Using I2C with Java leJOS

Versión 0.1

Juan Antonio Breña Moral

24-ago-08

Index

1.- Introduction 4

1.1.- Goals 4

1.1.1.- About this document 4

1.2.- LeJOS Project 4

1.3.- NXT Brick 5

1.3.1.- NXT Sensors used in the eBook 6

1.4.- About the author 8

2.- I2C Protocol 9

2.1.- Introduction 9

2.2.- I2C Bus terminology 10

2.3.- Terminology for bus transfer 10

3.- LeJOS and I2C 11

3.1.- LeJOS API 11

3.2.- I2C Examples with leJOS 12

3.3.- Migrating code I2C from others platforms 13

3.3.1.- Migrating I2C Code from RobotC to Java leJOS 13

3.3.2.- Migrating I2C Code from NXC to Java leJOS 14

Revision History

Name	Date	Reason For Changes	Version
Juan Antonio Breña Moral	24/08/2008	First publication	0.1

1.- Introduction

1.1.- Goals

Many developers around the world choose leJOS, Java for Lego Mindstorm, as the main platform to develop robots with NXT Lego Mindstorm. I consider that this eBook will help leJOS community, Lego Mindstorm community, Robot's developers and Java fans to develop better software.

Robotics will be very important for the humanity in the next 10 years and this eBook is an effort to help in this way.

Many people spend several hours in their robotics projects with problems with wires & electronics, protocols and problems with programming languages, Lego Mindstorm is easy and Java/leJOS is an excellent platform to demonstrate your software engineering skills to develop better robots. NXT Brick is the easiest way to enter in the robotics world and leJOS, the best platform in the moment to use software engineering ideas.

Enjoy, Learn, Contact with me to improve the eBook and share your ideas.
Download latest eBook release here: http://juanantonio.info/jab_cms.php?id=206

Juan Antonio Breña Moral.
www.juanantonio.info

1.1.1.- About this document

This document has been written to explain how to discover I2C protocol and how to use with your Java leJOS projects

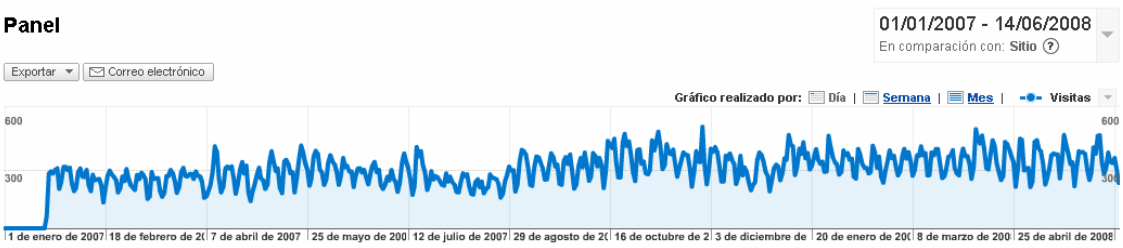
1.2.- LeJOS Project

LeJOS is Sourceforge project created to develop a technological infrastructure to develop software into Lego Mindstorm Products using Java technology.

Currently leJOS has opened the following research lines:

1. NXT Technology
    a. NXJ
    b. LeJOS PC API
    c. iCommand
2. RCX Technology
    a. leJOS for RCX

LeJOS project's audience has increased. Currently more than 500 people visit the website every day.



This eBook will focus in NXT technology with NXJ using a Windows Environment to develop software.

1.3.- NXT Brick

The NXT is the brain of a MINDSTORMS robot. It's an intelligent, computer-controlled LEGO brick that lets a MINDSTORMS robot come alive and perform different operations.

Motor ports
The NXT has three output ports for attaching motors - Ports A, B and C

Sensor ports
The NXT has four input ports for attaching sensors - Ports 1, 2, 3 and 4.

USB port
Connect a USB cable to the USB port and download programs from your computer to the NXT (or upload data from the robot to your computer). You can also use the wireless Bluetooth connection for uploading and downloading.

Loudspeaker
Make a program with real sounds and listen to them when you run the program

NXT Buttons
Orange button: On/Enter /Run
Light grey arrows: Used for moving left and right in the NXT menu

Dark grey button: Clear/Go back

NXT Display
Your NXT comes with many display features - see the MINDSTORMS NXT Users Guide that comes with your NXT kit for specific information on display icons and options

Technical specifications

- 32-bit ARM7 microcontroller
- 256 Kbytes FLASH, 64 Kbytes RAM
- 8-bit AVR microcontroller
- 4 Kbytes FLASH, 512 Byte RAM
- Bluetooth wireless communication (Bluetooth Class II V2.0 compliant)
- USB full speed port
- 4 input ports, 6-wire cable digital platform (One port includes a IEC 61158 Type 4/EN 50 170 compliant expansion port for future use)
- 3 output ports, 6-wire cable digital platform
- 100 x 64 pixel LCD graphical display
- Loudspeaker - 8 kHz sound quality. Sound channel with 8-bit resolution and 2-16 KHz sample rate.
- Power source: 6 AA batteries

1.3.1.- NXT Sensors used in the eBook

NXT Sensors used in the document are the following:

- NXT Motor
- Ultrasonic Sensor
- Compass Sensor
- NXTCam
- Tilt Sensor
- NXTCam
- NXTe

NXT Motor

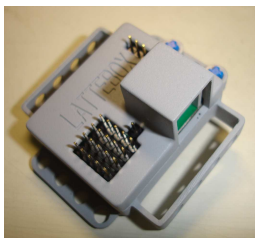
Ultrasonic Sensor

Compass Sensor

Tilt Sensor

NXTCam

NXTe



1.4.- About the author

Juan Antonio Breña Moral has collaborated in leJOS Research team since 2006. He works in Europe leading Marketing, Engineering and IT projects for middle and large customers in several markets as Defence, Telecommunications, Pharmaceutics, Energy, Automobile, Construction, Insurance and Internet.

Further information:
www.juanantonio.info
www.esmeta.es

2.- I2C Protocol

2.1.- Introduction

I2C (Inter-Integrated Circuit) is a multi-master serial computer bus invented by Philips that is used to attach low-speed peripherals to a motherboard, embedded system, or cellphone.

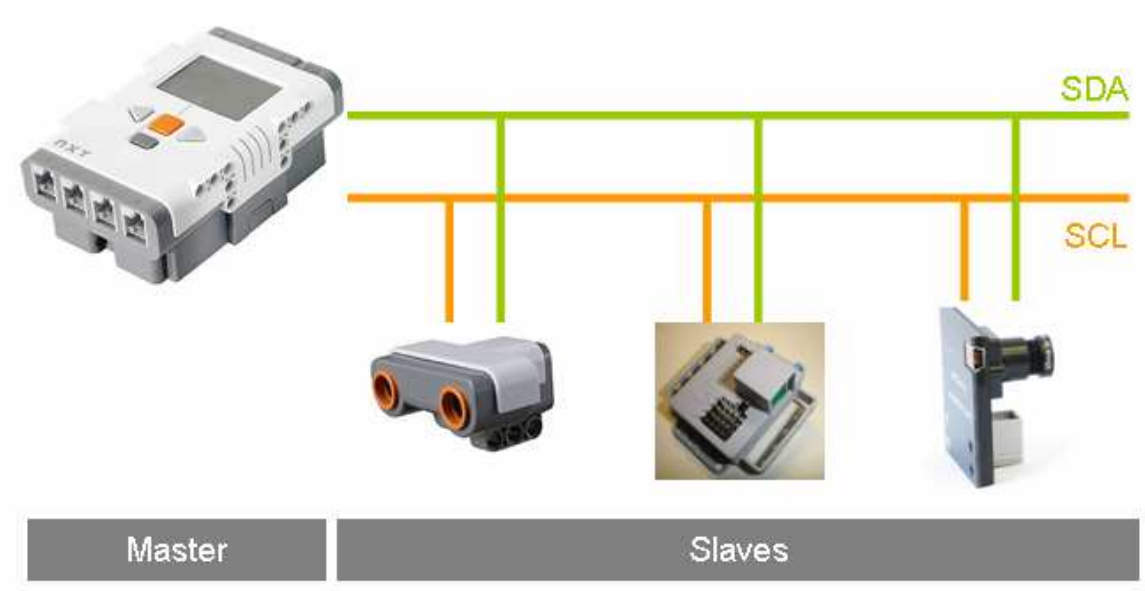
I2C uses only two bidirectional open-drain lines, Serial Data (SDA) and Serial Clock (SCL), pulled up with resistors.

Every device hooked up to the bus has its own unique address.

The bus has two roles for nodes: master and slave:

- Master node: node that issues the clock and addresses slaves
- Slave node: node that receives the clock line and address.

In NXT world the I2C Diagrama should be the following:



There are four potential modes of operation for a given bus device, although most devices only use a single role and its two modes:

- **Master transmit:** master node is sending data to a slave
- **Master receive:** master node is receiving data from a slave
- **Slave transmit:** slave node is sending data to a master
- **Slave receive:** slave node is receiving data from the master

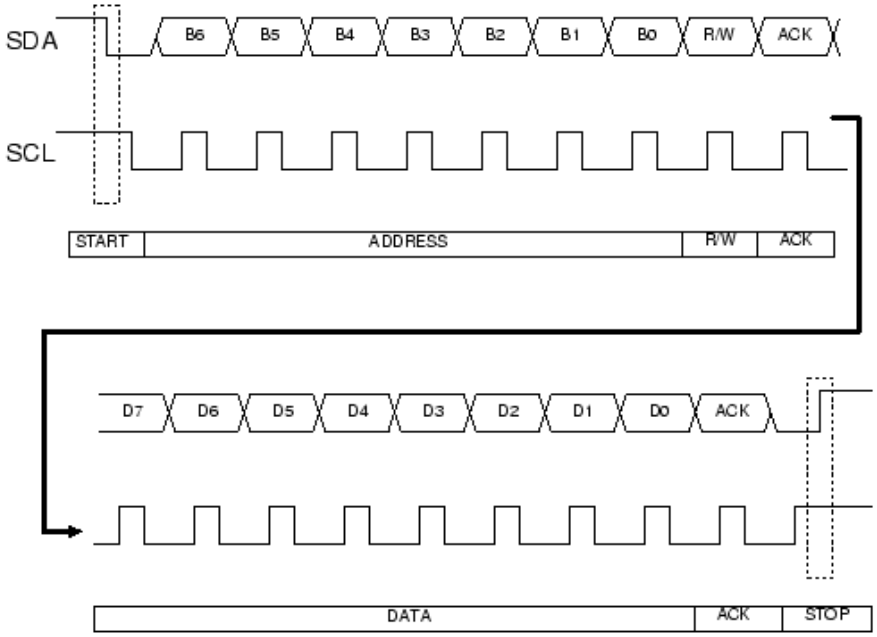
2.2.- I2C Bus terminology

The terminology used when you work with I2C is the following:

- **Transmitter:** The device that sends data to the bus. A transmitter can either be a device that puts data on the bus on its own accord (a 'master-transmitter'), or in response to a request from the master (a 'slave-transmitter').
- **Receiver:** the device that receives data from the bus. A receiver can either be a device that receives data on its own request (a 'master-receiver), or in response to a request from the master (a 'slave-receiver).
- **Master:** the component that initializes a transfer (Start command), generates the clock (SCL) signal and terminates the transfer (Stop command). A master can be either a transmitter or a receiver.
- **Slave:** the device addressed by the master. A slave can be either receiver or transmitter.
- **Multi-master:** the ability for more than one master to co-exist on the bus at the same time without collision or data loss. Typically "bit-banged" software implemented masters are not multi-master capable. Parallel to I²C bus controllers provide an easy way to add a multi-master hardware I²C port to DSPs and ASICs.
- **Arbitration:** the prearranged procedure that authorizes only one master at a time to take control of the bus.
- Synchronization - the prearranged procedure that synchronizes the clock signals provided by two or more masters.
- **SDA:** data signal line (Serial DAta)
- **SCL:** clock signal line (Serial CLock)

2.3.- Terminology for bus transfer

- **F (FREE):** the bus is free or idle; the data line SDA and the SCL clock are both in the high state.
- **S (START) or R (RESTART):** data transfer begins with a Start condition. The level of the SDA data line changes from high to low, while the SCL clock line remains high. When this occurs, the bus becomes 'busy'.
- **C (CHANGE):** while the SCL clock line is low, the data bit to be transferred can be applied to the SDA data line by a transmitter. During this time, SDA may change its state, as long as the SCL line remains low.
- **D (DATA):** a high or low bit of information on the SDA data line is valid during the high level of the SCL clock line. This level must be kept stable during the entire time that the clock remains high to avoid misinterpretation as a Start or Stop condition.
- **P (STOP):** data transfer is terminated by a Stop condition. This occurs when the level on the SDA data line passes from the low state to the high state, while the SCL clock line remains high. When the data transfer has been terminated, the bus is free once again.



3.- LeJOS and I2C

3.1.- LeJOS API

LeJOS project supports I2C devices connected to NXT brick. Every object which uses I2C protocol inherates from **I2CSensor**

The NXT devices which use I2C are:

- ColorSensor
- CompassSensor
- IRSeeker
- NXTe
- NXTCam
- OpticalDistanceSensor
- PSPNXController
- RCXLink
- RCXMotorMultiplexer
- RCXSensorMultiplexer
- TiltSensor
- UltrasonicSensor

The class I2CSensor has the following I2C methods:

- setAddress
- sendData
- getData

Further information about leJOS API here:
http://lejos.sourceforge.net/nxt/nxj/api/index.html

When it is necessary to write a class to manage a new leJOS device the question to do are:

- What is the I2C address to write and read data?
- What is the list of I2C registers to write and read data?
- How to interpret the values from I2C registers?

3.2.- I2C Examples with leJOS

To explain the concepts, I will use a NXT I2C Device, Mindsensors NXTServo. This device has been developed to manage RC Servos.

If we want to read the battery connected to that device, it is necessary to know the following parameters:

1. NXTServo I2C Address: **0xb0**
2. NXTServo I2C Register to read battery level: **0x41**

Now I will write a simple example which read the battery from NXTServo:

```
public class NXTServoTest{

	public static void main(String[] args){
		DebugMessages dm = new DebugMessages();
		dm.setLCDLines(6);
		dm.echo("Testing NXT Servo");

		MSC msc = new MSC(SensorPort.S1);
		msc.addServo(1,"Mindsensors RC Servo 9Gr");

		while(!Button.ESCAPE.isPressed()){
			dm.echo(msc.getBattery());
		}
		dm.echo("Test finished");
	}
}
```

The class MSC, Mindsensor Servo Controller, manages until 8 RC Servos. I will show 2 internal methods in the class MSC:

The constructor:

```
public static final byte NXTSERVO_ADDRESS = (byte)0xb0;

public MSC(SensorPort port){
	super(port);
	port.setType(TYPE_LOWSPEED_9V);
	this.setAddress(MSC.NXTSERVO_ADDRESS);

	this.portConnected = port;
	arrServo = new ArrayList();
}
```

If you observe the code, all I2C operation will use the address 0xb0

The method getBattery:

```
public int getBattery(){
	int I2C_Response;
	byte[] bufReadResponse;
```

```
        bufReadResponse = new byte[8];
        byte kSc8_Vbatt = 0x41;//The I2C Register to read the battery
        I2C_Response = this.getData(kSc8_Vbatt, bufReadResponse, 1);
        return(37*(0x00FF  &  bufReadResponse[0]));//  37  is  calculated
from
        //supply from NXT =4700 mv /128
}
```

In this example we read the I2C register 0x41 which store battery level. Every I2C action has a response. If the response is 0 then it is a success if the result is not 0 then it was a failure. Besides when you read a I2C register, you have to use a buffer, in this case bufReadResponse.

3.3.- Migrating code I2C from others platforms

When you develop NXT software, it is a usual that you get ideas from others developers who likes others platforms. In this section I will explain how to migrate I2C RobotC and NXC code

3.3.1.- Migrating I2C Code from RobotC to Java leJOS

RobotC has the following I2C functions to read and write registers.

Function	Description
sendI2CMsg(nPort, sendMsg, nReplySize);	Send an I2C message on the specified sensor port.
nI2CBytesReady[]	This array contains the number of bytes available from a I2C read on the specified sensor port.
readI2CReply(nPort, replyBytes, nBytesToRead);	Retrieve the reply bytes from an I2C message.
nI2CStatus[]	Currents status of the selected sensor I2C link.
nI2CRetries	This variable allows changing the number of message retries. The default action tries to send every I2C message three times before giving up and reporting an error. Unfortunately, this many retries can easily mask any faults that can exist.
SensorType[]	This array is used to configure a sensor for I2C operation. It also indicates whether ‘standard’ or ‘fast’ transmission should be used with this sensor.

Now I will show an example with the same method getBattery:

```
/*===================================
**
** Read the battery voltage data from
** NXTServo module (in mili-volts)
**
===================================*/
int  Get_Batt_V()
{
  byte sc8Msg[5];
  const int kMsgSize       = 0;
  const int kSc8Address    = 1;
  const int kReadAddress   = 2;
      byte replyMsg[2];
```

```
  // Build the I2C message
  sc8Msg[kMsgSize]         = 2;
  sc8Msg[kSc8Address]      = kSc8ID ;
  sc8Msg[kReadAddress]     = kSc8_Vbatt ;

  while (nI2CStatus[kSc8Port] == STAT_COMM_PENDING);
  {
   // Wait for I2C bus to be ready
  }
  // when the I2C bus is ready, send the message you built
  sendI2CMsg(kSc8Port, sc8Msg[0], 1);

  while (nI2CStatus[kSc8Port] == STAT_COMM_PENDING);
  {
   // Wait for I2C bus to be ready
  }
  // when the I2C bus is ready, send the message you built
  readI2CReply(kSc8Port, replyMsg[0], 1);

  return(37*(0x00FF & replyMsg[0]));  // 37 is calculated from

      //supply from NXT =4700 mv /128
}
```

Now the migration to Java leJOS:

```
/**
 * Read the battery voltage data from
 * NXTServo module (in mili-volts)
 *
 * @return
 */
public int getBattery(){
      int I2C_Response;
      byte[] bufReadResponse;
      bufReadResponse = new byte[8];
      byte kSc8_Vbatt = 0x41;//The I2C Register to read the battery
      I2C_Response = this.getData(kSc8_Vbatt, bufReadResponse, 1);
      return(37*(0x00FF & bufReadResponse[0]));// 37 is calculated
from
      //supply from NXT =4700 mv /128
}
```

3.3.2.- Migrating I2C Code from NXC to Java leJOS

NXC Programming Languaje has a set of functions to manage I2C:

Function	Description
LowspeedWrite(port, returnlen, buffer)	This method starts a transaction to write the bytes contained in the array buffer to the I2C device on the specified port. It also tells the I2C device the number of bytes that should be included in the response. The maximum number of bytes that can be written or read is 16. The port may be specified using a constant (e.g., IN_1, IN_2, IN_3, or IN_4) or a variable. Constants should be used where possible to avoid blocking access to I2C devices on other ports by code running on other

	threads. `x = LowspeedWrite(IN_1, 1, inbuffer);`
LowspeedStatus(port, out bytesready)	This method checks the status of the I2C communication on the specified port. If the last operation on this port was a successful LowspeedWrite call that requested response data from the device then bytesready will be set to the number of bytes in the internal read buffer. The port may be specified using a constant (e.g., `IN_1`, `IN_2`, `IN_3`, or `IN_4`) or a variable. Constants should be used where possible to avoid blocking access to I2C devices on other ports by code running on other threads. If the return value is 0 then the last operation did not cause any errors. Avoid calls to LowspeedRead or LowspeedWrite while LowspeedStatus returns `STAT_COMM_PENDING`. `x = LowspeedStatus(IN_1, nRead);`
LowspeedCheckStatus(port)	This method checks the status of the I2C communication on the specified port. The port may be specified using a constant (e.g., IN_1, IN_2, IN_3, or IN_4) or a variable. Constants should be used where possible to avoid blocking access to I2C devices on other ports by code running on other threads. If the return value is 0 then the last operation did not cause any errors. Avoid calls to LowspeedRead or LowspeedWrite while LowspeedStatus returns STAT_COMM_PENDING. x = LowspeedCheckStatus(IN_1);
LowspeedBytesReady(port)	This method checks the status of the I2C communication on the specified port. If the last operation on this port was a successful LowspeedWrite call that requested response data from the device then the return value will be the number of bytes in the internal read buffer. The port may be specified using a constant (e.g., IN_1, IN_2, IN_3, or IN_4) or a variable. Constants should be used where possible to avoid blocking access to I2C devices on other ports by code running on other threads. x = LowspeedBytesReady(IN_1);
LowspeedRead(port, buflen, out buffer)	Read the specified number of bytes from the I2C device on the specified port and store the bytes read in the array buffer provided. The maximum number of bytes that can be written or read is 16. The port may be specified using a constant (e.g., IN_1, IN_2, IN_3, or IN_4) or a variable. Constants should be used where possible to avoid blocking access to I2C devices on other ports by code running on other threads. If the return value is negative then the output buffer will be empty.

	x = LowspeedRead(IN_1, 1, outbuffer);
I2CWrite(port, returnlen, buffer)	This is an alias for LowspeedWrite. x = I2CWrite(IN_1, 1, inbuffer);
I2CStatus(port, out bytesready)	This is an alias for LowspeedStatus. x = I2CStatus(IN_1, nRead);
I2CCheckStatus(port)	This is an alias for LowspeedCheckStatus. x = I2CCheckStatus(IN_1);
I2CBytesReady(port)	This is an alias for LowspeedBytesReady. x = I2CBytesReady(IN_1);
I2CRead(port, buflen, out buffer)	This is an alias for LowspeedRead. x = I2CRead(IN_1, 1, outbuffer);
I2CBytes(port, inbuf, in/out count, out outbuf)	This method writes the bytes contained in the input buffer (inbuf) to the I2C device on the specified port, checks for the specified number of bytes to be ready for reading, and then tries to read the specified number (count) of bytes from the I2C device into the output buffer (outbuf). The port may be specified using a constant (e.g., IN_1, IN_2, IN_3, or IN_4) or a variable. Returns true or false indicating whether the I2C read process succeeded or failed. This is a higher-level wrapper around the three main I2C functions. It also maintainsa "last good read" buffer and returns values from that buffer if the I2C communication transaction fails. x = I2CBytes(IN_4, writebuf, cnt, readbuf);

To explain the concepts, will show an example from Lattebox NXTe which has leJOS and NXC support:

```
byte  bufConfigureSPI[] = {0x50, 0xF0, 0x0C};

void LowspeedWait()
{
 while(true){
    if (LowspeedCheckStatus(IN_3) == NO_ERR) break;
   }
}

void nxt_init()
{
     ResetSensor(IN_3);
     Wait(100);
     SetSensorType(IN_3, IN_TYPE_LOWSPEED);
     SetSensorMode(IN_3, IN_MODE_RAW);
     ResetSensor(IN_3);
     Wait(100);
```

```
      LowspeedWait();
      LowspeedWrite(IN_3,0,bufConfigureSPI);
      LowspeedWait();

}


public static final byte NXTE_ADDRESS = 0x28;
private final byte REGISTER_IIC = (byte)0xF0;//NXTe IIC address

/**
 * Constructor
 *
 * @param port
 */
public NXTe(SensorPort port){
      super(port);

      port.setType(TYPE_LOWSPEED_9V);
      port.setMode(MODE_RAW);

      portConnected = port;

      arrLSC = new ArrayList();

      this.setAddress((int) NXTE_ADDRESS);
      int I2C_Response;
      I2C_Response = this.sendData((int)this.REGISTER_IIC,
(byte)0x0c);
}
```