors range of Dist-Nx infra-red distance sensors are supported by the OpticalDistanceSensor class.

# **Advanced Communications**

# **External Bluetooth Devices**



## External Bluetooth devices

The NXT can connect to external Bluetooth devices that implement the Serial Port Profile (SPP).

Such devices can be searched for on the Bluetooth menu and added to the known devices.

### GPS

leJOS supports external Bluetooth GPS receivers that support the NMEA protocol via the GPS and NMEASentence classes. NMEASentence is a utility class used by GPS, and is not directly accessed.

One such device that has been tested with leJOS NXJ is the Holux M-1200.

Most such devices have a PIN that is required to connect to them, but it may have a default value such as “0000”.

To connect to a GPS device, you do:

```
final byte[] pin = {(byte) '0', (byte) '0', (byte) '0', (byte) '0'};

BTRemoteDevice btGPS = Bluetooth.getKnownDevice(name);

if (btrd == null) {
    LCD.drawString("No such device", 0, 0);
    Thread.sleep(2000);
    System.exit(1);
}

btGPS = Bluetooth.connect(btrd.getDeviceAddr(), pin);

if(btGPS == null)
    LCD.drawString("No Connection", 0, 1);
    Thread.sleep(2000);
    System.exit(1);
} else {
    LCD.drawString("Connected!", 0, 1);
}

GPS gps = null;
InputStream in;

try {
    in = btGPS.openInputStream();
    gps = new GPS(in);
    LCD.drawString("GPS Online", 0, 6);
} catch (Exception e) {
    LCD.drawString("GPS Connect Fail", 0, 6);
    Thread.sleep(2000);
    System.exit(1);
}
```

As you see from this example, the GPS constructor takes the input stream from the Bluetooth connection as a parameter:

- `public GPS(InputStream in)`

It uses the `NMEASentence` class to process messages (known as sentences) from the Bluetooth device. Currently only the `$GPGGA` sentence that gives the current latitude, longitude and altitude, is processed.

To read the current values of latitude, longitude and altitude, you use the methods:

- `public float getLatitude()`
- `public float getLongitude()`
- `public float getAltitude()`

You can also get the time stamp corresponding to these values by:

`public int getTime()`

# **Controlling a remote NXT**

## Controlling a remote NXT

The RemoteNXT class allows one NXT running leJOS to control another, remote NXT, running leJOS or the standard LEGO firmware. It uses the LEGO Communications Protocol (LCP) over Bluetooth to control the remote NXT.

Currently, the class is limited and I2C and RCX sensors are not supported, and the motors must be used in a simple way as the regulation thread is not used.

To access a remote NXT, you use the constructor:

- `public RemoteNXT(String name)` throws `IOException`

Example:

```
try {
    LCD.drawString("Connecting...", 0, 0);
    nxt = new RemoteNXT("NXT");
    LCD.clear();
    LCD.drawString("Connected", 0, 0);
    Thread.sleep(2000);
} catch (IOException ioe) {
    LCD.clear();
    LCD.drawString("Conn Failed", 0, 0);
    Thread.sleep(2000);
    System.exit(1);
}
```

The name of the remote NXT must have already been added to the known devices of the initiating NXT by do a Bluetooth search followed by “Add” from the leJOS NXJ Bluetooth menu.

The constructor opens the connection and creates instances of the remote motor and sensor ports.

It is then possible to get access to information about the remote NXT by using the methods:

- `public String getBrickName()`
- `public String getBluetoothAddress()`
- `public int getFlashMemory()`
- `public String getFirmwareVersion()`
- `public String getProtocolVersion()`

Example:

```
LCD.drawString(nxt.getBrickName(), 0, 6);
LCD.drawString(nxt.getFirmwareVersion(), 0, 7);
LCD.drawString(nxt.getProtocolVersion(), 4, 7);
LCD.drawInt(nxt.getFlashMemory(), 6, 8, 7);
```

There are also methods that act on the remote NXT:

- `public byte deleteFlashMemory()`

A remote Battery object is created that can be used to get the voltage of the remote battery using the normal Battery methods.

Example:

```
LCD.drawString( "Battery: " + nxt.Battery.getVoltageMilliVolt() 0,4);
```

Objects are

also created for the sensor ports of the remote NXT. These are accessed as S1, S2, S3 and S4.

Local sensor objects can then be created using these ports and use exactly as if they were connected to a local sensor port.

Example:

```
LightSensor light = new LightSensor(nxt.S1);  
LCD.drawString("Light: " + light.readValue(),0,5);
```

Motor objects are created for the remote motors. They are named A, B and C.

These can be used in the normal way, e.g:

```
nxt.A.setSpeed(360);  
nxt.A.forward();  
nxt.A.stop();  
nxt.A.backward();
```

# **Infra-red Communications**

## **IR Communications**

There are two third-party adapters available for communication by infra-red with the RCX and other devices such as the Power Function motors: the Mindsensors NRLink-Nx and the HiTechnic IRLink. Currently only the NRLink-Nx is supported by leJOS NXJ.

# **Communicating with the RCX**



## RCX Communications

### RCXLink

IR communication is supported by the RCXLink class. The constructor is:

- `public RCXLink(I2CPort port)`

For example:

```
RCXLink link = new RCXLink(SensorPort.S1);
```

The NRLink-Nx supports a set of macros in ROM and EEPROM that can be used to send messages to the RCX using the LEGO RCX IR protocol. The EEPROM macros can be overwritten allowing the user to define their own macros.

Macros can be run by:

- `public void runMacro(int addr)`

There are convenience methods for running the ROM macros:

- `public void beep()`
- `public void runProgram(int programNumber)`
- `public void forwardStep(int id)`
- `public void backwardStep(int id)`
- `public void setRCXRangeShort()`
- `public void setRCXRangeLong()`
- `public void powerOff()`
- `public void stopAllPrograms()`

A new macro in the EEPROM address range can be defined by:

- `public void defineMacro(int addr, byte[] macro)`

There is a convenience method to define and run a macro:

### ***PF Motors***

Power Function Motors are not yet supported by leJOS NXJ.

# Advanced User Interfaces

# Advanced User Interfaces

## ***TextMenu***

For applications that require input from the user via the NXT buttons, the `TextMenu` class can be used. It displays a text menu on the LCD and allows the user to use the NXT button to move through menu items, select a menu item or cancel a menu.

`TextMenu` is used by the leJOS NXJ start-up menu, and also by some sample programs such as `View`.

A text menu is constructed by one of:

- `public TextMenu( String[] items)`
- `public TextMenu( String[] items, int topRow)`
- `public TextMenu(String[] items, int topRow, String title)`

The top row defaults to zero (the top row of the LCD). If a title is specified, it is displayed on the top row and the menu moves down one line. A title can be reset by:

- `public void setTitle(String title)`

The items array specifies the names of the menu items. These can be reset by:

- `public void setItems(String[] items)`

The menu is displayed from `topRow` downwards to the bottom line of the screen. If there are more items than will fit on the screen, the item names will scroll. Menu item names can be up to 15 characters. If the string is longer, the first 15 characters will be displayed.

The current menu item is indicated by a “>” at the start of the line.

To display the menu and wait for the user to select an item, you use one of the `select()` methods:

- `public int select()`
- `public int select(int selectedIndex)`

`selectedIndex` specifies the item in the items array that is selected as the current item. It defaults to zero – the first item in the array.

`select()` suspends the current thread waiting for the user to select an item. It returns the index of the item that the user selects. The user scrolls through the menu with the `LEFT` and `RIGHT` keys and selects an item using `ENTER`. If `ESCAPE` is pressed, `select()` returns -1. This is typically used to cancel the menu and return to a previous one. The menu can also be quit by another thread calling:

- `public void quit()`

This causes `select()` to return -2.

`TextMenu` allows you to build a menu system with submenus. See `src/startup/StartUpText.java` (the leJOS NXJ start-up menu) or the View sample for examples of this. You can display as many lines of header information above menus as you require.

## ***Graphics***

## Using RCX sensors and motors with the conversion cables

RCX motors and sensors can be connected to the NXT using the conversion cables that can be purchased from LEGO.

### ***RCX Motors***

The RCX motors do not have inbuilt tachometer and so cannot support the advanced functions of the NXT motors such as the rotate and rotateTo methods and the speed regulation.

A simpler class is used to support the RCX motors. It has similar methods to the Motor class in the RCX version of leJOS.

To use an RCX motor you create an instance of the RCXMotor class using the constructor:

- `public RCXMotor(BasicMotorPort port)`

Example:

```
RCXMotor rcxMotor = new RCXMotor(MotorPort.A);
```

The methods supported by RCX motors are:

- `public void setPower(int power)`
- `public int getPower()`
- `public void forward()`
- `public boolean isForward()`
- `public void backward()`
- `public boolean isBackward()`
- `public void reverseDirection()`
- `public boolean isMoving()`
- `public void flt()`
- `public boolean isFloating()`
- `public void stop()`
- `public boolean isStopped()`
- `public int getMode()`

# **Using RCX Sensors**

## **RCX Sensors**

RCX sensors, other than the touch sensor, are active sensors that have voltage applied for all but the short period every three milliseconds when the measurement is taken.

### **RCX Light Sensor**

The RCX light sensor is supported by the `RCXLightSensor` class.

The constructor is:

- `public RCXLightSensor(LegacySensorPort port)`

For example:

```
RCXLightSensor light = new RCXLightSensor(SensorPort.S1);
```

The RCX light sensor is automatically activated, so current is applied to it and the LED comes on. It can be passivated and activated explicitly.

The methods are:

- `public int readValue()`
- `public void activate()`
- `public void passivate()`

### **RCX Touch Sensor**

As the RCX touch sensor is a passive sensor similar to the NXT version, it is supported by the standard `TouchSensor` class.

### **RCX Rotation Sensor**

The RCX rotation sensor is not currently supported by leJOS NXJ.

### **RCX Temperature Sensor**

The RCX rotation sensor is not currently supported by leJOS NXJ.



# **Using the development version of leJOS**

## **Using the Subversion version of leJOS**

# Reference

# **leJOS menu system**





## ***The leJOS NXJ Menu System***

### **Main menu**

When leJOS NXJ starts, it displays the leJOS NXJ logo for 3 seconds and then displays the main menu:

```
      MYNXT      BT
>Run Default
  Files
  Bluetooth
  System
```

The top lines shows the free RAM on the left, and the battery voltage on the right.

It then shows a menu implemented by the TextMenu class. You can scroll through the menu using the  and  keys, and select a menu item using the  key. You can quit a menu and return to the previous one by pressing the  key. On the main menu, this shuts down the NXT.

### ***Files menu***

When you select the “Files” menu item, the files menu is displayed:

```
      MYNXT      BT
>View.nxj
  SoundScope.nxj
  LCDTest.nxj
```

When you select a particular file, the file submenu is displayed:

```
MYNXT      BT
View.nxj
>Execute program
Delete file
```

The other menus are the Bluetooth and the System menus.

## ***System menu***

The System menu looks like:

```
MYNXT      BT
System
Free flash 96000
Battery    7.6
Free ram   47512
>Format
```

The format menu item wipes out all the flash memory, deleting all files.

## ***Bluetooth menu***

```
MYNXT      BT
Bluetooth
Status on vis
>Power off
Devices
Search
Visibility
```

The status line shows if power is on and where the device is visible to Bluetooth searches.

The Power on/power off menu item allows Bluetooth power to be turned on and off. When power is off, battery power is saved, but you will not be able to connect to the device, or make an outward connection to another device, over Bluetooth. You will also not be able to change the friendly name of the device.

The Visibility menu item switches visibility of the devices to Bluetooth inquiries on and off. When it is on the status line displays “vis” and when it is off, “invis”.

## Devices

The Devices menu item lists the devices currently in the Known Devices list for you NXT.

```
Devices
>NXT1
>NXT2
>HOLUX_M-1200
```

Selecting a specific device will show details of it:

```
Devices
NXT1
001653007848
  0    0    0    4
>Remove
```

The first line give the name, the second the address, and the third the status of the four possible Bluetooth connections.

The “Remove” menu item allows the device to be removed from the Known device list.

## Search

Selecting “Search” from the Bluetooth menu, starts a Bluetooth inquiry. The display is cleared and the first line displays “Searching...”. This takes about 10 seconds.

When the search is finished a menu of all the Bluetooth devices found, is displayed:

```
Found
>NXT1
>HOLUX_M-1200
>MyPhone
```

All discoverable devices that support an SPP profile are displayed. This includes other NXT, GPS devices, mobile phones etc.

Selecting one of the devices gives the following display:

```
Found
NXT1
001653007848

>Add
```

Selecting Add, adds the device to the list of know devices. This makes connecting to the devices from leJOS programs much easier.



# **PC Command- line Tools**

## Using the leJOS NXJ command line tools

The tools for compiling, linking and uploading leJOS NXJ programs are:

- nxjc
- nxjlink
- nxjupload
- nxj

### **nxjc – compile a leJOS NXJ program**

Compiles one or more java files.

Usage: nxjc <java-files>

Example: nxjc View.java

nxjc calls javac with parameters:

- -source 1.3
- -target 1.1
- -bootclasspath <bootclasspath>
- <java-files>

-bootclasspath is set because leJOS does not use the standard java.lang classes but has its own versions in classes.jar.

-source 1.3 and -target 1.1 are set because the leJOS VM does not support the latest java source and class file versions.

### **nxjlink – link a leJOS NXJ program**

Calls the leJOS linker.

Usage: nxjlink [-v|--verbose] [-a|--all] class -o <binary>

Example: nxjlink -v Tune -o Tune.nxj

Links the specified class with any classes that it references in the current directory and with the standard leJOS classes from classes.jar to produce a binary NXJ program that can be uploaded and run.

The -v or -verbose flag causes a list of class names and method signatures included in the binary to be output to the standard output.

The linker removes methods that are not used. Specify -a or -all to include all methods whether they are used or not.

Use the `-h` or `--help` flag to print out the options.

## **nxjupload – upload a leJOS NXJ program**

Usage: `nxjupload [-b|--bluetooth] [-u|--usb] [-d|--address address] [-n|--name name] [-r|--run] <binary>`

Example `nxjupload Tune.nxj`

Uploads the binary (.nxj) file. By default usb is tried first and then Bluetooth. If the `--bluetooth` flag is set, only Bluetooth is tried. If `--usb` is set, only USB is tried.

When Bluetooth is used, a search for Bluetooth devices is done, unless the `--address` flag is set, when a device with the given address is connected to.

The `--name` parameter limits the search for a NXT with the given name. If this is not specified, `nxjupload` tries to connect to each NXT that it finds and will upload to the first NXT that is successfully connects to.

If the `--run` parameter is specified, the program is run after it has been uploaded.

## **nxj – link and upload and optional run a leJOS NXJ program**

Usage: `nxj [options] <class>`

Example: `nxj -r Tune`

The `nxj` command links and uploads a leJOS NXJ program. It is the equivalent of `nxjlink` followed by `nxjupload`.

# **PC GUI Tools**

# The NXJ Browser

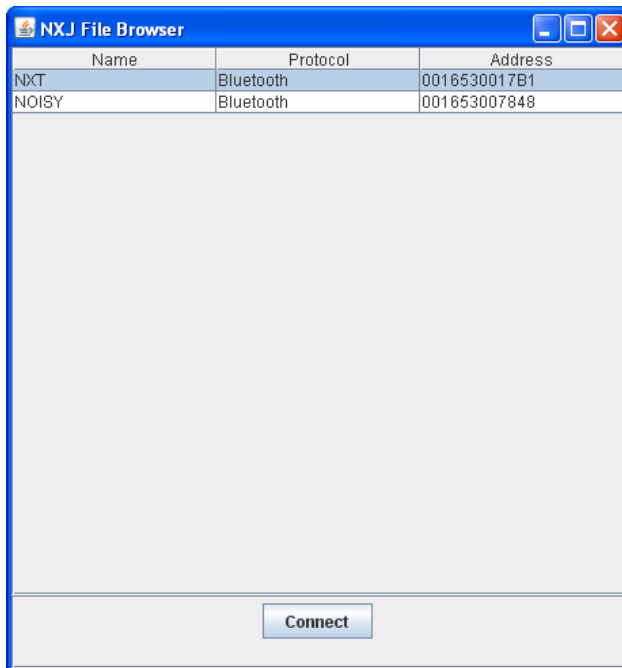
leJOS NXJ includes a PC-based file browser for viewing and manipulating the files on the NXT.

It is started by the nxjbrowse script. If you are using an IDE, you can set NXJ Browse up as an external tool.

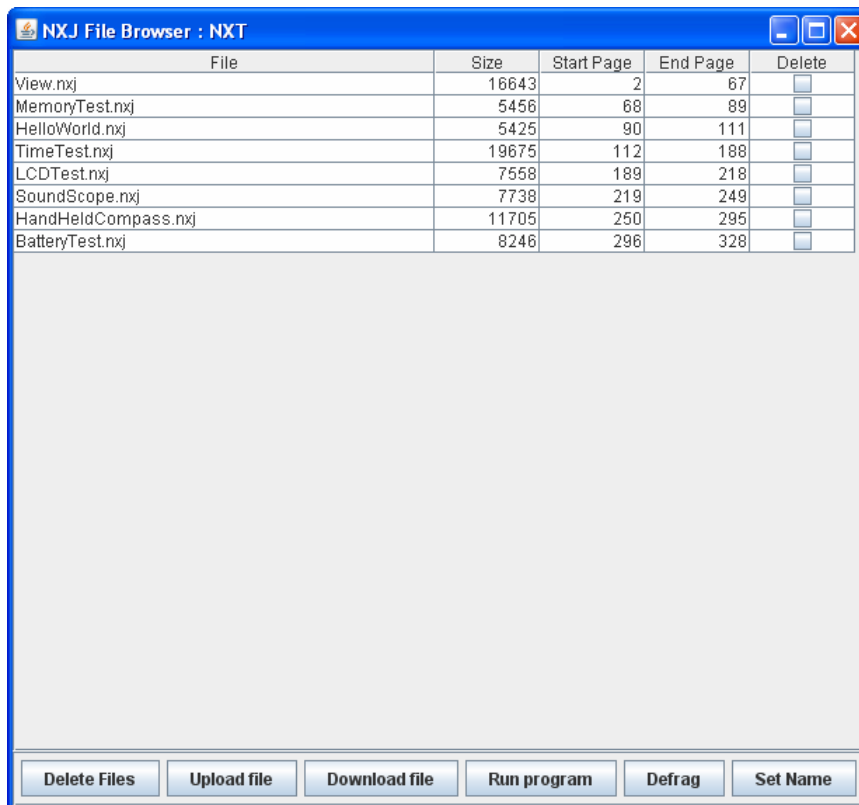
The NXJ Browser need the NXT it is connecting to, to be turned on and to be running the leJOS start-up menu. Although it should work while a user program is running, if the program starts the LCPBTResponder thread, this is not recommended.

The NXJ Browser first looks for a NXT connected by USB, and if it finds one shows just that in the selection window.

If it cannot find a USB-connected NXT, it does a Bluetooth inquiry to find any NXTs in range and shows them in the selection window. You then select the one you wish to connect to, and press connect.



After pressing Connect, the main browser screen is shown:



The NXJ browser shows you all the files on your NXT, together with their size in bytes and their location in the file system (start page, end page). See “Understanding the leJOS NXJ File System” to understand the meaning of the start and end pages.

The friendly name of the NXT you are connected to is shown in the title bar of the Windows – the name is “NXT” in this case.

From this screen, you can:

- Delete programs
- Upload programs and other files to the NXT
- Download files from the NXT to the OC
- Run programs
- De-fragment the file system
- Set the friendly name of the NXT

## Deleting programs

To delete files from the NXT, click the **Delete** tick boxes and press **Delete Files**. The next will make a chirping sound for each file deleted, and the NXJ Browser display will be updated. If the NXT is on the Files menu, its LCD will also be updated. An automatic defrag will occur when files are deleted, so there should be no gaps in the file system.

## Uploading files

When you click ***Upload File***, an Open file dialog box appears. Browse for the file you want to upload, select it, and click **Open**. NXJ Browse is not normally used for uploading program files as this is better done using the ***nxj*** or ***nxjupload*** tool or the leJOS Eclipse plugin. However, it is useful for uploading WAV files and data files for programs.

## Downloading files

Files can be downloaded from the NXT to the PC by selecting the NXT file and pressing “Download”. This opens a dialog that allows you to select the folder and filename to download the file to.

## Running programs

A program on the NXT can be run by selecting it and pressing “Run”. This shuts down NXJ Browser.

## De-fragmenting the file system

The NXT file system can be defragmented (removing any gaps between files) by pressing “Defrag”. In the latest versions of leJOS the file system does not normally need to be defragmented.

## Setting the friendly name of the NXT

The Bluetooth friendly name of the NXT can be set by pressing “Set Name”, filling in the name (up to 16 characters) and pressing OK. Note that the Bluetooth power must be on in order to set the friendly name, even if the connection to the NXT is by USB.

## Remote Monitoring and Tracing

You can monitor a leJOS program over a Bluetooth connection while it is running by starting the LCPBTResponder thread and running the NXJ Monitor tool on the PC.

You an example of this, see the MonitorTest sample.

NXJ Monitor lets you see the values of sensors, and the values of motor tachometers, in near real-time, while your program executes.

It also lets you program displaying tracing messages to indicate what it is doing. These are displayed on the NXJ Monitor screen.

The program being monitored should include the following code in its start-up sequence:

```
LCPBTResponder lcpThread = new LCPBTResponder();  
lcpThread.setDaemon(true);  
lcpThread.start();
```

On the PC, the NXJ Monitor tool should be run. NXJ Monitor lists the NXTs available over Bluetooth in the same way as the NXJ Browser tool, and lets you choose which one you wish to connect to.

When a connection has been made, the following window is displayed:



## ***Socket Proxy***

A program on the PC can use a Bluetooth connection to talk to a program on the NXT. However a Bluetooth connection only works over 10s of metres. If you want to communicate with a program that is not running on a local PC but which may be running on a remote computer on the internet or a local area network, then a TCP/IP socket connection is more useful. Socket connections can be used for controlling robots at a distance – telerobotics.

The problem is that the NXT does not support TCP/IP connections. The way to get round this is to run a proxy on the local PC that implements Socket and ServerSocket connections. `lejos.pc.tools.SocketProxy` is such a proxy.

SocketProxy can be used in two ways: it allows connection as a client to a server running on the internet, and it runs as a server allowing clients on the internet to connect to it.

### ***Running as a client***

# Data Logging

***Datalogger***

Data Viewer

# **The Sample programs**

# **The leJOS NXJ sample programs**

## ***Building the samples***

Notes on individual sample programs

# Using other IDEs

## **Other Integrated Development Environments**

*Netbeans*

**BlueJ**

# **Understanding leJOS NXJ**

## The leJOS NXJ version of Java

This section teaches you about the specific version of leJOS supported on the NXT.

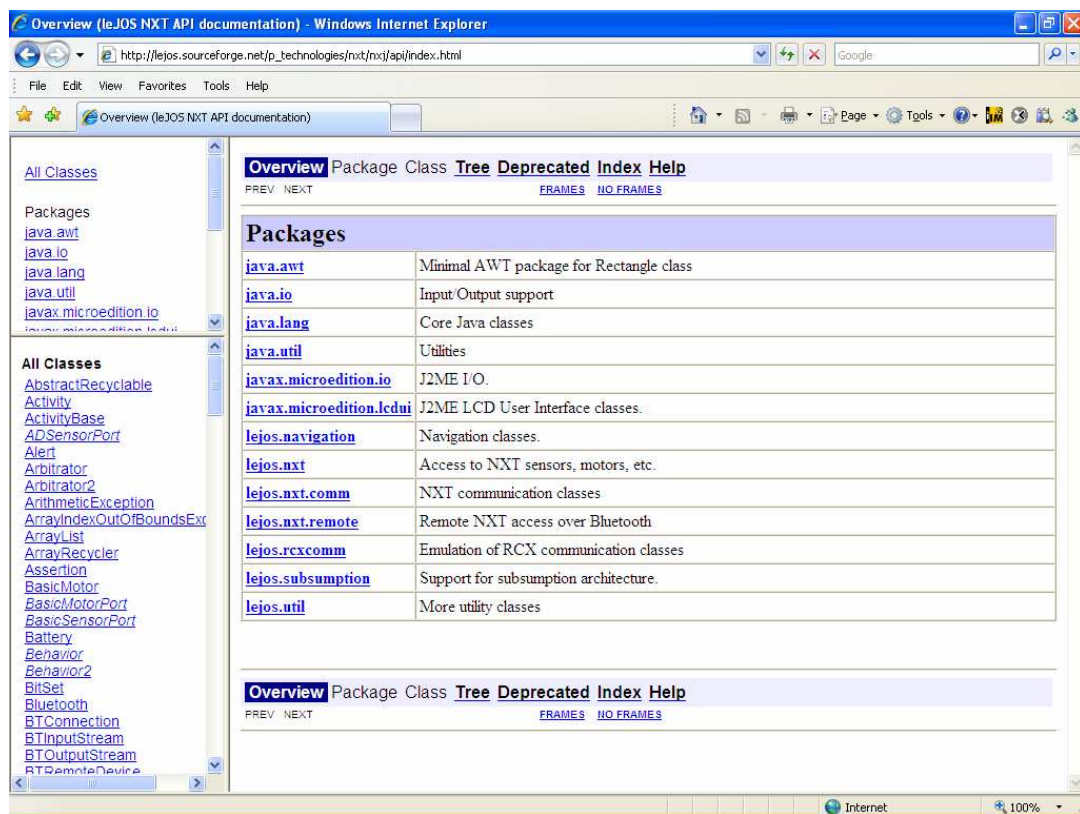
leJOS NXJ does not support a completely standard version of Java although it is getting close to it all the time. The Java VM in the leJOS NXJ firmware has a very small footprint – the whole firmware is less than 32kb – and this means that there are some restrictions when compared to a full Java implementation.

Some of these restrictions are:-

1. The latest source version of Java is not supported – you are restricted to Java version 1.3 source.
2. The Class class is not supported: there is no dynamic loading of classes and no reflection API.
3. The Runnable interface is not supported.
4. The java.lang API is not fully implemented. The most useful classes and methods are there, but a lot of the lesser used classes and methods are not supported.
5. Apart from java.lang, the only standard java packages that are implemented are java.math, java.io, and java.util. These are all cut down versions.
6. The only class from java.awt that is implemented is Rectangle, and this is a cut down version.
7. The graphics interface for the LCD screen is based on the javax.microedition.lcdui package, but this is only partially implemented. There is enough there to produce graphical user interfaces with forms, lists, gauges, images, text boxes, alerts, etc.
8. The Bluetooth API uses the StreamConnection interface from javax.microedition.io.
9. No other javax packages are implemented.
10. There are some restrictions in leJOS around the use of initializers, checking interfaces and use of monitors – see the README.html file for details.

The javadocs for the leJOS NXJ API is available online at [http://lejos.sourceforge.net/p\\_technologies/nxt/nxj/api/index.html](http://lejos.sourceforge.net/p_technologies/nxt/nxj/api/index.html) and is part of the leJOS distribution in docs/apidocs:





## Java Utilities

The java.util classes supported by leJOS are:

- ArrayList
- BitSet
- Hashtable
- Queue
- Random
- Stack
- Vector

# Understanding motors and motor ports

## Motor classes

BasicMotor

RCXMotor

Motor

## MotorPort

To use the NXT motors it is not necessary to explicitly use the MotorPort class: you can just use the variables Motor.A, Motor.B and Motor.C.

However it is useful to understand how motor ports work as they are used by:

- The Motor class
- The RCXMotor class
- The RemoteNXT class
- The RCXMotorMultiplexer class
- Remote RCX motors accessed via the RCXLink class

There is a hierarchy of interfaces defined for motor ports:

- BasicMotorPort
- Tachometer
- TachoMotorPort

All motor ports support the BasicMotorPort interface which allows control of a motors power and mode (forward, backward, stop, float).

Port that supports this include:

- NXT ports connected to NXT motors
- NXT ports connected via the RCX conversion cable to RCX motors
- Ports on the RCXMotorMultiplexer adapter
- Ports on remote NXTs accessed via the RemoteNXT class
- Ports on remote RCXs accessed via the RCXLink class

The implementations of BasicMotorPort include:

- MotorPort
- RemoteMotorPort
- RCXPlexedMotorPort
- RCXRemoteMotorPort

The tachometers that are built in to the NXT motors support the Tachometer interface.

NXT motor ports support the TachoMotorPort interface which includes the BasicMotorPort and Tachometer interfaces.

Implementations of TachoMotorPort include:

- MotorPort
- RemoteMotorPort

All this sounds rather complicated, but is simple to use:

- For NXT motors, you use Motor.A, Motor.B and Motor.C
- For RCX motors connected by conversion cable you use RCXMotor(MotorPort.A), RCXMotor(MotorPort.B) or RCXMotor(MotorPort.C)
- For NXT motors on a remote NXT, you use remoteNXT.Motor.A, remoteNXT.Motor.B or remoteNXT.Motor.C where remoteNXT is an instance of RemoteNXT.
- For RCX motors connected by the RCX Motor Multiplexer, you use rcxMotorMultiplexer.A, rcxMotorMultiplexer.B, rcxMotorMultiPlexer.C or rcxMotorMultiplexer.D, where rcxMotorMultiplexer is an instance of RCXMotorMultiPlexer.
- For RCX motors connected t remote RCXs via the RCXLink class you use rcxLink.A, rcxLink.B or rcxLink.C where rcxLink is an instance of the RCXLink class.

# Understanding sensors and sensor ports

## Sensor Ports

NXT sensor ports support three different types of sensor:

- NXT Analog/Digital Sensors
- I2C Sensors
- Legacy RCX sensors

All sensors ports support a basic interface that allows the type and mode of a sensor to set. The types of sensors are:

```
public static final int TYPE_NO_SENSOR = 0x00;
public static final int TYPE_SWITCH = 0x01;
public static final int TYPE_TEMPERATURE = 0x02;
public static final int TYPE_REFLECTION = 0x03;
public static final int TYPE_ANGLE = 0x04;
public static final int TYPE_LIGHT_ACTIVE = 0x05;
public static final int TYPE_LIGHT_INACTIVE = 0x06;
public static final int TYPE_SOUND_DB = 0x07;
public static final int TYPE_SOUND_DBA = 0x08;
public static final int TYPE_CUSTOM = 0x09;
public static final int TYPE_LOWSPEED = 0x0A;
public static final int TYPE_LOWSPEED_9V = 0x0B;
```

and the modes are:

```
public static final int MODE_RAW = 0x00;
public static final int MODE_BOOLEAN = 0x20;
public static final int MODE_TRANSITIONCNT = 0x40;
public static final int MODE_PERIODCOUNTER = 0x60;
public static final int MODE_PCTFULLSCALE = 0x80;
public static final int MODE_CELSIUS = 0xA0;
public static final int MODE_FARENHEIT = 0xC0;
public static final int MODE_ANGLESTEP = 0xE0;
```

Most of the time, with leJOS NXJ, these types and modes do not need to be set explicitly as it is done by the constructor for the sensor class being used, e.g. TouchSensor, LightSensor and UltrasonicSensor.

These type and mode constants are defined by the interface SensorConstants.

To allow these types and modes to be set, all sensor ports implement the interface BasicSensorPort which extends the interface SensorConstants.

It implements the methods:

- `public int getMode();`
- `public int getType();`

- `public void setMode(int mode);`
- `public void setType(int type);`
- `public void setTypeAndMode(int type, int mode);`

Corresponding to each of the different types of sensor, there is a corresponding interface:

- `ADSensorPort` which extends `BasicSensorPort`
- `LegacySensorPort` extends `ADSensorPort`
- `I2CPort` extends `BasicSensorPort`

The implementation of the NXT sensor port – `SensorPort` – supports all these interfaces. The reason for separating out the different interfaces is that there are other implementations of sensor ports that only support a subset of these interfaces, and different types of sensors only require particular interfaces to be implemented:

- I2C Sensors just require `I2CPort`
- NXT Analog/Digital sensors just require `ADSensorPort`
- RCX sensors such as the RCX Light sensor require `LegacySensorPort`

Port splitters like the Mindsensors Split-Nx only support I2C sensors and thus, effectively, only support the `I2CPort` interface.

There are other implementations that only support the other interfaces. For example the current implementation of remote sensor ports – `RemoteSensorPort` – currently only supports the `ADSensorPort` interface.

The classes for RCX Sensors multiplexers – such as the forthcoming Mindsensors version – will only support the `LegacySensorPort` interface.

## Sensors

Each sensor supported by leJOS NXJ has a specific class that is used to access the sensor. Each of these sensor classes has, as a parameter, a sensor port that supports the required interface. Any sensor port class that implements the interface can be specified as the parameter. As the `SensorPort` class supports all the interfaces, if the sensor being accessed is directly connected to the NXT, the parameter should be one of `SensorPort.S1`, `SensorPort.S2`, `SensorPort.S3` or `SensorPort.S4`.

If a port splitter is used the parameter again should be one of `SensorPort.S1`, `SensorPort.S2`, `SensorPort.S3` or `SensorPort.S4`. This specifies the port that the splitter is connected to. If multiple sensors are connected to the splitter they must each have different I2C addresses. Most I2C sensors can have their address changed – see the manufacturers instructions. To specify the address that a sensor uses, if it is not the default, the `setAddress` method of `I2CSensor` should be used.

The sensor ports supported by leJOS NXJ together with the class that supports them and the type of sensor port they require is given in the following table:

Sensor	Class	Sensor Port Interface
LEGO NXT Touch Sensor	TouchSensor	ADSensorPort
LEGO NXT Light Sensor	LightSensor	ADSensorPort
LEGO NXT Sound Sensor	SoundSensor	ADSensorPort
LEGO NXT Ultrasonic Sensor	UltrasonicSensor	I2CPort
RCX Touch Sensor	TouchSensor	ADSensorPort
RCX Light sensor	RCXLightSensor	LegacySensorPort
HiTechnic Compass Sensor	CompassSensor	I2CPort
HiTechnic Color Sensor	ColorSensor	I2CPort
HiTechnic Acceleration Sensor	TiltSensor	I2CPort
HiTechnic Gyro Sensor	GyroSensor	ADSensorPort
HiTechic IR Seeker	IRSeeker	I2CPort
Mindsensors Compass Sensor	CompassSensor	I2CPort
Mindsensors NXTCam	NXTCam	I2CPort
Mindsensors Acceleration sensors	TiltSensor	I2CPort
Mindsensors Dist-Nx sensors	OpticalDistanceSensor	I2CPort
Mindsensors NRLink-Nx	RCXLink	I2CPort
Mindsensors RCX Motor multiplexer	RCXMotorMultiplexer	I2CPort

## Understanding the leJOS File System

leJOS NXJ implements a file system using flash memory. The flash memory is read and written in 256-byte pages. The first two pages hold the file table (directory) and the rest of the pages hold user files. Files are held as a contiguous set of bytes – i.e they use a single range of page numbers with no gaps. This allows a file to be addressed as a region of memory.

### Flash

The **flash** class has methods to read and write 256-byte pages of flash memory. It should not be used by user programs.

### File

The File class has static methods that manipulate the file system as a whole and instance methods that give access to specify files.

Static methods:

### FileInputStream

### FileOutputStream

## Understanding LCP

LEGO defines a protocol called the LEGO MINDSTORMS NXT Communications Protocol (LCP) which is used to send commands to the standard LEGO firmware. The specification is available at <http://mindstorms.lego.com/Overview/NXTreme.aspx> in the Bluetooth Development Kit. The commands are separated into direct commands and system commands. The direct commands are described in a separate document: LEGO MINDSTORMS NXT : Direct Commands.

Direct commands are those that are designed for user programs and tools to use to control robots. The system commands are designed for tools that upload and download files and do other administrative tasks.

leJOS NXJ emulates many of the direct and systems commands so that many tools that work with the standard LEGO firmware also work with leJOS.

Many of the leJOS NXJ tools including nxj, nxjupload, nxjbrowse and nxjmonitor use LCP. leJOS NXJ has some minor additions to LCP to make its tools work better.

The implementation of LCP is in the `lejos.nxt.comm.LCP` class. As leJOS sensors and motors work a bit differently than the standard firmware, the semantics of LCP on leJOS are not always identical to the standard LEGO firmware.

The start-up menu – `StartUpText.java` – uses LCP to support the leJOS NXJ tools and third-party tools. This means that LCP commands can be executed over Bluetooth or USB when the menu is running.



## Understanding leJOS NXJ use of memory

The NXT has 256kb of flash memory and 64kb of RAM.

Flash memory can be read like RAM (access is a bit slower) but can only be written in 256-byte pages by specific hardware instructions,. Flash memory cannot be read while a page is being written.

The leJOS NXJ firmware is written in a combination of C and ARM assembler code. It consists of the initialization code, the Java VM and device drivers for all the hardware subsystems. The leJOS firmware is a complete firmware replacement and has no reliance on the standard LEGO firmware. The first 32kb of flash memory is allocated to the leJOS NXJ firmware. Most code is executed from flash memory, but a small amount (e.g. the code that writes pages of flash memory) is copied to RAM. Read-only data is held in flash memory but read/write data is copied to RAM. The firmware uses a fixed size stack and interrupt stack.

The leJOS NXJ Java VM executes one Java program at a time. This can either be a user program or the leJOS start-up menu. One Java program can execute another. When this is done the first Java program is removed from memory, and the second one is then executed. This is how the start-up menu executes user programs.

The start-up menu occupies up to 48kb of memory that starts at address 32k (i.e. after the end of the firmware). The last word of the 48kb allocated to the start-up menu holds the size of the start-up menu).

Java programs execute from flash memory. Static read-only data is held in flash memory. Static read-write data is copied to RAM. Objects are created in a heap that starts at the top of the RAM and grows downwards. The Java stack starts at the bottom of free RAM memory and grows up. A garbage collector frees memory used by unreferenced objects when the heap becomes full.

## ***Understanding how the NXJ tools work***

### **nxjflash**

In order to flash firmware to the NXT, the NXT must be in firmware update mode. The NXT is put into firmware update mode by pressing the reset button for 4 seconds or more. This causes the NXT to run a small boot assistant program called SAM-BA. The was written by Atmel, the maker of the ARM chipset that the NXT uses. SAM-BA includes a USB driver and accepts commands sent over the USB link. These command allows data to be uploaded to RAM and code to be executed. Early version of leJOS used this mechanism to run in RAM before there was a flash version of leJOS NXJ.

On Microsoft Windows and MAC OS X, the standard LEGO USB drivers are used. These come with the LEGO software, which is why this software must be installed. On Microsoft windows, when the NXT is in firmware update mode, a USB cable is attached to your PC and the NXT is switched on (by pressing the orange button), if you go to Control Panel > System > Hardware > Device Manager you will see under “Lego Devices” an entry for the USB driver, labelled “LEGO MINDSTORMS NXT Firmware Update Mode”.

nxjflash uses the libusb library to drive the USB interface. On Linux libusb provides the USB driver. On Microsoft Windows, you need to run the libusb-win32 filter driver. This allows nxjflash to drive the LEGO MINDSTORMS NXT Firmware Update Mode USB driver.

nxjflash drives libusb via a library written by David Anderson, called libnxt. This supports SAM-BA commands. To flash new firmware, it uploads the firmware image a 256-byte page at a time and then executes a small RAM-resident routine to write the page to flash memory. In this way the leJOS NXJ firmware, lejos\_nxt\_rom.bin is written to flash memory. This occupies the first 32kb of flash memory.

nxjflash also uploads the leJOS NXJ start-up menu StartUpText.bin. This menu is written in Java and built like any other leJOS NXJ Java programs. It implements the leJOS NXJ menu system and supports threads for executing Lego Communication Protocol (LCP) commands over USB and Bluetooth. StartUpText.bin resides in flash memory starting at address 32k. 48kb of flash memory is reserved for it. The highest address word in the 49k. When both the firmware and the menu have been uploaded, nxjflash sends a SAM-BA command to the NXT causing it to jump to address zero and the leJOS NXJ firmware executes.

## **nxjupload**

nxjupload uploads programs or other files over USB or Bluetooth. It is a command line interface that is suitable for use from command windows, ant scripts, and as an external command from IDEs such as Eclipse.

nxjupload sends LCP system commands to the NXT to upload the file. The commands are OPEN\_WRITE, WRITE and CLOSE.

The call of nxjupload looks like:

```
nxjupload [options] <binary-file>
```

By default nxjupload first looks for a NXT connected by USB. If it does not find one, it tries Bluetooth. It does a Bluetooth inquiry looking for NXTs. If it finds any it tries to connect to each one, and uploads the file to the first one it successfully connects to. This means that if you have multiple NXTs, it will upload to the one that is currently switched on.

This behaviour can be modified by the following options:

- -b or --bluetooth: only Bluetooth is tried
- -u or --usb: only USB is tried
- -n or --name: the upload is done to the named NXT. A Bluetooth inquiry is done, but only the named NXT is looked for.
- -d or --address <address>: the upload is to the device with this Bluetooth address. USB is not tried and no Bluetooth inquiry is done.
- -r or --run: run the program after upload by sending a STARTPROGRAM LCP command.

nxjupload uses the apache Commons CLI command line processor.

## **nxjlink**

nxjlink call the linker (class js.tinyvm.TinyVM).

The call looks like:

```
nxjlink [options] class -o <binary file>
```

The -cp or --classpath option is used to specify where the linker looks for classes. Note that the linker classpath is separate to the classpath used to execute js.tinyvm.TinyVM.

The linker first looks for the specified class in the linker classpath, and then looks for all classes that this references to form a “closure” of the classes. The linker class path should

include classes.jar and all the user classes in the program. The class specified on the command line must be the one containing the main method. The Jakarta apache Byte Code Engineering Library (BCEL) is used to process the class files.

The linker omits methods that have not been referenced unless the `-a` or `-all` flag is specified. The way this is done uses a simple algorithm and does not manage to omit all unreferenced methods.

The linker produces a leJOS NXJ binary file and writes it to the file specified in the `-o` parameter.

nxjlink needs to know the byte order of the processor it is producing the binary for. For the NXT this is set by `-writeorder LE` for Little-Endian.

If the `-v` or `-verbose` flag is set, a list of the classes and methods in the binary is output to standard out.

nxjlink uses the apache Commons CLI library for command line processing.

## **nxj**

The nxj command

## **nxjc**

## **nxjbrowse**

# **Developing leJOS**

# Developing leJOS

*Developing Java classes*

*Developing the firmware*

*Developing PC Tools*

**Supporting new sensors**

A/D Sensors

I2C Sensors

# **Getting Started on MAC OS X**

# **Getting Started on Linux**