A course on Wireless Sensor Networks (WSNs)

Luis Sanabria, Jaume Barcelo

April 11, 2013

Contents

Acknowledgements

Thanks to Alejandro Andreu for opening the path in the explorarion of wireless sensor networks and helping in the preparation of these assignments.

Chapter 1

About the course

1.1 Course Data

Code: 21754

Course name: "Xarxes de Sensors Sense Fils"

Teacher: Luis Sanabria and Jaume Barcelo

Credits: 4

Year: 3rd or 4th year (optional)

Trimester: Spring

1.2 Introduction

The reduction in price and size of computing and wireless communication platforms over the last years opens a new possibility for gathering and processing information: Wireless Sensor Networks. A wireless sensor node is an electronic device of small dimensions that gathers measures from the environment and transmit the data wirelessly. In wireless sensor nodes, communication is often established with other wireless sensor

nodes to exchange or pass information. It is common to have this data directed to an special device that gathers all the data and is called the network sink. As wireless sensor nodes are often battery-powered, energy saving is a relevant issue in these networks.

What follows is an extract of the first pages of [10].

> Wireless Sensor Networks (WSNs) are a result of significant breakthroughs on wireless transceiver technology, the need of event sensing and monitoring. One might think of a WSN as the skin of our bodies; apart from its importance on many other subjects, our skin senses events nearby it, like touch, temperature changes, pressure and so forth. These events are generated by an external entity, the nerves or sensors of our skin are capable to react to such events and transmit this information to the brain.
>
> There are enormous differences among characteristics of WSN and the skin, but the example given above will work as head start to understanding the technology. For instance, our skin sends the sensed event information towards the brain through the nerves, we could safely relate this medium to a wired network infrastructure. While in WSN, as its name suggests sends the sensed data towards a central node (Sink) via a wireless medium. Because of the limited radio range of each node, the route to the Sink is generally composed of jumps through different nodes (which is called a multihop route).
>
> The majority of wireless nodes in a WSN are very constrained devices due to the restrictions in

costs and sometimes harsh environments where these networks are deployed. These constraints go from cost, processing power, memory, storage, radio range, spectrum and, more importantly, battery life. One of the most popular low-end nodes model, the TelosB, is equipped with 16 MHz CPU, very small flash memory (48 KB avg.), about 10 KB of RAM and works on the very crowded 2.4 GHz spectrum at rates around 250 Kbps. These limitations force WSN engineers to design applications capable of working with low processor-intensive tasks and powered with limited battery (usually two AA batteries).

Many WSN applications process the sensed event before sending the data, this processing tries to reduce the information to send. As mentioned in [1], it is less energy consuming to process one bit of information than sending it. WSN protocols and applications are tailored to power conservation rather than throughput, mainly due to cost, dimension, processing and power constraints.

WSNs may contain different kind of sensors that help monitor metrics related to: temperature, humidity, pressure, speed, direction, movement, light, soil makeup, noise levels, presence or absence of certain kinds of objects, mechanical stress and vibration. Also further information like node location can be derived from a Global Positioning System (GPS) device embedded at each node.

Because of the variety of measures than can be monitored with these small and (generally) cheap devices, a wide range of applications have been de-

veloped; the authors of [1] divide them in: military, environmental, health, home and industry applications.

- *Military Applications:* one of the first applications of WSNs. The main advantages in this area are the fact that the deployment of low cost sensors (that are subject to destruction in a battlefield) proposes a cheaper approach to sensing different types of metrics, which in turn brings new challenges to WSN applications (increased power and processing constraints). Some of the applications are related to: monitoring the movement of troops, equipment and ammunition, battlefield surveillance, terrain reconnaissance, damage assessments, snipper detection [7], [8] and threat detection, as in the case of biological, radiological or chemical attacks.

- *Environmental Applications:* most of these applications are related to animal tracking, weather conditions and threat contention [9], [11].

- *Health Applications:* a great deal of these applications are dedicated to monitor patients inside hospitals and provide them with better care. This is achieved by tracking the patients vitals or other information of interest and making it available to doctors at any time from anywhere securely through the Internet.

- *Home Applications:* technology is making its way inside our homes from various fronts, and WSN are no exception. Sensor nodes inside domestic devices will result in an increased interaction among them and allow access via the Internet. Theses applications are of great importance in fields like domotics towards a smart home/work environment. Home surveillance and multimedia WSNs for home environments are also a growing field of research.

- *Industrial Applications:* historically the monitoring of material fatigue was made by experts introducing the observed situation inside PDA devices to be collected on a central site for processing. Further sensing techniques were developed on the form of wired sensors; nevertheless its implementation was slow and expensive due the necessary wiring. WSNs bring the best of both methods by sensing the events without the need of expert personnel and the cost of wiring.

- Other implementations as mentioned in [1] are: inventory management, product quality monitoring, smart offices/houses; guidance in automatic manufacturing environments, interactive museums, factory process control and automation, machine diagnosis, transportation, vehicle tracking and detection, spectrum sensing for cognitive radio networks, underground and underwater monitoring.

1.3 Syllabus

- Lectures
    1. Introduction to WSNs.
    2. Arduino Platform.
    3. XBee and XBee explorer. AT commands.
    4. XBee API mode.
    5. A sensor network with Arduino.
    6. A sensor network without Arduino.
    7. Publishing sensed data
    8. Invited talk
    9. Quiz
- Labs and seminars
    1. Blinking LED (Dimming optional)
    2. Blinking LED with push-button (dimming optional)
    3. XBee chat
    4. Wireless doorbell
    5. Sunset sensor
    6. Sensor network with Arduino
    7. Sensor network with XBee in API mode
    8. Sleeping and actuating
    9. Uploading sensed data to the Internet

1.4 Bibliography

Most of the lab assignments follow the book that you can find at the university library:

Robert Faludi "Building Wireless Sensor Networks" ([6]).

The following list of "common mistakes" can be very useful when debugging your projects:

`http://www.faludi.com/projects/common-xbee-mistakes/`

Check also:

Massimo Banzi "Getting Started with Arduino".

1.5 Evaluation Criteria

The grading is distributed as follows:

- Quiz, 10%
- Each lab assignment, 10%

It is necessary to obtain a decent mark in all the different evaluation aspects. To pass the course, 50 out of the total 100 points need to be obtained.

1.6 Team work

You will work in teams of three people. Try to make the groups as heterogeneous as possible: people that are experienced with Arduino and people that are not, people from different majors, people with strong programming skills and people good at electronics, etc.

Each group delivers a single report per session and the teachers may ask questions to individual members of the team.

1.7 Use your imagination

The lab assignments are somewhat easy. The goal is that you complement what you do in the lab with other ideas of your own. You are encouraged to explore WSNs beyond the basics introduced in the assignments and document your findings in the reports. Doing something on your own beyond the assignment takes a lot of effort and is time-consuming. Nevertheless, as engineers, we should be able to come up with new ideas and solutions on our own.

1.8 Non-stop Arduino

In our school there are two additional courses that make use Arduino: "Sensors and data acquisition" and "Interactive Systems".

1.9 Survival guide

1.9.1 Questions and doubts

We like to receive questions and comments. Normally, the best moment to express a doubt is during the class, as it is likely that many people in the class share the same doubt. If you feel that you have a question that needs to be discussed privately, we can discuss it right after the class.

1.9.2 Continuous feedback

At the end of the lecture, we will ask you to anonymously provide some feedback on the course *using a form like this one*. In particular, I always want to know:

- What is the most interesting thing we have seen in class.
- What is the most confusing thing in the class.
- Any other comment you may want to add.

  In labs, I will ask each group to hand in a short (few paragraphs) description of the work carried out in class, and the members of the group that have attended the class. Note that this is different from the deliverables, which are the ones that are actually graded.

1.9.3 How to make you teachers happy

Avoid speaking while we are talking.

Chapter 2

Introduction to Arduino

2.1 Open Hardware

"There's a fine line between open source and stupidity", says Massimo Banzi to a reporter from Wired Magazine while having dinner at a restaurant in Milan.

Banzi is the man behind Arduino, an open hardware platform. The open about it relates to the fact that the device's manufacturing schematics, programming language and software development environment are free and open source. This basically means that everyone interested on building hardware-coupled solutions may take an Arduino board's schematics, modify it at will, send the new design to a China manufacturer and get the final product back home for around €10 [5].

Open hardware is supported by a variety of available licenses (like open software with LGPL, GPL, Copyleft, and others) that ensure that the protected platform can be copied, enhanced and even sold, but always recognizing the original authors. It also ensures that the resulting products are open as the original.

2.2 The Arduino Platform

Arduino was developed to teach Interaction Design [3], that meant that it required the ability to sense the surroundings and do something about it.

The platform is equipped with simple digital and analog input/output interfaces, that can be programmed to sense or react to some events. Figure 2.1 shows the Arduino Duemilanove board.

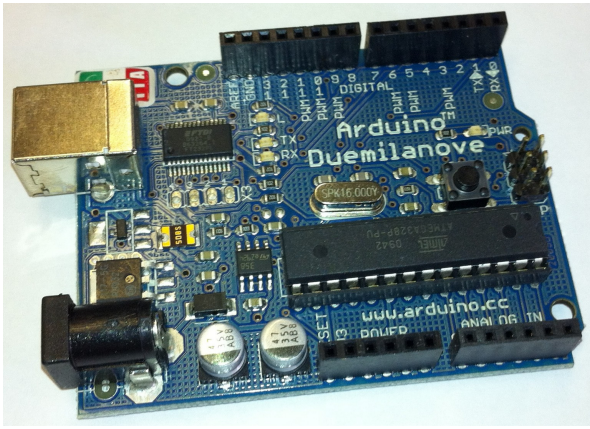

Figure 2.1: Arduino Duemilanove board

There are numerous sensors and actuators that work with Arduino. In relation to sensors: temperature, air pollution, light, GPS modules and sound are among the popular; as LEDs, speakers and digital/analog outputs are common actuators. Also, interfaces like buttons can be programmed and used as a human interactive input.

The design and electrical components of the Arduino board are available for anyone [2]. Figure 2.2 shows the connections layout of the Duemilanove model (compare with Figure 2.1).

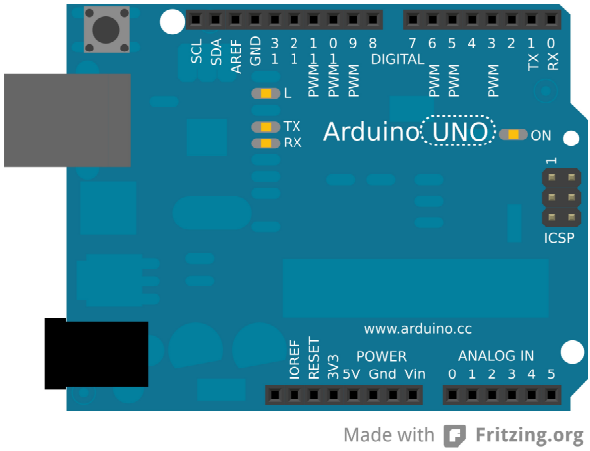

Figure 2.2: Arduino Duemilanove board: layout

Chapter 3

Introduction to XBee

One of the main characteristics of WSNs is the ability each node has to wirelessly communicate with other nodes. During this course we will be doing this with ZigBee protocol compliant radios, like XBee [6].

Throughout this section you will be introduced to the different components and code that will allow you to set a basic wireless network with XBee modules.

3.1 The Zigbee and IEEE 802.15.4 standards

XBee is a Zigbee compliant hardware. Zigbee is a specification that is build on top of IEEE 802.15.4. Zigbee covers the upper layers of the protocol stack while IEEE 802.15.4 covers the lower layers (MAC and PHY).

ZigBee is intended for low-throughput, low-power, low-cost applications.For this reason, it is much simpler than other protocols such as WiFi (IEEE 802.11). It has support for mesh topologies, which means that ZigBee devices relay messages for

each other through multiple wireless hops. The name ZigBee comes from the fact that the bees can dance to pass messages to each other, also in a multi-hop fashion.

There are channesl in the 868 MHz, 915 MHz and 2.4 GHz ISM bands and the speed is up to 250 kbps in the 2.4 GHz band.

Applications include domotics and wireless sensor and actuator networks.

3.1.1 ZigBee profiles

- ZigBee Co-ordinator: It is the most powerful device. There is a single coordinator in each network. It is the node that creates the network and the other nodes simply join. Quite often, this is the sink of the wireless sensor network that gathers all the data that is transmitted. One of the co-ordination tasks is to assign short addresses as will be explained in the next subsection.

- ZigBee Router: Routers are intermediate devices. They can relay packets for other nodes. They join a network that already exists and then announce it using beacons. Therefore, they can have "children", nodes that join the network by establishing communication with the router.

- End Devices: These are the simplest devices. They cannot forward packets; they cannot have children that depend on them and quite often they sleep to save energy.

3.1.2 Network layer and addressing

Addressing follows a hierarchical scheme that is explained in detail in [4]. An example is provided in Fig. 3.1

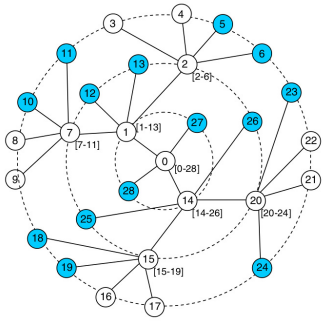

Figure 3.1: Hierarchical address scheme in ZigBee (picture from [4])

The hierarchical scheme is appropriate for tree topologies and tree routing. Three different topologies are supported in ZigBee: Star, tree and mesh. They are shown in Fig. 3.2.

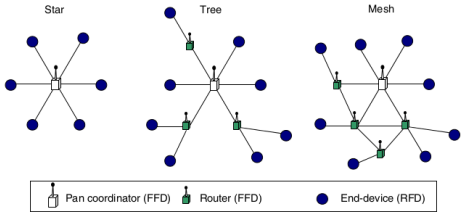

Figure 3.2: Supported topologies in ZigBee (picture from [4])

The mesh topology requires a routing protocol and storing routing tables. It is very resource consuming and a device might not have the resources to execute it. If a device does not have resources to launch a route discovery, it reverts to tree routing, which uses a trivial routing algorithm. The flow chart is shown in Fig. 3.3.

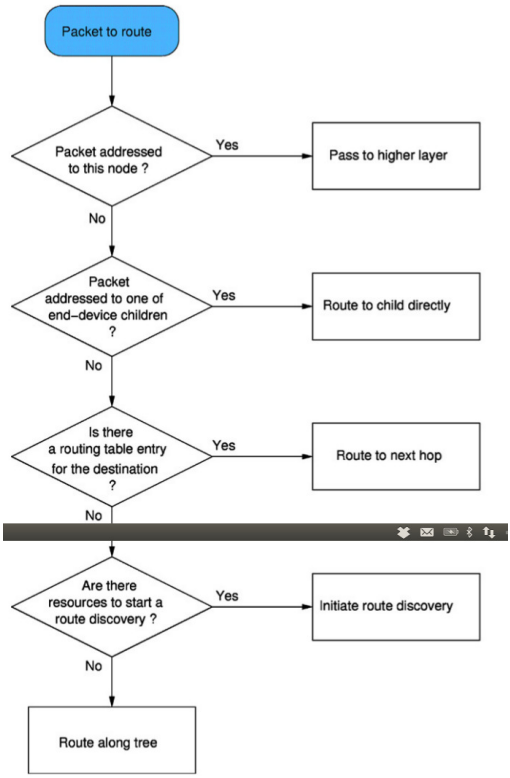

Figure 3.3: Routing a packet in ZigBee (picture from [4])

3.2 The XBee module hardware configuration

XBee modules come in different configurations. The one we will be using is called XBee Series 2 with wire antenna as it is shown in Figure 3.4.

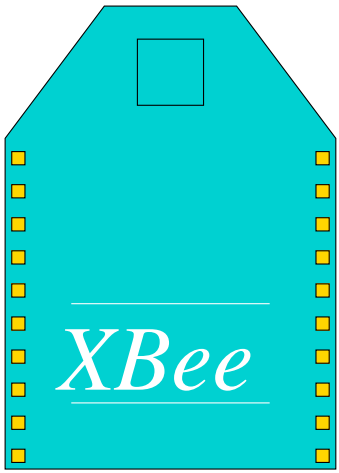

Figure 3.4: XBee Series 2 with wire antenna

This device supports different kinds of ZigBee in mesh networking. Its wire antenna provides omnidirectional coverage, or what is the same as saying that its coverage is pretty much the same in all directions when the antenna is straight and perpendicular to the module.

If you flip the XBee, you will be able to see the pins through which it can send/receive data to/from sensors, communicate with Arduino, connection to a power supply and GND (more information about the pins can be found in page 15 of [6]).

Preparing the XBee for configuration

We can access and program the XBee through any terminal application and a USB connection. The *breakout board* shown in Figure 3.5 allows us to: 1) plug the XBee into a breadboard, facilitating the wired connections with other components (including the Arduino); as well as the ability to 2) establish a USB connection to configure the XBee.

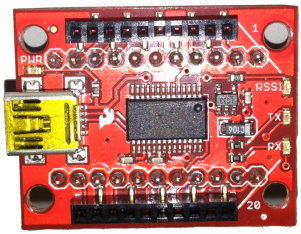

Figure 3.5: XBee Explorer board from SparkFun

As the pins on the XBee are separated differently than the holes in the breadboard, every time a configuration or wired connection is needed, the XBee should be placed in the breakout board as shown in Figure 3.6, and then placed on the breadboard.

It is important to notice that once the XBee is placed on the breakout board, the pins functions change. The new role of each pin is now the displayed underneath the breakout board, as in Figure 3.7.

Now that the device is properly placed, we need to setup a connection so the current configuration can be reviewed and changed.

Accessing the firmware

The XBee has a microcontroller running a configurable firmware. This firmware holds the necessary information for addressing,

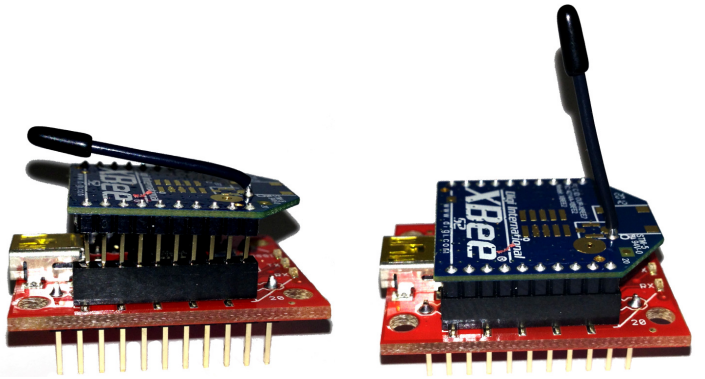

Figure 3.6: XBee and breakout board: Left: XBee outside; see the different spacing of the pings. Right: XBee inside; setup for configuring and pluging into breadboard.

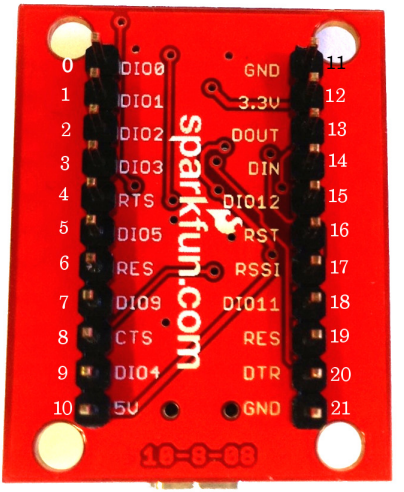

Figure 3.7: XBee Explorer pins

communication, security and utility functions. You can configure this firmware to change different settings like: local address, security settings, destination address and how the analog sensors connected to its pins are read.

As for now, the official way to update this firmware is through a program called *X-CTU* and can be downloaded for free from the *XBee manufacturer's website*.

X-CTU is only available for the Microsoft Windows operating system, nevertheless you have the virtualization option in OS X, as well as WINE Windows emulator in Linux.

To take a peek at the current XBee configuration:

1. Plug it into one of your computer's USB ports and launch the X-CTU application.

2. Select the appropriate Com Port listed under *Select Com Port*. This port should be the same in which your XBee is connected.

3. Confirm that everything is setup correctly by clicking on the *Test/Query* button. If everything is alright, a pop-up window will display the modem type and firmware version.

4. Change to the *Modem Configuration* tab on the top the X-CTU window. This tab will show you how the firmware is configured.

5. Under *Modem Parameters and Firmware*, click on the *Read* button. This will fill the current window with the current firmware configuration, as well as the XBee's *Modem* and *Function Set*.

Luckily, X-CTU is only required for upgrading the firmware. For changing the XBee's configuration we only need a USB

connection and a terminal software. Nevertheless, this is only possible if the XBee is on *AT command mode* (capable of receiving human commands and forward messages without performing any modification, see Table 3.1). To set it on, follow these steps:

1. In the *Model Configuration* tab of the X-CTU, check that the *Modem type* is set to `XB24-ZB`.

2. To start, we are going to configure two XBee radios. Under *Function Set*, choose `ZIGBEE COORDINATOR AT`.

3. Choose any version greater than `0x2070`.

4. Click on the *Write* button to program this device as a coordinator.

Once the installation is complete, gently remove the USB from the first XBee radio a plug it into another. Repeat the process described above, but now under *Function Set* choose `ZIGBEE ROUTER AT`. Select the highest version available and click on *Write* to program the device.

It is important to distinguish between the two XBee you just configured, given that they behave differently. Every ZigBee network must contain only one coordinator radio, this way the network can be properly defined and managed. Mark which configuration each radio has with a sticker to eliminate any confusion.

Table 3.1: XBee AT modes

Transparent mode	**Command mode**
Talk *through* the XBee	Talk *to* the XBee itself
Any data can me sent through	Only responds to AT commands
Default state	+++ to enter mode
Wait 10 seconds to return to this mode	Times out after 10 seconds of no input

Configuring the XBee through a terminal

There are a lot of terminal applications. Fortunately, most of them need the same kind of information to establish a connection through USB. Table 3.2 gathers the required settings for a serial terminal software attempting to establish a connection with the XBee.

Table 3.2: Default terminal settings for establishing a connection with an XBee

Setting	Value
Baud	9600
Data	8 bit
Parity	None
Stop bits	1
Flow control	None
Line feed	CR+LF or Auto Line Feed
Local echo	on

To check if you are already inside the XBee, try asking the radio to go to command mode issuing the `+++` instruction. If after a moment an `OK` appears at the right hand side, then you are in!

Reviewing some AT commands for the XBee

Issuing commands from a serial connection (like the one you established with the terminal program) to the XBee follows a simple guideline: *instruction parameter* `<CR>`. Where `<CR>` accounts for *carriage return*, and just means that you have to press the Return (Enter) key to submit the command. Passing and empty *parameter* just outputs the current value of the specified register (the *instruction* part of the command).

All AT commands start with the `AT` prefix (accounting for *attention*) and then are followed by a two letter character command identification. Some of the basic AT commands are described below, as well as in [6].

- `AT`: gets the attention of the XBee. Its normal output is `OK`. If you do not receive this output, you've probably timed out of command mode and need to reissue the `+++` command to get back in it.

- `ATID`: without any parameter it shows the current Personal Area Network ID (PAN ID) that is assigned to the radio. You can set a PAN ID passing an hexadecimal number in the range 0x0-0xFFFF as a parameter.

- `ATSH/ATSL`: it shows the *high* or *low* parts of the unique XBee 64-bit serial number, respectively. This number cannot be changed, so passing a parameter will produce an `ERROR` response.

- `ATDH/ATDL`: it shows the *high* or *low* parts of the destination address the local radio will forward messages to, respectively. Putting address information after `ATDH` or `ATDL` will set the *high* or *low* parts of the destination address, accordingly.

- `ATWR`: saves the current configuration to firmware, so it will become the default configuration the next time you power on the XBee.

Lab Practices

The following chapters gather all the labs hands-on practices that will guide through the process of setting schematics on your Arduino, configure a simple WSN, collect sensor data and upload it to a repository for future use.

Each practice suggests a chapter to read before attempting it. This way you will feel more familiarised with the terms used.

Code on! `p(^-^q)`

Chapter 4

Practice: Installing the Arduino IDE

Suggested read: Chaper 2

In the following practice, you will spend some time getting to know the Arduino platform, its connections and how to interact with it through a PC.

4.1 Reviewing the hardware

As you were able to see in Figure 2.2, the Arduino board contains a whole computer on a small chip, although it is at least a thousand times less powerful.

Taking a closer look at Figure 2.2, you will be able to see *14 Digital IO pins (pins 0-13), 6 Analogue IN pins (0-5) and 6 Analogue OUT pins (pins 3, 5, 6, 9, 10, and 11).*

The *Digital IO* pins, as the name suggests can be set to input or output. Their function is specified by the sketch you create in the IDE (more on IDE in Section 4.2). The *Analogue*

IN ports take analogue values (i.e., voltage readings from a sensor) and convert them into a number between 0 and 1023. As for the *Analogue OUT* ports, are actually digital pins that can be reprogrammed for analogue output using the sketch you can create in the IDE.

4.2 The Arduino IDE

The Arduino Integrated Development Environment (IDE) is the responsible for making your code work in the Arduino board. Without entering in much unnecessary detail, what the IDE does is to translate your code into C language and compile it using `avr-gcc`, which makes it understandable to the micro-controller. This last step hides away as much as possible the complexities of programming micro-controllers, so you can spend more time thinking on your actual code.

You can download the Arduino IDE *from here*. If you are using *Linux* or *Windows* operating systems, just double click the downloaded file. This will open a folder named *arduino-[version]*, such as *arduino-1.0*. Place the folder wherever you want in your system. For Ubuntu users, a good alternative is to use the *Ubuntu Software Center* when available. On the Mac, just double click the downloaded file, this will open a disk image containing the Arduino application. Drag a drop the application icon to your Applications folder.

Do not open your installed application yet. First you must teach your computer to detect the Arduino hardware through the USB ports.

4.2.1 Configuring the USB ports for detecting the Arduino

In Linux and OS X, the USB controllers are the same used by the operating system.

On the Mac, plug the Arduino into an USB port.

The PWR light on the board should come on. Also, the LED labelled "L" should start blinking.

Then, a pop-up window telling you that a new network interface was found should appear. Proceed clicking "Network Preferences...", and then "Apply". Although it may appear with a status of "Not Configured", the Arduino is ready for work.

Windows machines, plug your Arduino and the "Found new Hardware Wizard" will appear. After the wizard tries to find the driver on the Internet, you will be able to select "Install from a list or specific location" button. Choose it and click next. You will be able to find the drivers under the "Drivers" folder of the Arduino Software download.

Once the drivers are installed, you can launch the IDE and start using Arduino.

4.2.2 Identifying the port connected to the Arduino

In the case of the **Mac**, once in the Arduino IDE, select "Serial Port" from the "Tools" menu. Select `/dev/cu/.usbmodem`; this is the name that your computer uses to refer to the Arduino board.

For **Windows**, under the operating system "Start" menu open the "Device Manager" by right-clicking on "Computer" (Vista) or "My Computer" (XP), then choose properties. On

XP, click "Hardware" and choose Device Manager. On Vista, click "Device Manager".

Look for the Arduino device in the list under "Ports (COM & LPT)". Your device name will be followed by a port number, usually "COM#", where # refers to a number.

Once you have identified the COM port number for the Arduino connection, you can select that port from the Tools > Serial Port menu in the Arduino IDE.

Now the Arduino IDE can talk with the Arduino board and program it.

4.2.3 What's the deal with Linux users?

As mentioned before, IDE uses the same USB controllers than Linux. So, in order to effectively detect your Arduino in Linux, simply connect it to your PC, open a Terminal a type `ls /dev/tty*`. This will display all available ports. Your Arduino serial port will probably be something like `/dev/ttyUSB0` or `/dev/ttyACM0`, but you can be sure by typing `dmesg` in the Terminal and looking at the line that details the last connected USB device to a determined `/dev/tty*` port.

Chapter 5

Practice: Blinking LED

Suggested read: Chapers 2 and 4

In the following practice you will write your first Arduino application. Although simple, mastering it will provide you with clear understanding of the IDE and the components that conform the Arduino platform.

It consist of a simple code that will turn on/off LED(s) plugged to the digital IO ports of the Arduino.

5.1 Preparing your development environment

For Practice 5, you will need:

- as many LEDs as you want, but always less than the number of digital IO ports.
- a USB cable to connect your Arduino board to the PC.
- the Arduino IDE, up and running.

Turn your Arduino on by plugging it to the PC. Make sure you have selected the appropriate COM port, as it is explained in Practice 4 according with your operating system.

5.2 The code

Once inside, enter the following code:

```
1  const int LED = 13;
2
3  void setup()
4  {
5      pinMode(LED,OUTPUT);
6  }
7
8  void loop()
9  {
10     digitalWrite(LED, HIGH);
11     delay(1000);
12     digitalWrite(LED, LOW);
13     delay(1000);
14 }
```

Listing 5.1: Blinking LED example code

As you might be able to see, the code is completely readable. Let's review it line by line.

- Line 1: `const int LED = 13`, assigns the value 13 to a `int`erger variable, named LED. In this case, this number corresponds to the digital IO port #13.
- Line 3: `void setup()` is the name of the next block of code. It is very similar to functions in languages like C/C++ and it is generally used to assign variables to ports, as well as their role.
- Line 5: `pinMode(LED,OUTPUT)` tells the Arduino how to set the pins. In this case, pin LED (#13) is set up as

an OUTPUT. `pinMode` is a function, and the words or numbers inside the parenthesis are its arguments.

- Line 8: `void loop()`: is where you define the behaviour of your device. The statements contained in `loop()` are repeated over and over again until the device is turned off.

- Line 10: `digitalWrite(LED, HIGH)` works as a power socket for pins. In this case, the command is indicating to turn pin `LED` into `HIGH`, which instructs Arduino to turn the output pin to 5V. If you have connected a LED in this pin, the result is that it will turn on (hopefully). Turning on and off the pin allow us to see what the software is making the hardware do; the LED is an actuator.

- Line 11: `delay(1000)` tells the processor to wait for 1000 *milliseconds* before proceeding to the next code line.

- Line 12: `digitalWrite(LED, LOW)` as with Line 10, this function turns pin `LED` to 0V, causing the connected LED to turn off. You can do a mental map in which $HIGH \rightarrow ON$, $LOW \rightarrow OFF$.

- Line 13: because the last instruction was to set the LED off, this will keep it that way for an additional 1000 *millisecods*.

To see your work, just insert the longer leg of the LED into the digital IO port you assigned to variable `LED` on your code (digital pin 13), and the shorter leg to ground (GND). Figure 5.1 shows the desired layout.

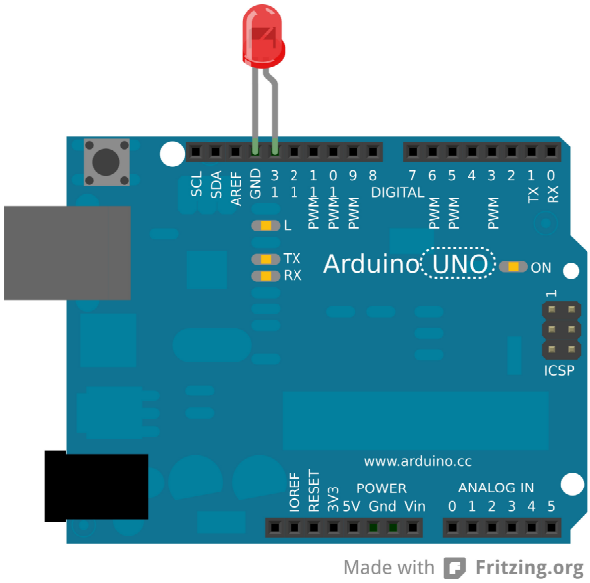

Figure 5.1: Blinking LED layout

Chapter 6

Practice: Blinking LED Advanced

Suggested read: Chapers 2 and 5

It will be very boring to just have a blinking LED. That is because in this practice we will be incorporating some hardware and software tweaks that will allow us to have a little more control over the LED. Or let's say, we will make a basic lamp.

What we want to prototype is a LED that turns on or off whenever we press a bottom. Before we dwell into detail, let's review what we will need:

- A breadboard (we will be using Figure 6.1 as a guide).
- Wire to tie together the different parts of your circuit.
- One 10K Ohm resistor.
- One pushbutton switch.

Breadboards will help us to build circuits. It allows for effective connection between components without worrying about

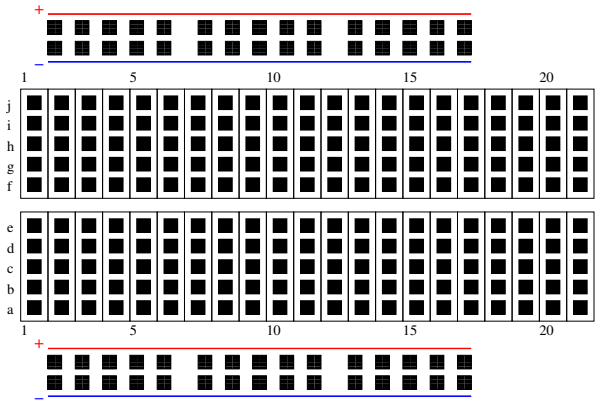

Figure 6.1: Breadboard

the electrical subtleties or hazards. Taking Figure 6.1 as a reference, the breadboard has internal electrical connections that makes it possible to tie multiple components to a single point. It does so by representing a *physical* connection as multiple rows of the same column. That is: holes 1a and 1d are physically connected inside the breadboard's circuitry, whereas 3d and 4a are not.

Each breadboard is divided by thick spaces among different sections. In Figure 6.1, there are four distinct sections: two with the + and − symbols, and two with numbers and letters. The latter was described above, whereas the former works in the opposite way: holes are connected with other holes in the same row. This section is often used to power the circuit, but more on that further in the practice.

Before writing any code, try to assemble the parts as shown in Figure 6.2.

To avoid any confusion, let's review the layout component by component:

1. Place the pushbutton on your breadboard. In Figure 6.2, the two pushbutton "legs" are inserted into holes 5c and

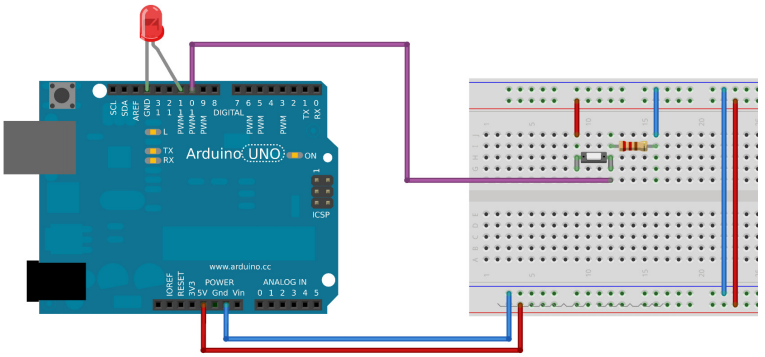

Figure 6.2: Blinking LED advanced layout

6c. In this example, the pushbutton will be energised through the 5c leg.

2. Connect one of the legs of the resistor to the negative leg of the pushbutton (hole 6b). This will physically connect the resistor to one of the legs of the pushbutton. Insert the other leg on hole 9b.

3. In order to avoid confusion, cables on all figures are color coded. *Red* represents power cables, *blue* are connections to GND and *cyan* are connections to IO pins on the Arduino. Try to duplicate the layout of Figure 6.2.

Note that connecting the + row to the Arduino's 5V pin will provide 5V to all the + row. The same is true for GND and the − row. This is very useful to avoid running out of 5V of GND pins.

6.1 The code

Type the following instructions as a new file in the Arduino IDE:

```
1  // Turns on LED when pushbuttom is pressed and
2  // turns it off when pressed again.
3
4  const int LED = 11;
5  const int BUTTON = 10;
6  int val = 0;
7  int old_val = 0;
8  int state = 0;
9
10 void setup(){
11     pinMode(LED, OUTPUT);
12     pinMode(BUTTON, INPUT);
13 }
14
15 void loop(){
16     val = digitalRead(BUTTON);
17
18     //check if the button was pushed
19     if((val == HIGH) && (old_val == LOW)){
20         state = 1 - state;
21         delay(10);
22     }
23
24     old_val = val;
25     if(state == 1){
26         digitalWrite(LED, HIGH);
27     }else{
28         digitalWrite(LED, LOW);
29     }
30 }
```

Listing 6.1: Blinking LED advanced example code

Let's review the code:

- Line 4: sets the pin for the LED.
- Line 5: assigns the input pin where the pushbutton is connected.

- Line 6: `val` is the variable holding the state of the input pin corresponding to the pushbutton.
- Line 7: `old_val` holds `val`'s previous value.
- Line 8: the variable `state` determines de condition of the LED. 0 = off and 1 = on.
- Line 11: the function `pinMode()` sets the roll of each pin. In this case, pin `LED` is set to `OUTPUT`.
- Line 12: sets pin `BUTTON` to `INPUT`.
- Line 16: asks whether there is any power at the specified pin. It returns HIGH or LOW if the button is being pushed or not, respectively.
- Line 19: if the button is being pushed, then `val` = HIGH and `old_val` = LOW. This provokes a change in `state`.
- Line 21: prevents errors in the change of `state`. Given that `loop()` repeats several hundred thousand times per second, making the processor wait a little bit allows for a correct reading of the pushbutton.
- Line 24: the value of `val` is now old. Notice that once the LED is turned on, `val` = `old_val` = LOW. Furthermore, `val` only changes when the button is pushed.
- Line 26: turn LED on.

Chapter 7

Practice: Simple chat with XBee

Suggested read: Chapter 3

WSNs are composed of nodes able to send messages among themselves. In this practice you will be guided through the configuration of (at least) two XBees to build a basic chat application. Furthermore, you will have the opportunity to familiarize yourself with the XBee and the different AT commands described in Chapter 3.

You will need:

- One XBee Series 2 configured as a `ZIGBEE COORDINATOR AT`.
- One XBee Series 2 configured as a `ZIGBEE ROUTER AT`.
- As many breakout boards and USB cable A to mini B as XBee radios.
- One computer per XBee. It is less confusing than establishing multiple terminal sessions from one computer.

We need to be able to distinguish the coordinator radio from the router radios. It is easier if you write this addresses down. Proceed as follows:

1. Establish a terminal connection to the coordinator radio.
2. Once inside, issue the `+++` to enter to command mode.
3. Type `ATSL` to reveal the lower part of the XBee serial number.
4. Write it down: **Coordinator address:** 0013A200 _____

Repeat the same for the router AT.

Router address: 0013A200 _____

Now, let's configure the coordinator.

7.1 The code: coordinator

The settings for the coordinator are contained in Table 7.1 below.

Table 7.1: XBee coordinator settings for simple chat

Description	Command	Parameter
PAN ID	`ATID`	**2013**
Destination address *high*	`ATDH`	**0013A200**
Destination address *low*	`ATDL`	_____

Note that the *Destination address low* specified in Table 7.1 correspond to the router radio.

Issuing the commands on the terminal window will look like the listing below.

```
1 +++
2 OK
```

```
3 ATID 2013
4 OK
5 ATDH 0013A300
6 OK
7 ATDL ____ //put the lower part of the router address
8 OK
9 ATID
10 2013
11 ATDH
12 0013A300
13 ATDL
14 ____
15 ATWR
16 OK
```

Listing 7.1: Coordinator settings as seen in the terminal

You will receive an `OK` after issuing a command (as in Line 4) as well as when writing to the firmware (Line 16).

7.2 The code: router

The settings for the router must contain the same information collected for the coordinator. Fill out Table 7.2 accordingly.

Table 7.2: XBee router settings for simple chat

Description	Command	Parameter
PAN ID	`ATID`	**2013**
Destination address *high*	`ATDH`	**0013A200**
Destination address *low*	`ATDL`	_____

Note that the *Destination address low* specified in Table 7.2 correspond to the coordinator radio.

7.2.1 Chat!

Now you just have to connect each XBee to one computer and establish a terminal connection to each one (or connect

the two radios to the same computer running two different terminal applications, one for each XBee). Make sure all the connection settings are as specified in Table 3.2, so you will not have any problems.

If both radios are in transparent mode (see Table 3.1), everything you type in one terminal will be forwarded to the other XBee.

Chapter 8

Practice: Wireless doorbell

Suggested read: Chapters 3 and 7

This practice guides you through the construction of a wireless doorbell system. It is composed by two components: the switch and the buzzer.

On the switch side, we will be prototyping a layout like the one shown in Figure 8.1. While the sound will be produced by a buzzer on the other radio, like in Figure 8.2.

You will need:

- Eventhough the two components may fit in one breadboard; to make it more real, it is better to use two separate breadboards.
- Hookup wire. It is recommended to have at least four different colors.
- Two Arduino boards.

- USB A-to-B cable for the Arduinos.
- One 10KΩ resistor.
- One momentary switch or pushbutton for input.
- One buzzer for output.
- One XBee radio configured as `ZIGBEE COORDINATOR AT`.
- One XBee radio configured as `ZIGBEE ROUTER AT`.
- Two breakout boards.
- USB cable for the XBee breakout board.

Every ZigBee network has only one coordinator. Other nodes can be configured as routers. To configure your XBee radios, please refer to Chapter 3.2.

It is strongly suggested that you mark down the XBees to distinguish the coordinator from the router(s).

8.1 Wireless doorbell connections layout

8.1.1 Switch

Follow these guidelines to prepare the connections for the switch that will make the doorbell ring (or buzz in our case). If lost, you can always take a look at the final layout in Figure 8.1.

1. Energize the breadboard by hooking up a red wire from the Arduino 3.3V output to one of the power rails of the breadboard.

2. Hook up a blue wire from the ground (GND) connection on the Arduino to the ground rail on the breadboard.

3. Place the XBee/breakout board with both sides on different sections of the breadboard, in a way that the space separating the sections passes under the XBee/breakout board.

4. Use a red wire to connect pin 3.3V (or pin 1 as in Figure 3.7) of the XBee to the power rail on the breadboard.

5. Use another color cable to hookup the XBee's GND pin to the ground in the breadboard.

6. Grab another color cable to connect pin TX/DOUT of the XBee to digital pin 0 (RX) on the Arduino.

7. Then do the reverse way communication by connecting XBee's RX/DIN pin to the digital pin 1 (TX) on your Arduino.

8. With the coordinator XBee, attach the button to the digital input 2 of your Arduino, making sure to use the 10KΩ resistor as in Figure 8.1.

The other XBee (as a router) will work as the buzzer. For this, replicate the schematics shown in Figure 8.2.

8.2 The code

8.2.1 XBee code

First, start with the coordinator. It is recommended that you attach a sticker or some kind of mark to each XBee so you will know which one is the coordinator and the router.

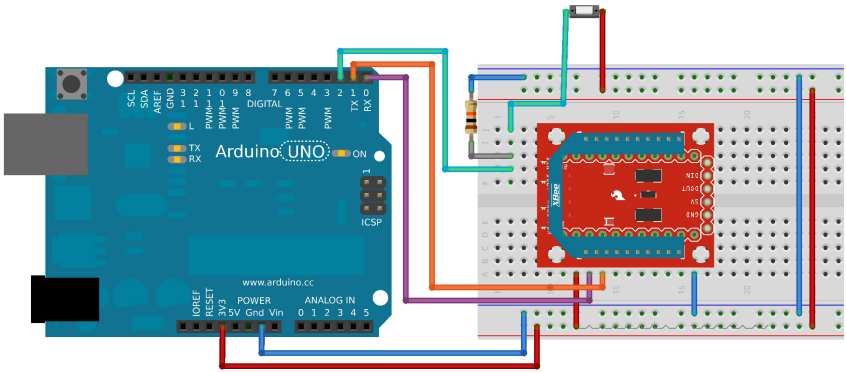

Figure 8.1: Wireless doorbell: switch layout

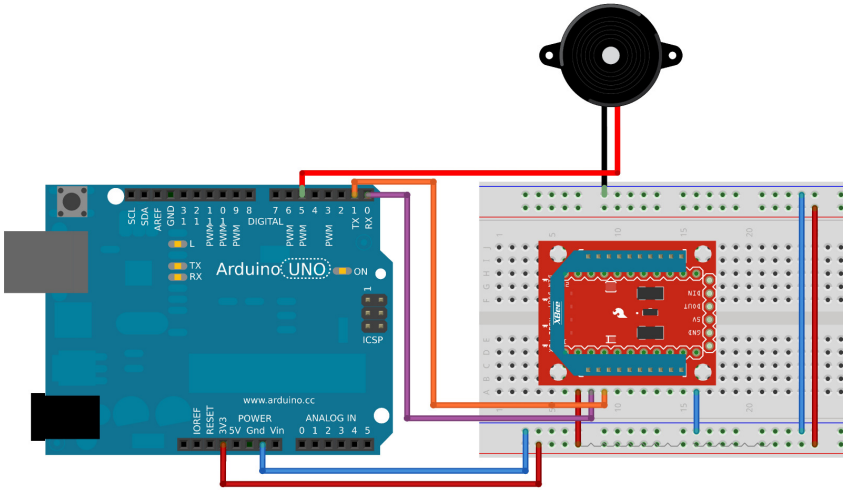

Figure 8.2: Wireless doorbell: buzzer layout

1. Start a terminal session with the XBee. You can use any terminal application or the terminal utility in the X-CTU.

2. To start interactive mode, type `+++` and the XBee will respond with an `OK` message. You're in!

3. Select a PAN ID number between 0x0 and 0x(16 Fs), so there are plenty PAN IDs. Then, configure the XBee to work in this PAN ID by entering `ATID` followed by the PAN ID you selected and press enter. The XBee should respond with an `OK` message. If not, then maybe you were thrown out of command mode. Retry by issuing the `+++` command again.

4. Because we are working with a pair of radios, this one should have as a destination address the address of the other XBee you are using. So, enter the *high* part of the other XBee destination address by typing `ATDH 0013A200`.

5. Then, enter the *low* part of the destination address by typing `ATDL` followed by your other XBee's lower address.

6. Write your configuration by issuing `ATWR` and pressing enter.

Repeat this same process with your other XBee. Remember that:

- You will be thrown out of command mode after 10 seconds of inactivity. To re-enter enter the `+++` command and wait for the `OK` response.

- The destination addresses are those of the other XBee! This is a frequent mistake.

- After making changes, always save them by entering the `ATWR` command.

8.2.2 The Arduino code

The button Arduino

This code corresponds with the button side of the example. There's a trap!: when uploading programs to your Arduino, disconnect the digital pin 0 (RX) and then reconnect it after the loading is completed. Otherwise you will receive an error.

```
int BUTTON = 2;
void setup() {
    pinMode(BUTTON, INPUT);
    Serial.begin(9600);
}
void loop() {
// send a capital D over the serial port if the
    button is pressed
    if (digitalRead(BUTTON) == HIGH) {
        Serial.print('D');
        delay(10); // prevents overwhelming the
            serial port
    }
}
```

Listing 8.1: Configuring the Arduino of the buttom side of the example.

The buzzer Arduino

This Arduino will receive a signal when the button is pressed and will ring the buzzer.

```
int BELL = 5;
void setup() {
    pinMode(BELL, OUTPUT);
    Serial.begin(9600);
}
void loop() {
    // look for a capital D over the serial port and
        ring the bell if found
```

```
8       if (Serial.available() > 0) {
9           if (Serial.read() == 'D'){
10          //ring the bell briefly
11              digitalWrite(BELL, HIGH);
12              delay(10);
13              digitalWrite(BELL, LOW);
14          }
15      }
16  }
```

Listing 8.2: Configuring the Arduino of the buzzer side of the example.

Chapter 9

Practice: A WSN with XBee's AT mode

Suggested read: Chapers 3 and 7

In this assignment we will build a sensor network using the AT (as opposed to the API) mode. This means that we will directly write characters in one end and read them at the other end. We can replace any of the two ends by a serial terminal emulation (*minicom*).

We start by flashing two XBees using X-CTU. We will install the AT firmware in them. One has to be the coordinator and the other the router.

Then we have to configure the PANID, and the destination address for each of them. The configuration can be done using XCTU or the minicom. Verify the configuration observing the RSSI LED and then connecting minicom to both XBee and sending information from one XBee to the other as in the chat assignment (see Chapter 7).

We will use one of the XBees on the solderless breadboard, connected to the Arduino. As usual, use the Arduino 3.3V to

power the breadboard and connect the serial pins of the XBee to the Arduino. Use the breadboard to install your light and temperature sensors and connect them to the analog inputs of the Arduino.

Use the example code in Listing 9.1 as a reference to read data from the sensors and send it using the serial connection from the Arduino to the XBee. Then the XBee will automatically send it to the remote XBee. Note that in the code we include identifiers for the node and sensor.

```
int ledPin = 13;

void setup()
{
  pinMode(ledPin, OUTPUT);
  Serial.begin(9600);
}

void loop()
{
  digitalWrite(ledPin, HIGH);
  delay(1000);
  digitalWrite(ledPin, LOW);
  delay(1000);
  Serial.print("N "); //node
  Serial.print("10 ");
  Serial.print("S "); //sensor
  Serial.print("1 ");
  Serial.print("T ");
  Serial.println(analogRead(A0));
}
```

Listing 9.1: Reading from analog input and sending the data via the XBee.

The other XBee will be connected directly to the computer. This will be the sink and will gather the sensed data. We will program this part in Python using the Listing 9.2 for reference.

```

```

```
# Derived from code by Alejandro Andreu
import serial
import time
import sys
import shlex

print 'Receiving data in transparent mode'

def main():
    if len(sys.argv) is 1:
        print 'You must provide at least one
            argument.'
        sys.exit()
    filename = sys.argv[1]
    try:
        with open(filename) as f: pass
    except IOError as e:
        print e, '\n'
        sys.exit()

    s = serial.Serial(filename, 9600)

    print 'opened'

    while 1:
        received = s.readline()
        print received
        splitted = shlex.split(received)
        for i in range(len(splitted)):
            print splitted[i]
        time.sleep(1)

if __name__ == "__main__":
    main()
```

Listing 9.2: Simple code that reads the message that arrive to the XBee in AT mode.

Now we can collaborate with other groups to build larger networks and do more interesting stuff. For example

- Compute time averages, geographical averages and time-

geographical averages.

- Use EWMA or other filters for time average.
- Create alarms that use a combination of the data received from different sensors and nodes. For example, if the majority of nodes report light and temperature readings, trigger a fire alarm.
- The alarm can be a LED on the Arduinos. Use broadcast addresses if you want to reach all the nodes in the PANID.

Chapter 10

Practice: Sunset Sensor

Suggested read: Chaper 9

In this lab assignment you will create a sunset detector using a light-dependant resistor (LDR). If there is plenty of light, it means than the Sun is high in the sky. During the day, the detector will light up a white LED.

In the case of complete darkness, the Sun has already gone. At night, the detector will light up a blue LED.

The sunset detector will display an alarm (either a red LED or buzzer) for intermediate lighting ranges.

The detector consists of two parts that communicate wirelessly, namely the sensor board and the processing board. The sensor board contains the LDR and an XBee that takes measures and sends them to the other part. The processing board contains an XBee to receive the data and an Arduino to process it. The processing board also contains the alarm (LED, buzzer or both).

Use a resistor in series with the LDR to obtain a range of values readable for the XBee analog input. Take a sample every 255 ms.

On the processing board, use the API firmware to be able to serially read the values that the remote XBee is sending. Check which is the received value and if it is in the range of interest (intermediate) activate the alarm (LED or buzzer).

10.1 What you need

- Breadboard (two better than one).
- Jumper wire.
- Arduino UNO (and USB-A-to-B cable)
- 2 XBee S2.
- 2 XBee explorer (and at least one USB-to-mini-USB cable)
- Three LEDs (white, red, blue)
- A buzzer (optional)
- A photoresistor (or Light Dependant Resistance). 10Kohm in the dark, 1Kohm in bright light.
- 20Kohm resistor.

Figure 10.1: LDR

10.2 Configuration

Use X-CTU to configure your XBees. One of them must be configured as a router (e.g., router AT 22A7 at the time of writing) and the other as a coordinator API (e.g., coordinator API 21A7 at the time of writing).

For the router, configure the PANID, the destination address (both high and low), enable channel verification (Networking → JV), set D0 to analog (I/O settings → D0 → ADC) and set the sampling rate to 128 ms (I/O settings → I/O sampling → IR - I/O sampling rate → FF).

For the coordinator, you just have to configure the PANID and the destination's address.

10.3 Connections

For prototyping purposes, we will place the sensing and the processing board next to the other as in Fig. 10.2, to make it possible to power both of them using the Arduino. For real deployment, you could use batteries to power the sensor board.

Connect digital outputs 10, 11 and 12 to the LEDs. Optionally you can also connect a buzzer if you are feeling noisy.

Connect also the LDR in series with a resistor that is twice as much the LDR (20 Kohm). The input to the XBee is precisely the connection between the LDR and the resistor.

Make sure that the RSSI LEDs of the explorer are on, which means that both of them are in the same network. Note that we are using the series communication pins to pass information from the XBee to the Arduino. This wires need to be removed to program the Arduino, as the programming process also uses the serial communication port.

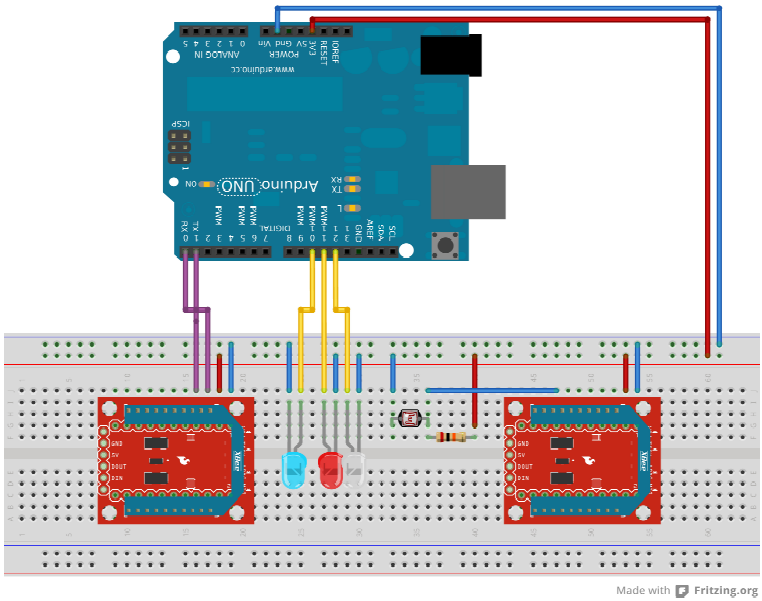

Figure 10.2: Sunset sensor connections

10.4 The code

The following code reads a word value (2 bytes) from the serial port and blinks one of the LEDs accordingly. The ranges of day, sunset and night might need to be adjusted.

There is a debug LED that you can use to troubleshoot. This LED blinks when Arduino reads data from the XBee.

```
1  // This code is derived from Robert Faludi's book
2  // "Building Wireless Sensor Networks"
3  // Check the original for further explanations
4
5  int LED_NIGHT = 10;
6  int LED_SUNSET = 11;
7  int LED_DAY = 12;
8  int debugLED = 13;
9  int analogValue = 0;
10
```

```
11 void setup() {
12     pinMode(LED_DAY,OUTPUT);
13     pinMode(LED_SUNSET,OUTPUT);
14     pinMode(LED_NIGHT,OUTPUT);
15     pinMode(debugLED,OUTPUT);
16     Serial.begin(9600);
17 }
18
19 void loop() {
20     digitalWrite(LED_NIGHT, LOW);
21     digitalWrite(LED_SUNSET, LOW);
22     digitalWrite(LED_DAY, LOW);
23     // make sure everything we need is in the buffer
24     if (Serial.available() >= 21) {
25         // look for the start byte
26         if (Serial.read() == 0x7E) {
27             //blink debug LED to indicate when data
                   is received
28             digitalWrite(debugLED, HIGH);
29             delay(10);
30             digitalWrite(debugLED, LOW);
31             // read the variables that we're not
                   using out of the buffer
32             for (int i = 0; i<18; i++) {
33                 byte discard = Serial.read();
34             }
35             int analogHigh = Serial.read();
36             int analogLow = Serial.read();
37             analogValue = analogLow + (analogHigh *
                   256);
38         }
39     }
40
41     if (analogValue > 0 && analogValue <= 350) {
42         digitalWrite(LED_NIGHT, HIGH);
43         delay(10);
44         digitalWrite(LED_NIGHT, LOW);
45     }
46
47     if (analogValue > 350 && analogValue <= 750) {
48         digitalWrite(LED_SUNSET, HIGH);
49         delay(10);
```

```
50            digitalWrite(LED_SUNSET, LOW);
51        }
52
53        if (analogValue > 750 && analogValue <= 1023) {
54            digitalWrite(LED_DAY, HIGH);
55            delay(10);
56            digitalWrite(LED_DAY, LOW);
57          }
58    }
```

Listing 10.1: Sunset sensor

10.5 Advanced optional assignment

If you are willing to do more complicated stuff, try to move the alarm to the sensor board. Now the sensor board receives the data, sends it to the processing board for processing, and waits for an instruction from the processing board to ring or light the alarm.

Chapter 11

Practice: Blink a LED on the XBee from a computer

In this assignment we will only use two XBees, a protoboard and a LED. We will not use the arduino.

Flash the two XBees. The "local" one will be the coordinator and the "remote" one, a router. Use the API mode for the coordinator. For the remote one, you can use either coordinator, router or end device. As we are going to interact with the local XBee using the python library, it is necessary to set the API mode to two (AP=2).

Connect the remote XBee to a protoboard and power it through the USB. Connect the 5V to the supply voltage bus running along the protoboard. Finally, connect a LED to bus (long leg) and to the D1 (short leg). We can blink the remote link by changing the state of the D1 pin from a python program as in the example code 11.1.

```
1
2 #! /usr/bin/python
3
```

```
4  from xbee import XBee
5  import serial
6  import time
7
8  ser = serial.Serial('/dev/ttyUSB0', 9600)
9  xbee = XBee(ser)
10
11 while True:
12     try:
13         xbee.send('remote_at',
14                   frame_id='A',
15                   dest_addr_long='\x00\x00\x00\x00\
                         x00\x00\xFF\xFF',
16                   dest_addr='\xFF\xFE',
17                   options='\x02',
18                   command='D1',
19                   parameter='\x05')
20
21         time.sleep(1)
22
23         xbee.send('remote_at',
24                   frame_id='A',
25                   dest_addr_long='\x00\x00\x00\x00\
                         x00\x00\xFF\xFF',
26                   dest_addr='\xFF\xFE',
27                   options='\x02',
28                   command='D1',
29                   parameter='\x04')
30
31         time.sleep(1)
32     except KeyboardInterrupt:
33         break
34
35 xbee.send('remote_at',
36           frame_id='A',
37           dest_addr_long='\x00\x00\x00\x00\x00\x00\
                xFF\xFF',
38           dest_addr='\xFF\xFE',
39           options='\x02',
40           command='D1',
41           parameter='\x05')
42
```

```
43 ser.close()
```

Listing 11.1: This example code alternatively changes a remote pin to up and low to blink an LED.

11.1 Next steps

In the example code we use broadcast packets. Try to blink LEDs in three different XBees alternatively. During the first second, the first LED is on. For the second second, the second LED is on. And the third LED is on for the third second. To achieve this you will need to use targeted packets with an explicit destination.

We can also implement the "sunset sensor" lab using a computer instead of the arduino.

Chapter 12

Practice: Sensor Ping

Suggested read: Chapers 3 and 7

In this assignment we will implement a "ping" over the sensor network. We will use a script in Python to send a probe packet to a remote XBee. Time is measured between the submission of the packet and the receipt of the acknowledgement and we present the information on the screen to the user. To makes things more interesting, we connect an Arduino to the remote XBee that flashes a LED each time it receives a packet. The number of times that the LED has to be flashed is included in the probe packet.

We will start by using X-CTU to flash one of the XBees as a coordinator API and the other as router API. To prevent interference with other groups, write the PAN ID that you are going to use on the blackboard and make sure that you use a PAN ID different from other groups. The other XBee should be configured as a Router with the same PAN ID. It is important to set the API mode to 2 as shown in Fig. 12.1 for the two XBees. Note that in the computer we are using the "xbee" library.

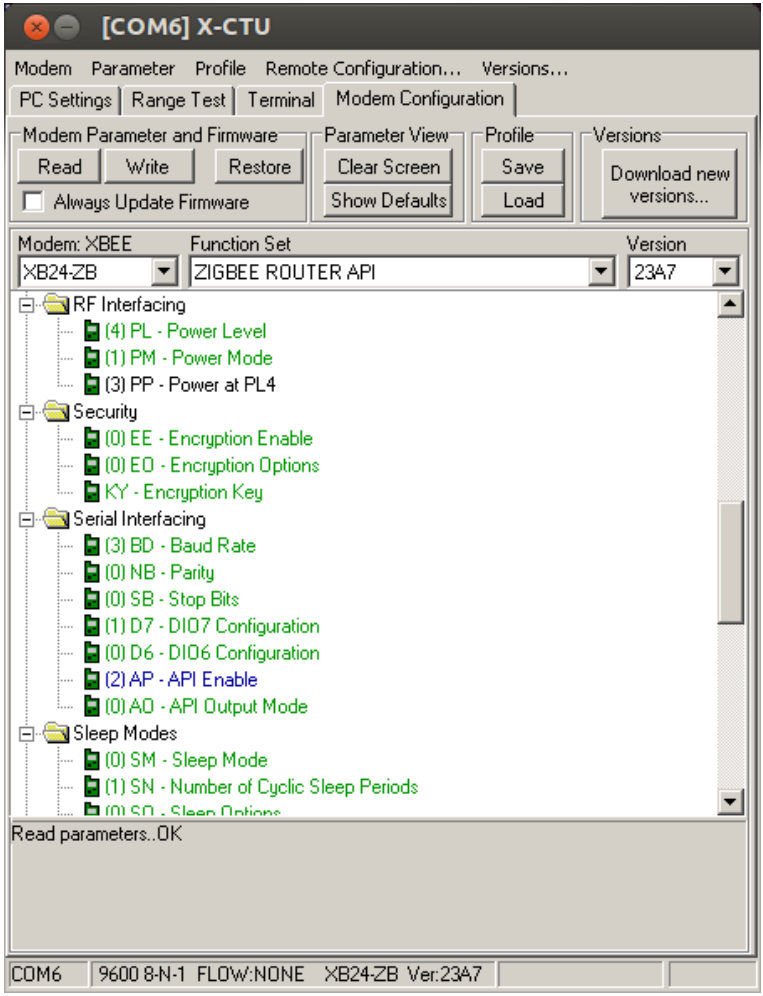

Figure 12.1: Setting the API mode to 2 (AP=2)

In the computer, we will create a program that sends a ping packet to the destination every 10s (or whatever you prefer). Replace the long destination address by the one of your XBee. Not that the program in Listing 12.1 sends an initial probe packet to obtain the 16-bit address of the destination and then it uses this address for subsequent probes. The time that elapses from the sending of the probe to the reception of the acknowledgement is the round-trip-time and is printed on the screen.

```
1
2  #! /usr/bin/python
3
4  import time
5  from datetime import datetime
6  import serial
7  from xbee import XBee, ZigBee
8
9  PORT = '/dev/ttyUSB0'
10 BAUD_RATE = 9600
11
12 # Open serial port and enable flow control
13 ser = serial.Serial(PORT, BAUD_RATE, bytesize=8,
       parity='N', stopbits=1, timeout=None, xonxoff=1,
       rtscts=1, dsrdtr=1)
14
15 # Create API object
16 xbee = ZigBee(ser,escaped=True)
17
18 DEST_ADDR_LONG = "\x00\x13\xA2\x00\x40\x8B\x3D\x91"
19
20 #part to discovery shot 16-bit address
21 xbee.send("tx",data="\x01",dest_addr_long=
       DEST_ADDR_LONG,dest_addr="\xff\xfe")
22 response = xbee.wait_read_frame()
23 shot_addr = response["dest_addr"]
24
25 # Continuously read and print packets
26 while True:
27     try:
28         print "send data"
```

```
29            tstart = datetime.now()
30            xbee.send("tx",data="\x03",dest_addr_long=
                  DEST_ADDR_LONG,dest_addr=shot_addr)
31            response = xbee.wait_read_frame()
32            tend = datetime.now()
33            print tend - tstart
34            time.sleep(10)
35        except KeyboardInterrupt:
36            break
37
38    ser.close()
```

Listing 12.1: Simple code that reads the message that arrives to the XBee.

The provided code should work as soon as you connect the two XBees. You can plug the USB cables to power the two XBees and check that everything is working. Observe the terminal on the computer with the measured delay and the RSSI indicator in the Explorer that flashes when data is transmitted. Try to disconnect the remote XBee and observe what happens.

To make things more interesting, we will connect an Arduino to the remote XBee and flash a light each time it receives a packet. We will use the "xbee-arduino" library in the Arduino as explained in Listing 12.2. Each time that we receive a packet we read the first byte and flash the LED as many times as indicated in that byte.

```
1  #include <XBee.h>
2
3  int ledPin = 13;
4
5  XBee xbee = XBee();
6  XBeeResponse response = XBeeResponse();
7  // create reusable response objects for responses we
       expect to handle
8  ZBRxResponse rx = ZBRxResponse();
9
10 void flashLed(int pin, int times, int wait) {
```

```
11
12      for (int i = 0; i < times; i++) {
13        digitalWrite(pin, HIGH);
14        delay(wait);
15        digitalWrite(pin, LOW);
16
17        if (i + 1 < times) {
18          delay(wait);
19        }
20      }
21  }
22
23  void setup() {
24    xbee.begin(9600);
25    pinMode(ledPin, OUTPUT);
26    flashLed(ledPin, 10, 50);      // sets the LED off
27  }
28
29  void loop() {
30
31
32    // 1. This will read any data that is available:
33    xbee.readPacket();
34    if (xbee.getResponse().isAvailable()) {
35      if (xbee.getResponse().getApiId() ==
          ZB_RX_RESPONSE) {
36        xbee.getResponse().getZBRxResponse(rx);
37        //flashLed(ledPin, 1, 100);     // sets the LED
              off
38        flashLed(ledPin, rx.getData(0), 100);     //
              sets the LED off
39      }
40    }
41  }
```

Listing 12.2: Sunset sensor

Program the Arduino and then connect it to the XBee. We are going to use the serial port to communicate with the XBee, and this same port is also used to program the Arduino. Don't connect the Arduino/XBee serial connection until you have

programmed the Arduino. After the Arduino is programmed you can connect the TX/RX ports to the DIN/DOUT ports of the XBee. Use the 3.3V and GND of the Arduino to power the XBee.

Test that everything is working and the Arduino flashes the number of times indicated by the computer.

12.1 Next steps

You can perform additional experiments building on the basic examples provided here. For example:

- Collaborate with another group to make ping measurements in a multi-hop setting.
- Use a different frame identifier for each ping and compute packet loss.
- Use broadcast messages to flash LEDs in all the Arduinos of the network.
- Set a LED on a remote breadboard when the RTT is above a threshold.

Chapter 13

Practice: Collecting data in a computer

Suggested read: Chapters 9 and 10

In this assignment we will use some Python libraries to receive the data transmitted by the sunset sensor in a computer instead of the Arduino. You can use one of the university computers, a laptop or a RaspBerry.

Install the XBee Python libraries.

`http://pypi.python.org/pypi/XBee/2.0.0`

This code offers an implementation of the XBee serial communication API.

We re-use the processing board of the previous assignment (refer to Chapter 10) and this time we will connect the XBee that receives the data to the computer using the USB cable. Remember that in the last assignment it was connected to the Arduino.

```
1
2 import serial
3 from xbee import ZigBee
```

```
4
5  print 'Printing data from remote XBee'
6
7  serial_port = serial.Serial('/dev/ttyUSB0', 9600)
8  zigbee = ZigBee(serial_port)
9
10 while True:
11     try:
12         print zigbee.wait_read_frame()
13     except KeyboardInterrupt:
14         break
15
16 serial_port.close()
```

Listing 13.1: Simple code that reads the message that arrive to the XBee.

You can see the results of running the program in Fig. 13.1.

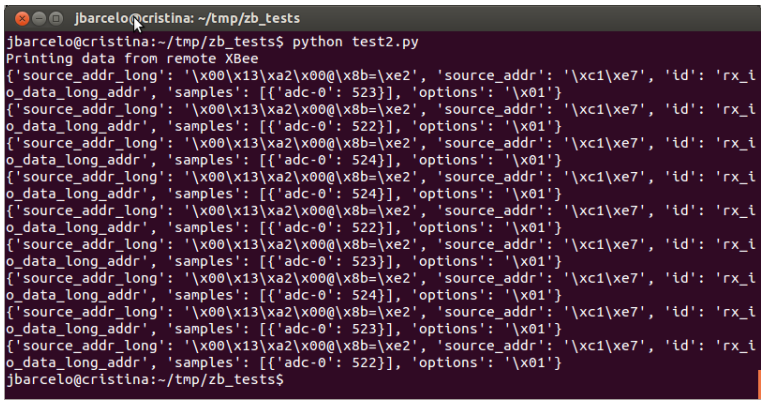

Figure 13.1: A test run of a Python program to read the messages that arrive to the XBee.

If the processing of each incoming packet takes a long time, the processing must be made asynchronous so that newer packets can be also processed in parallel. An example of long processing time can be uploading the data to the Internet.

We will define a (callback) function that is called whenever a packet is arrived. See the Listing 13.2 for example code.

```
1
2 import serial
3 import time
4 from xbee import ZigBee
5
6 print 'Asynchronously printing data from remote XBee
    '
7
8 serial_port = serial.Serial('/dev/ttyUSB0', 9600)
9
10 def print_data(data):
11     """
12     This method is called whenever data is received.
13     Its only argument is the data within the frame.
14     """
15     print data['samples']
16
17 zigbee = ZigBee(serial_port, callback = print_data)
18
19 while True:
20     try:
21         time.sleep(0.001)
22     except KeyboardInterrupt:
23         break
24
25 zigbee.halt();
26 serial_port.close()
```

Listing 13.2: Simple code that asynchronously reads the message that arrive to the XBee.

You can change the configuration of your router from the coordinator. The code snippet in Listing 13.3.

```
1
2 zigbee.send('remote_at',
3           frame_id='A',
4           dest_addr_long='\x00\x13\xa2\x00\x40\x8b\
               x3d\xe2',
```

```
5                 options='\x02',
6                 command='IR',
7                 parameter='\xF2')
```

Listing 13.3: This example remotely sets the configuration of the XBee, in particular the sample rate.

Collaborate with other groups to create larger networks and explore what you can do with the data using Python programs (computing averages? sending an email upon a trigger event? dynamically adapting the sampling rate as a function of the measured vales?).

Chapter 14

Practice: Sleep

Flash an XBee with the End-Device AT firmware using X-CTU. First, plug the XBee in the XBee explorer socket and connect the USB cable. Use the `dmesg` command to find to which device is the USB attached. Look for a line similar to

```
[ 6370.421000] usb 3-2.2:  FTDI USB Serial Device
converter now attached to ttyUSB0
```

The command to invoke X-CTU will be something similar to

```
wine .wine/drive_c/Program\ Files/Digi/XCTU/X-CTU.exe
```

Then set the port as in Fig. 14.1 and test that you can connect to the XBee using the "Test/Query" button as in Fig. 14.2.

Now we switch to the "Modem Configuration" tag. We will flash the XBee with an "end device" firmware. Only end devices can sleep. Routers and coordinators must be always up. We choose, for example, the firmware "zigbee end device at" in the "Function set" drop down menu. And we write the firmware to the XBee.

Let's also clean all previous configuration. Click the "restore" and "read" buttons to restore settings to defaults. You

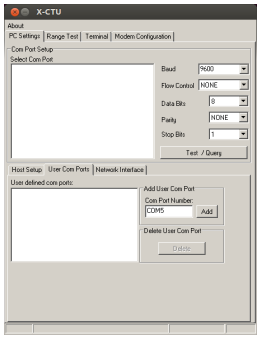

Figure 14.1: Setting the port.

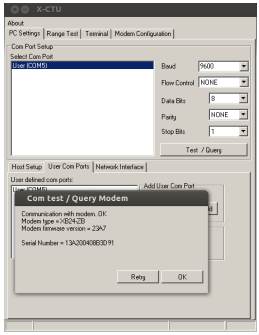

Figure 14.2: Testing communication with the XBee.

should obtain something similar to Fig. 14.3.

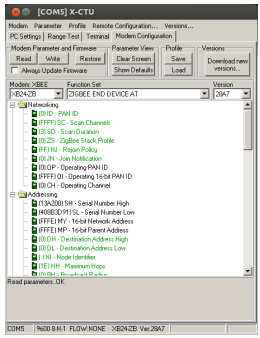

Figure 14.3: Clean 28a7

Let's prepare the new configuration. Set the PAN ID to your group ID. Set D0 to analog (2). Set the sampling rate to 255 ms (IR=1FF). We write the changes to the XBee. The configuration should look as shown in Fig. 14.4

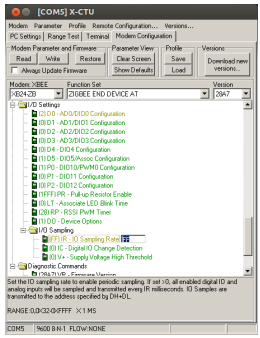

Figure 14.4: Configured end device. D0 set to 2 and IR set to 1FF.

Now we need to configure the coordinator just as we configured it for Chapter 13. We write a coordinator API firmware and restore to factory defaults.

We set the PAN ID. And we write the configuration.

Now run the python program to read incoming data, and we will see as our screens fills with the received data, as in Fig. 14.5.

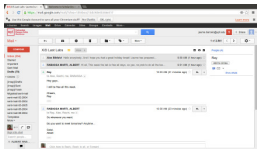

Figure 14.5: Data received by the XBee connected to the computer.

We have not sleep much so far. It's time for a nap. In the end device, change the ATSM to 5. This gives us the possibility of waking up the XBee using the DTR pin. Now we change the SP to 200 (default is 20) and we will see that the XBee takes a nap, then sends a few samples, and then takes a nap again. Let's set the SP to AAA so that the naps are longer. You will observe that the naps are fairly long, so we can use the DTR pin to wake the XBee. We can also wake up the XBee for a round of measures by changing the DTR pin from high to low.

Repeat the lab assignment "collecting data in a computer" but this time the sensors will be awake for 1 second and then sleep for 10 seconds. While the sensor is awake, it will send a sample every 100ms.

Now we will connect an LED between DIO9 and ground to see when the XBee wakes up. Other ways to know when the XBee wakes up are looking at the RSSI led or the CTS flag in the "terminal" tab in X-CTU.

Now for a really long sleeping time, set the sleep options (SO) to 4. This tells the XBee that, in case of not receiving any packets, it should multiply the sleep time by the value in SN (number of sleep periods). Set SN also to 4 and save the settings to multiply by 4 the sleep time.

14.1 Next Steps

We can try sleeping in an actuator network. In this case, we have to set the SP to the parent equal to the longest SP of all the children, to make sure that the parent stores the packets until the children wake up.

Chapter 15

Practice: Publishing data in Cosm

Create a user in *Cosm.com* . You will receive an email with a pointer to create an API-Key. Create this key as you will need it to interact with cosm.

```
1
2 curl --request POST \
3      --data '{"title":"My feed", "version":"1.0.0"}'
           \
4      --header "X-ApiKey: YOUR_API_KEY_HERE" \
5      --verbose \
6      http://api.cosm.com/v2/feeds
```

Listing 15.1: Command to create a feed

Use the API Key you obtained in the previous step. Next edit a JSON file (name it, for example, cosm.json) with the contents detailed in Listing 15.2.

Observe the output as in Fig. 15.1 and you will observe a feed id (the last number in the Location line).

Create a JSON file with some dummy data to update to Cosm as in Listing 15.2.

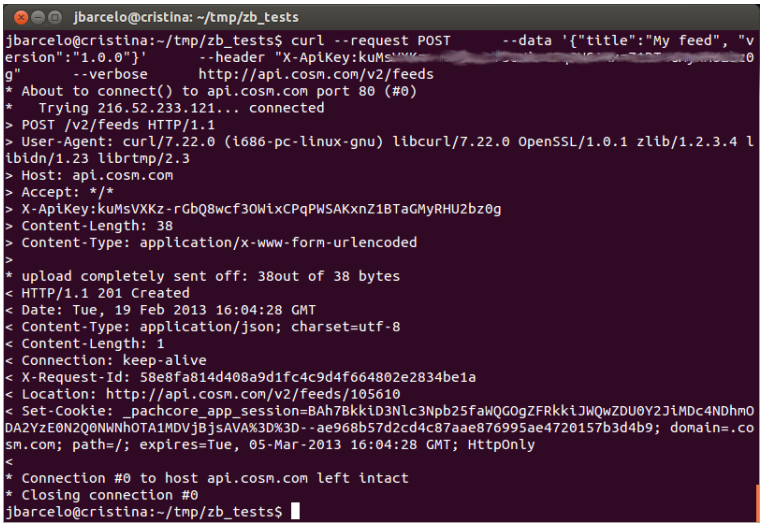

Figure 15.1: Creating a feed in COSM using curl

```
{
  "version":"1.0.0",
  "datastreams":[
      {"id":"0", "current_value":"100"},
      {"id":"two", "current_value":"500"},
      {"id":"3.0", "current_value":"300"}
  ]
}
```

Listing 15.2: JSON file with data to be uploaded to Cosm

And now you can update the data to Cosm using the curl command.

```
curl --request PUT \
    --data-binary cosm.json -header "X-ApiKey:
        YOUR-API-KEY-HERE" -verbose
        http://api.cosm.com/v2/feeds/YOUR-FEED-ID
```

Listing 15.3: Command to upload data to a feed

And if we look on Cosm's web interface, we should be able to see some beautiful plots as in Fig. 15.2

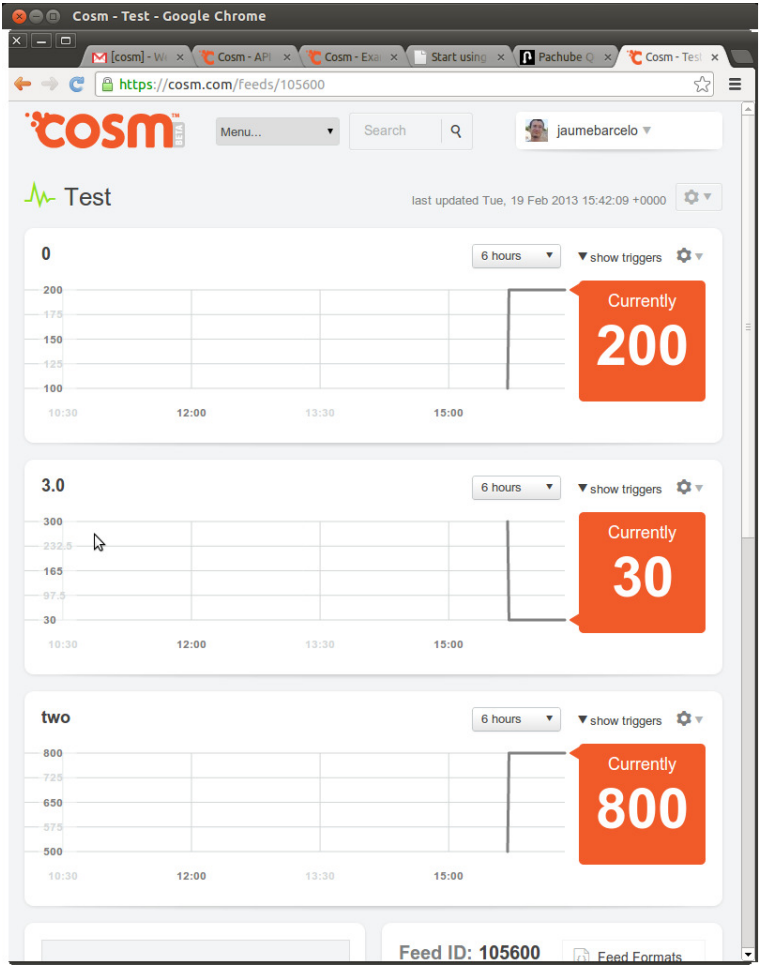

Figure 15.2: Plotting data

You can also read data from the command line as exemplified in Fig. 15.3.

The next step is to gather information from an XBee and publish it in Cosm. Use the code detailed in Listing 15.4 as a

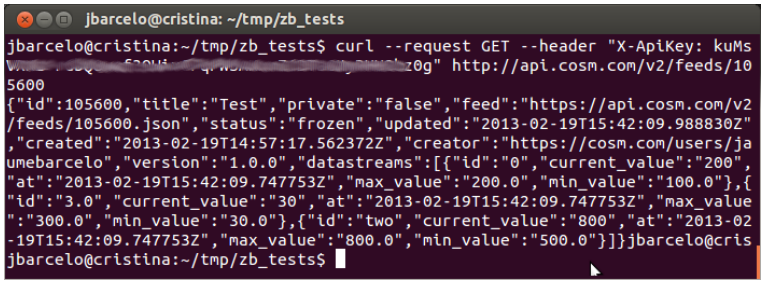

Figure 15.3: Reading a Cosm feed from the command line

reference.

```
# Derived from code by Alejandro Andreu
import commands
import json
import serial
import time
from serial import SerialException
from xbee import ZigBee

print 'Asynchronously printing data from remote XBee
    '


serial_port = serial.Serial('/dev/ttyUSB0', 9600)

def print_data(data):
    """
    This method is called whenever data is received.
    Its only argument is the data within the frame.
    """
    print data['samples'][0].keys()[0]

    # Create a JSON file and fill it with the
        received samples
    json_data = {}
    json_data['version'] = '0.2'
    json_data['datastreams'] = ()
    json_data['datastreams'] = json_data['
        datastreams'] + ({'id': data['samples'][0].
        keys()[0], 'current_value': str(data['samples
```

```
        '][0].values()[0])},)
    # Add more datastreams if needed
    with open('cosm.json', mode='w') as f:
        json.dump(json_data, f, indent = 4)
    # Upload information to COSM. Use your own Api
        Key and feed identifier
    commands.getstatusoutput('curl ... write the
        curl command here')

zigbee = ZigBee(serial_port, callback = print_data)

time.sleep(1)

zigbee.halt();
serial_port.close()
```

Listing 15.4: Command to upload data to a feed

Now you can pursue more challenging goals. For example:

- Gather information from multiple sensors in a node.
- Gather information from multiple nodes.
- Transform the measures of the temperature sensor to Celsius degrees.

Bibliography

[1] I. Akyildiz and M.C. Vuran. *Wireless sensor networks*. John Wiley & Sons, Inc., 2010.

[2] Arduino Team. Arduino homepage. http://www.arduino.cc, October 2012.

[3] Banzi, M. *Getting Started with arduino*. Make Books, 2008.

[4] P. Baronti, P. Pillai, V.W.C. Chook, S. Chessa, A. Gotta, and Y.F. Hu. Wireless sensor networks: A survey on the state of the art and the 802.15. 4 and ZigBee standards. *Computer communications*, 30(7):1655–1695, 2007.

[5] C. Thompson. Build it. Share it. Profit. Can Open Source Hardware Work? http://bit.ly/ShWD43, October 2008.

[6] R. Faludi. *Building Wireless Sensor Networks: with ZigBee, XBee, Arduino, and Processing*. O'Reilly Media, Incorporated, 2010.

[7] Á. Lédeczi, A. Nádas, P. Völgyesi, G. Balogh, B. Kusy, J. Sallai, G. Pap, S. Dóra, K. Molnár, M. Maróti, et al. Countersniper system for urban warfare. *ACM Transactions on Sensor Networks (TOSN)*, 1(2):153–177, 2005.

[8] J.A. Mazurek, J.E. Barger, M. Brinn, R.J. Mullen, D. Price, S.E. Ritter, and D. Schmitt. Boomerang mobile counter shooter detection system. In *Proceedings of SPIE*, volume 5778, page 264, 2005.

[9] J. Polastre, R. Szewczyk, A. Mainwaring, D. Culler, and J. Anderson. Analysis of wireless sensor networks for habitat monitoring. *Wireless sensor networks*, pages 399–423, 2004.

[10] Sanabria, L. Localization Procedure for Wireless Sensor Networks. Master's thesis, Universitat Pompeu Fabra, Barcelona, 2012.

[11] R. Szewczyk, E. Osterweil, J. Polastre, M. Hamilton, A. Mainwaring, and D. Estrin. Habitat monitoring with sensor networks. *Communications of the ACM*, 47(6):34–40, 2004.