eZWSN: Experimenting with Wireless Sensor Networks using the eZ430-RF2500

By:

Thomas Watteyne

eZWSN: Experimenting with Wireless Sensor Networks using the eZ430-RF2500

By:
Thomas Watteyne

Online:
< http://cnx.org/content/col10684/1.10/ >

C O N N E X I O N S

Rice University, Houston, Texas

Table of Contents

Prerequisites & Timeline[1]

IMPORTANT: Basic rules apply when using the eZ430-RF2500 boards:

- beware of static electricity, don't touch the components directly;
- never disconnect a target board from the USB programmer if still plugged into the computer;
- never disconnect a target board from the battery unit with jumper on (two leftmost pictures);
- **connect as shown on the two rightmost pictures, otherwise you destroy the board !**

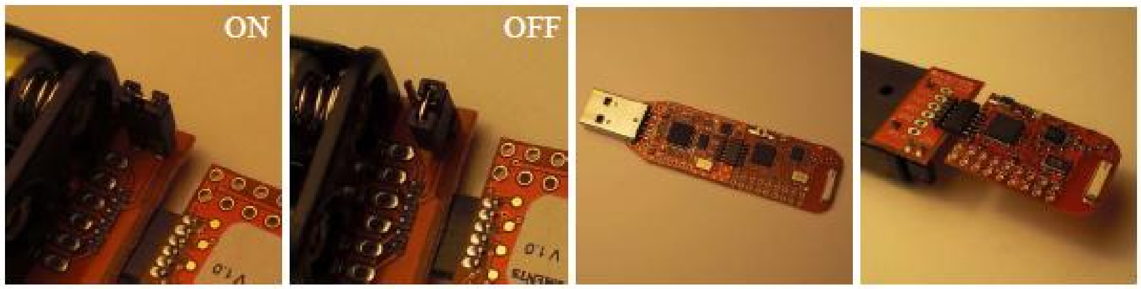


Figure 1

For this tutorial, you need:

- an MSP430 eZ430-RF2500 Development Tool kit, i.e. two target boards, a USB programmer and a battery unit [2];
- a computer running Windows, with a free USB port;
- the following freeware (available online): IAR Quick Start, SmartRF studio, PuTTY, eZ430-RF2500 Sensor Monitor Demo, cygwin (with packets GNUplot and sys), Xming for Windows [3];
- ideally, an oscilloscope like the Tektronix TDS 210 with a 1Ohm resistor mounted in an open jumper.

This tutorial is meant to be completed within 8 hours (one full day or two half days):

- Lab 1. Discover the development environment, run a basic demo (Section 1.1) (hour 1).
- Lab 2. Discover the eZ430-RF2500 board and its components (Section 2.1) (hour 2).
- Lab 3. Simple examples for the MSP430 (Section 3.1) (hours 3-4).
- Lab 4. Simple examples for the CC2500 (Section 4.1) (hour 5).

[1]This content is available online at <http://cnx.org/content/m21579/1.5/>.
[2]This set of 2 motes is manufactured and sold by Texas Instruments for $49
[3]If this software is not installed, refer to the Installation Instructions (Section)

- Lab 5. RF measurements (Section 5.1) (hours 6-7).
- Lab 6. Implementing a preamble sampling MAC protocol (Section 6.1) (hour 8).
- Lab 7. Running a Complete Wireless Sensor Network (Section 7.1) (hour 8).

NOTE: This Lab was made possible by Prof. Kristofer S.J. Pister[4] from UC Berkeley, and by the generous support of Texas Instruments[5] . The author would also like to thank the students for their constructive feedback, as well as Ferenc Kovak, Kevin Mulligan and Winthrop Williams for their logistic support.

[4]http://robotics.eecs.berkeley.edu/∼pister/
[5]http://www.ti.com/

Installation Instructions[6]

NOTE: This document details the software which needs to be installed on the host computer before taking the tutorial. Depending on the Labs you want to take, you only need to install a subset of the 5 tools described below.

IAR Kickstart

ASIDE: This is required for all labs.

IAR is the tool used to program the eZ430-RF2500 boards. The free Kickstart version is limited to compiling 4kB of code.

- Download from http://focus.ti.com/docs/toolsw/folders/print/iar-kickstart.html[7] .
- Install with default parameters.

TI Sensor Monitor Demo

ASIDE: This is required for all labs because it installs the drivers enabling your computer to talk to the mote.

Sensor Monitor is demo software which provides a graphical interface to the Texas Instruments Sensor Monitor Demo.

- Download from http://www.ti.com/litv/zip/slac139b[8] .
- Follow default installation for Demo Visualizer, Sensor Monitor Installer.exe
- Store the eZ430-RF2500 Wireless Sensor Monitor IAR Source v1.02 directory on your computer.

PuTTY

ASIDE: This is required for all labs.

PuTTY is used to read from the serial COM port created by the eZ430-RF2500 board when connected to the host computer.

- Create folder `C:/Program Files/putty/`
- Download http://the.earth.li/~sgtatham/putty/latest/x86/putty.exe[9] into that folder.

[6]This content is available online at <http://cnx.org/content/m21597/1.3/>.
[7]http://focus.ti.com/docs/toolsw/folders/print/iar-kickstart.html
[8]http://www.ti.com/litv/zip/slac139b
[9]http://the.earth.li/~sgtatham/putty/latest/x86/putty.exe

Xming

ASIDE: This is only used in Lab 5[10] .

Xming is a Linux-like X server which enables cygwin script to display windows.

- Download from http://www.straightrunning.com/XmingNotes/[11] .
- Follow default install, but choose not to install any SSH client.

Cygwin

ASIDE: This is only used in Lab 5[12] .

Cygwin is a Linux-in-Windows environment, enabling Linux-like script execu- tion on a Windows machine.

- Install from http://www.cygwin.com/setup.exe[13] .
- Follow default installation. When asked for a Download Site, choose `http://www.gtlib.gatech.edu` [14].
- When packages are listed, click on the button in the upper right corner until obtaining Full. You obtain a list of all installable packages (much like programs in Windows), marked with the word Skip when not installed, or the version number when installed.
- Using that list, add the following packages. This is done by clicking on the work Skip next to the package name.
  - **python**: An interactive object-oriented scripting language
  - **python-numpy**: Python scientific computing module
  - **gcc-g++**: C++ compiler
  - **gnuplot**: A command-line driven interactive function plotting utility
- Click next several times to finish the installation.

The Python gnuplot extension enables us to visualize data output by the mote connected to the computer. To install it:

- Open cygwin
- download gnuplot-py-1.8.zip[15]
- unzip that file and place folder `gnuplot-py-1.8/` within inside folder `C:/cygwin/`
- open cygwin
- type the following lines
  - `cd /cygdrive/c/cygwin/`
  - `python setup.py install`
- open `C:/cygwin/lib/python2.5/site-packages/Gnuplot/gp_cygwin.py` with WordPad
- change line

  ```
  default term = 'windows'
  ```

  to

  ```
  default term = 'x11'
  ```

[10]http://cnx.org/content/m21597/latest/m21598
[11]http://www.straightrunning.com/XmingNotes/
[12]http://cnx.org/content/m21597/latest/m21598
[13]http://www.cygwin.com/setup.exe
[14]This is a must as GATech's download site contains the python-numpy package we will need.
[15]http://superb-west.dl.sourceforge.net/sourceforge/gnuplot-py/gnuplot-py-1.8.zip

- change line

  ```
  gnuplot command = 'pgnuplot.exe'
  ```

  to

  ```
  gnuplot command = 'gnuplot.exe'
  ```

Start Menu Shortcuts

ASIDE: While not strictly necessary, creating Start Menu shortcuts will make your life easier.

- Create a folder `ezwsn` in the Start Menu folder (typically `C:/Documents and Settings/All Users/Start Menu/`)
- in that folder, create shortcuts to the following programs:
  - PuTTY
  - IAR Embedded Workbench
  - XLaunch *(part of Xming)*
  - cygwin
  - eZ430-RF2500 Sensor Monitor

Source Code and Resources[16]

If you take this lab

When taking the labs contained in this collection, you will be asked to create project in IAR, import drivers and copy-paste source code from these web pages. As an alternative, and for your convenience, you can download the full source code used in this Lab in IAR format:

- Full IAR source code[17]

If you are an intructor

This tutorial has been given as a lab for undergraduate and graduate students, using the following documents. You are free to use these documents for instructional use, provided you mention the original authors name.

To increase the quality of the contents, we greatly appreciate feedback and comments.

- Lab instructions[18] in pdf format. These contain the same information as this tutorial, but the students are asked to answer a few questions and to fill out the initially empty result tables.
- Exam questions[19] regarding this lab (multiple choice).

[16]This content is available online at <http://cnx.org/content/m22470/1.4/>.
[17]See the file at <http://cnx.org/content/m22470/latest/watteyne_ezwsn_sourcecode.zip>
[18]See the file at <http://cnx.org/content/m22470/latest/watteyne_ezwsn_lab.pdf>
[19]See the file at <http://cnx.org/content/m22470/latest/watteyne_ezwsn_exam.pdf>

Chapter 1

Lab 1: The Development Environment

1.1 1.1. eZ430-RF2500 Sensor Monitor Demo[1]

NOTE: The eZ430-RF2500 board comes with demo software, which creates a WSN in a star topology around the **Access Point**. A node which is not an access point is called an **End Device**. By running this demo, you will learn how to use the IAR tool.

1.1.1 Running the Code

- Locate the folder containing the source code of the Sensor Monitor Demo, which was created when you followed the installation instructions (Section) [2].
- Double-click on `eZ430-RF2500 Sensor Monitor Demo v1.02.eww`, this opens IAR.
- In the Workspace, select the `Overview tab`, right click on the `End Device` project and choose `Set as Active`.
- Plug in the USB programmer, and select `Project > debug` *(Ctrl+D)*. The source is compiled, downloaded onto the target board and a default breakpoint causes execution to stop at function `main()`. Stop the debug session by selecting `Debug > Stop Debugging` *(Ctrl+Shift+D)*;
- Disconnect the USB programmer and swap the target boards between the USB programmer and the battery unit.
- In Workspace, set the `Access Point` project as active. Repeat the programming process. You now have two programmed target boards; close IAR.

1.1.2 Running the Demo

- Plug in the Access Point target board mounted on the USB programmer in the computer. Both LEDs start blinking.
- Launch `Start Menu > ezwsn > eZ430-RF2500 Sensor Monitor`, displaying the temperature of the Access Point.
- You see the temperature of the access point. Switch on your end Device, which appears on the screen

[1] This content is available online at <http://cnx.org/content/m21580/1.3/>.

[2] Alternatively, this code is available in the downloadable source code (<http://cnx.org/content/m22470/latest/watteyne_ezwsn_sourcecode.zip
in the `ti_demo` folder.

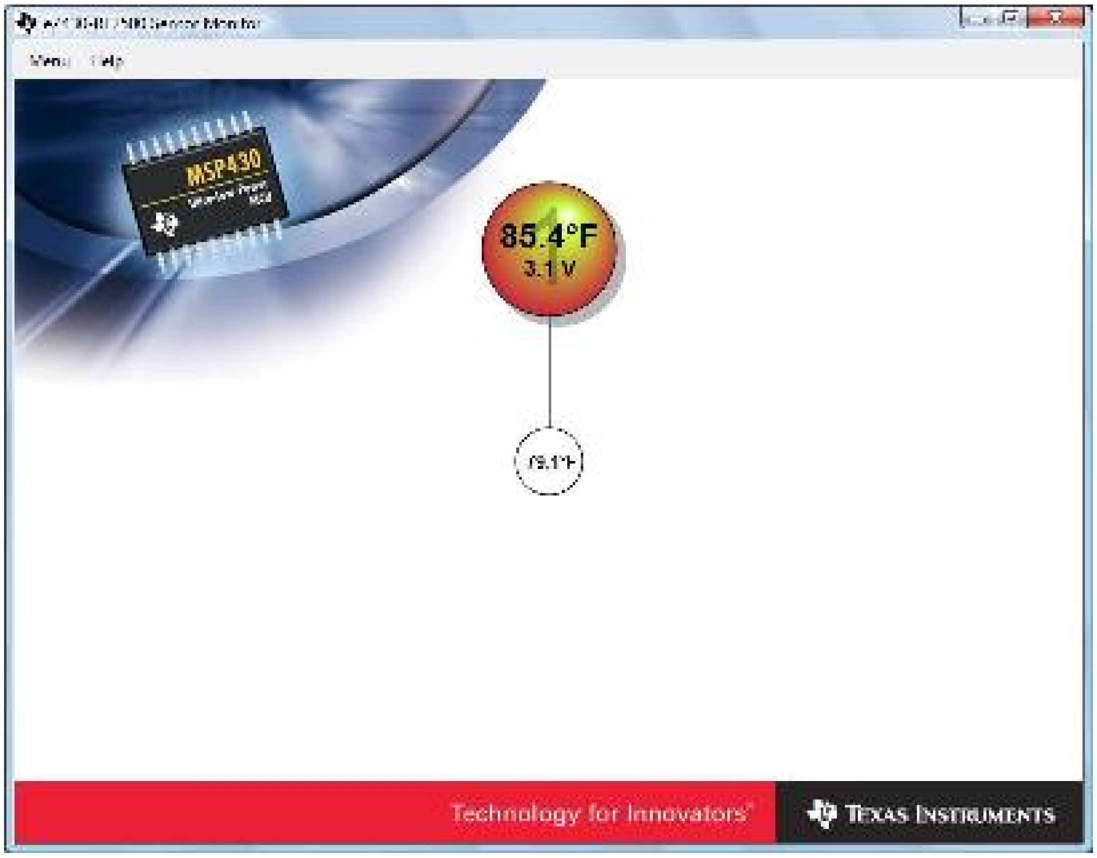


Figure 1.1: You're one when you have a similar window on your screen.

1.2 1.2. Oscilloscope: Read the Current Consumption[3]

NOTE: The End Device is instructed to transmit a message every second; its radio is off the remainder of the time. Using the oscilloscope together with the 1-Ohm resistor, you will be able to visualize the current consumption of the board.

- Leave the Access Point plugged into the computer.
- Switch the target board on using the jumper with the 1-Ohm resistor:

[3]This content is available online at <http://cnx.org/content/m21581/1.1/>.

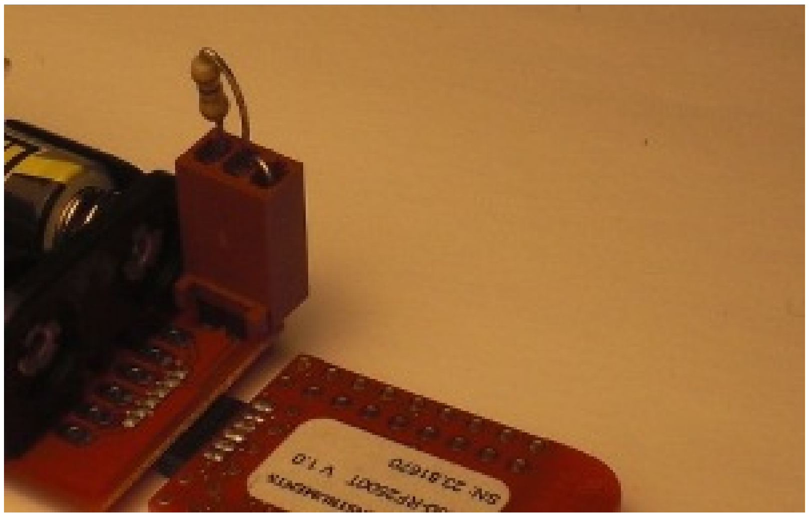


Figure 1.2: Use the 1-Ohm jumper to switch on the target board.

- Connect the oscilloscope onto the resistor as in the following figure. What you read is the current consumption in amps of the board (U = RI, with R = 1).

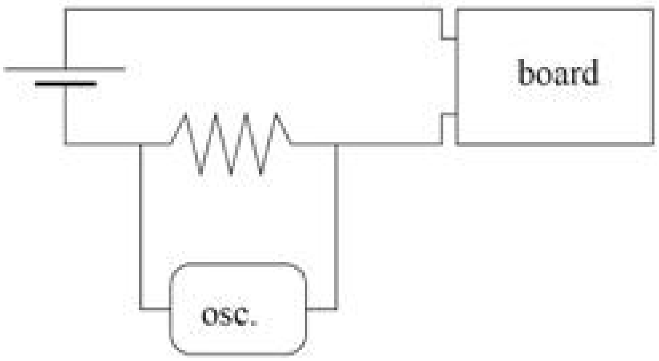


Figure 1.3: Read the current consumption of the board.

- You should read energy consumptions close to the following Figure. There are fourmain phases: the radio is switched on (A), IDLE mode (B), RX mode (C) and TX mode (D).

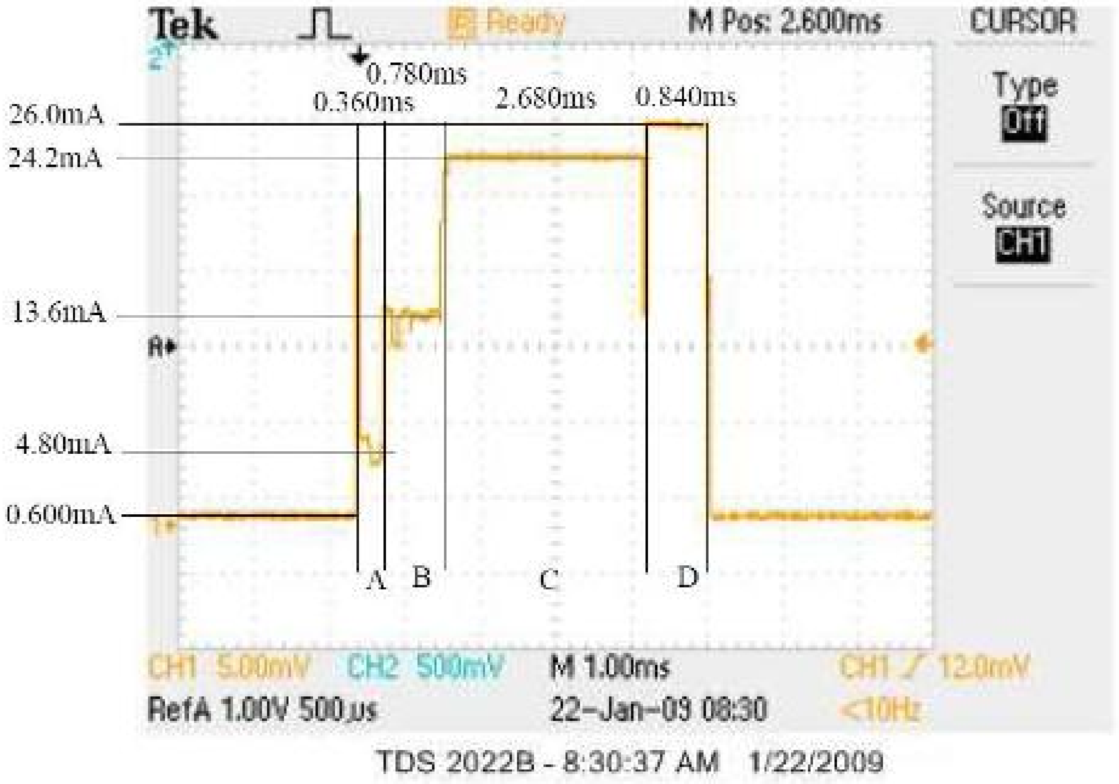


Figure 1.4: You're done when you see a screen similar to this one on your oscilloscope.

- We can assume the boards are powered by two AAA batteries with a capacity of 1000 mA*hr, under the hypothetical condition in which the batteries hold their voltage ideally until their capacity is exhausted. The following Table can be used to calculate the approximate lifetime, using the previous measurements: Average current consumption is 0.700mA; Lifetime with a 1000mA*hr battery is about 2 months.

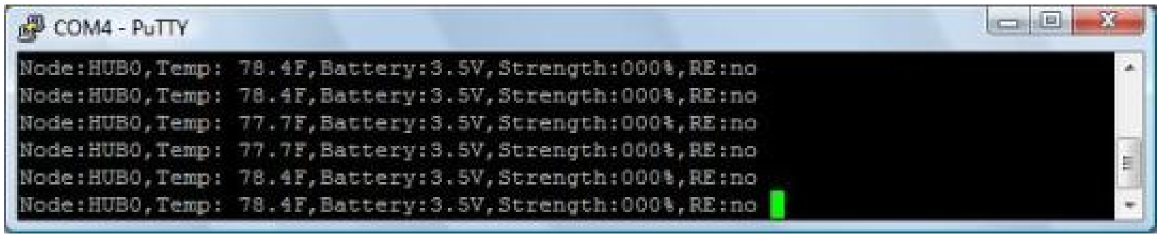
Phase	duration	av. current
sleep	995.34ms	0.600mA
radio is switched on	0.360ms	4.80mA
IDLE mode	0.780ms	13.6mA
RX mode	2.680ms	24.2mA
TX mode	0.840ms	26.0mA

Table 1.1: Duration and average current consumption of the different phases observed.

1.3 1.3. Read Directly From COM Port[4]

NOTE: So far, the SensorMonitor visualizer has extracted the data the access point node and displayed it graphically. In this section, you will read the data coming from the access point directly from the COM port, i.e. you'll have access to the raw data. You will learn how to use PuTTY and Windows' Device Manager.

- Close all open windows; plug in the access point board.
- Open the windows Device manager (Vista: `Start, Settings, Control Panel, Device Manager`; XP: `Start, Settings, Control Panel, Computer Management, Device Manager`).
- Under `Ports (COM and LPT)`, you'll see a device called `MSP430 Application UART (COMx)`, remember that number x;
- Open PuTTY, select Connection Type: Serial and enter the correct COMx in Serial Line, leave Speed at 9600, click Open
- You obtain a screen close to the following:



```
Node:HUB0,Temp: 78.4F,Battery:3.5V,Strength:000%,RE:no
Node:HUB0,Temp: 78.4F,Battery:3.5V,Strength:000%,RE:no
Node:HUB0,Temp: 77.7F,Battery:3.5V,Strength:000%,RE:no
Node:HUB0,Temp: 77.7F,Battery:3.5V,Strength:000%,RE:no
Node:HUB0,Temp: 78.4F,Battery:3.5V,Strength:000%,RE:no
Node:HUB0,Temp: 78.4F,Battery:3.5V,Strength:000%,RE:no
```

Figure 1.5: You're done when you see a screen similar to this one.

[4]This content is available online at <http://cnx.org/content/m21582/1.3/>.

Chapter 2

Lab 2: eZ430-RF2500 Board and its Components

2.1 2.1. Crash Course on the MSP430f2274[1]

> NOTE: So far, you have played around with existing code and have had a high level view of the mote. It is now time to dig into the hardware and understand both what the mote board is composed of, and how you can program it. This section is purely theoretical (i.e. no exercises), but serves are a basis for the subsequent sections in which you will have to write software for the board.

The heart of this platform is its **MSP430 microcontroller**, by Texas Instruments. There is a complete family of MSP430 micro-controllers, the variants of which are different in the amount of RAM/ROM and I/O capabilities, mainly. The one you will program is the MSP420f2274, featuring 32KB + 256B of Flash Memory (ROM) and 1KB of RAM.

The MSP430 is a 16-bit RISC microcontroller. 16-bit means that all registers hold 16 bits; interconnection between the elements of the micro-controller is done using 16-bit buses. RISC – for Reduced Instruction Set Computer – refers to the fact that there are (only) 27 core instructions.

2.1.1 Operation of the MSP430

The following figures shows the internal architecture of the MSP430. The CPU contains 16 **registers**; its operation goes as follows. A system clock ticks at a programmable rate (e.g. 1MHz), so each μs an instruction is fetched from memory (ROM), copied into the right register and executed. An example execution can be adding two registers and copying the result to a third. In practice, these low-level details are taken care of by the compiler, which translates C into assembler language and binary code. In this tutorial, we only work with the higher-level C.

[1]This content is available online at <http://cnx.org/content/m21583/1.3/>.

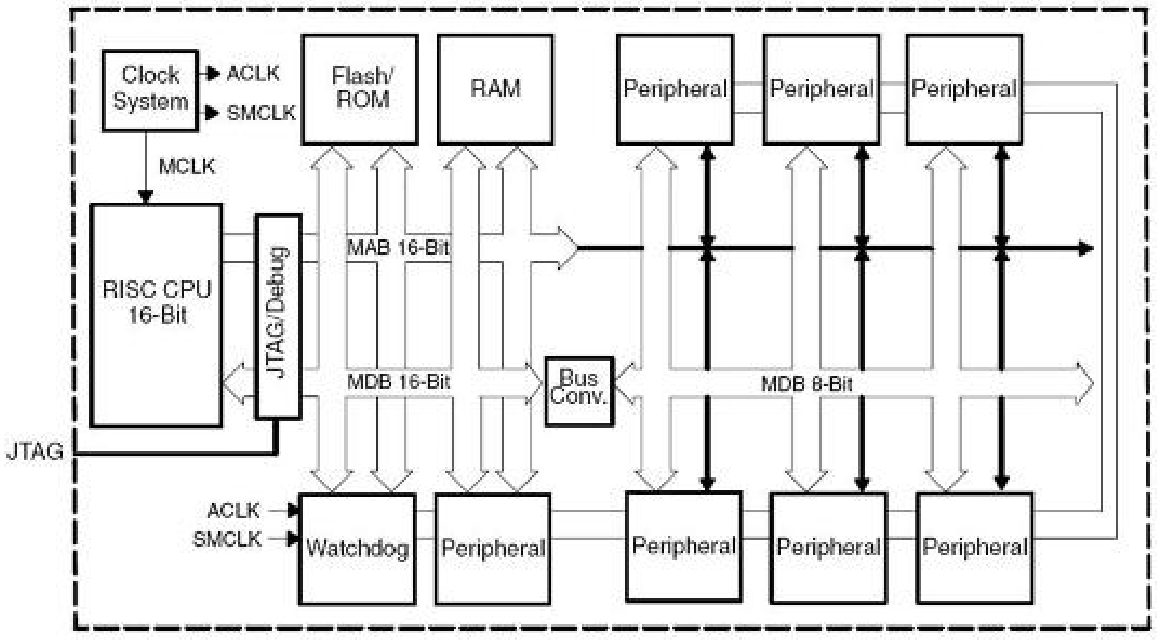


Figure 2.1: The internal architecture of the MSP430.

2.1.2 Programming the MSP430

When you program a mote, you program its microcontroller, i.e. you put the compiled binary code at the right location in the MSP430's ROM memory. When the board is switched on, the MSP430 starts by fetching the first instruction at a predetermined location in ROM; this is where the programming tool puts your compiled code.

To configure other components (e.g. to set the transmission power of the radio), you need to program the MSP430 in such a way that it configures the radio at the beginning of your program.

2.1.3 Interrupts

A program for a wireless mote is in practice a sequence of very small pieces of codes executed when some event happens: e.g. when the button is pressed, turn on the red LED. When an event happens in an electronic element outside the MSP430 (e.g. the button is pressed), this element informs the MSP430 by changing the electric state of the wire which connects this element to the MSP430. This wire is connected to one of the ports of the MSP430 (e.g. port `P1.2` in the case of the button on the eZ430-RF2500). You need to program the MSP430 in such a way that changing the status on port P1.2 generates an interrupt. When an interrupt is generated, theMSP430 stops its current execution (if any), and starts executing a specific function called the Interrupt Service Routine (**ISR**) associated to that particular interrupt. Once this function is finished (normally an ISR is a very small function), it resumes its normal execution (if any).

You will write ISRs in to response when pushing the button (3.3 (Section 3.3)), timer interrupts (3.4 (Section 3.4)) and when receiving a packet at the radio (4.2 (Section 4.2)).

2.1.4 Timers

When writing code, you may want to wait some time before doing something (e.g. when I receive a packet, wait 10ms, and send a reply packet). This can be done using a timer, a specific component of the MSP430. Physically, a timer is a 16-bit register which is incremented at each clock cycle, i.e. once every μs with a 1MHz clock. It starts at 0, and counts up until a programmable value, upon which is generates a timer interrupt, reset to 0, and starts counting up again.

You will use timer in section to have a LED flash at a given rate (3.4 (Section 3.4)).

2.1.5 I/O Functionalities

The MSP430 has 40 pins:

- 4 have analog functions to power the board;
- 2 are used for testing at the factory;
- 2 are used if an external crystal is used as clock source, which is not the case on the eZ430-RF2500 platform;
- 32 have digital functions.

The 32 digital pins are grouped into 4 ports of 8 pins each. Each pin has a name in the form `Px.y`, `y` represents the position of the pins within port `x`. All pins can be generic I/O pins, a number of 8-bit registers are used to configure them:

- `PxDIR.y` sets the direction of port `PxDIR.y`; output if `PxDIR.y=1`, input if `PxDIR.y=0`;
- `PxOUT.y` sets the state of port `Px.y` when set as output;
- `PxIN.y` reads the state of port `Px.y` when set as input;
- `PxIE.y` enables interrupts on that port;

Each of these registers hold 8 bits, one for each pin. As a result, `P1DIR=0b11110000`[2] means that pins `P1.0` through `P1.3` are input, while `P1.4` through `P1.7` are outputs. To set/reset a specific pin, you need to use the binary operators presented in the following code:

```
   Assuming A =
0b01101001, we have:
~A = 0b10010110
A |= 0b00000010: A=0b01101011
A &= ~0b00001000: A=0b01100001
A ^= 0b10001000: A=0b11100001
A << 2: A=0b10100100
A >> 2: A=0b00011010Binary operators used to set/reset individual
bits.
```

Note that most of the 32 digital pins can also be used for specific functions (SPI interface, input for Analog-to-Digital conversion, ...), see the MSP430x22x2, MSP430x22x4 Mixed Signal Microcontroller[3] datasheet for details.

2.1.6 Low-Power Operation

As the MSP430 spends its time waiting for interrupts, it is important to reduce its energy consumption during idle periods by shutting down the clocks you are not using. The more clocks you shut down, the less

[2] `0bx` means that `x` is written in binary; `0xx` means that `x` is written in hexadecimal. We thus have `0x1A=0b00011010`. Use Windows Calculator in Scientific mode for quick conversions.

[3] http://focus.ti.com/lit/ds/symlink/msp430f2274.pdf

energy you use, but make sure you leave on the clocks you need. There are four low power modes (LPM1, ..., LPM4) which shut down different clocks (details in the MSP430x2xx Family User's Guide[4]).

In practice, you only need to leave on the auxiliary clock which clocks a timer to wake the MSP430 after some time. This is achieved by entering low-power mode 3, by adding this line at the end of your main function:

```
__bis_SR_register(LPM3_bits);
```

TIP: You now know enough about the MSP430 for this tutorial, but if you want to work with the MSP430, you are strongly advised to read the MSP430x2xx Family User's Guide[5] and the MSP430x22x2, MSP430x22x4 Mixed Signal Microcontroller[6] datasheet (in that order).

2.2 2.2. Crash Course on the CC2500[7]

The CC2500 is the **radio chip** on the eZ430-RF2500. It functions in the **2400- 2483.5 MHz** frequency band and provides an excellent option for WSN applications because of its low-power characteristics. This chip has 20 pins:

- 2 for connecting a (mandatory) 26MHz external crystal oscillator;
- 2 for connecting the antenna;
- 10 for powering the chip;
- 6 for digital communication with the MSP430 (to be detailed in section 3.3)

The chip contains **47 registers** to configure operating frequency,modulation scheme, baud rate, transmission power, etc. Because these registers are erased during power down, the MSP430 should configure all of them at startup. 13 commands allow the MSP430 to control the state of the CC2500 (transmit, power down, receive, . . .). The CC2500 follows a state diagram, as detailed in the CC2500, Low-Cost Low-Power 2.4 GHz RF Transceiver[8] datasheet.

In practice, Texas Instruments provides some code which hides the low-level details of the CC2500 behind a higher level API. These **drivers** are part of the SimpliciTI project[9], available for free. We will use these off-the-shelf drivers in this tutorial.

TIP: You now know enough about the CC2500 for this tutorial, but if you want to work with the CC2500, you are strongly advised to read the CC2500, Low-Cost Low-Power 2.4 GHz RF Transceiver[10] datasheet (after having read the documents about the MSP430).

2.3 2.3. The eZ430-RF2500 Board[11]

2.3.1 Overview

The following figure shows the different components on the eZ430-RF2500. In particular, some of pins of the MSP430 are exported as **extension pins** `P1` through `P18`. Note that some of these pins may be unused pins of the MSP430, others are already used, but duplicated as extension ports for debugging purposes.

[4]http://focus.ti.com/lit/ug/slau144e/slau144e.pdf
[5]http://focus.ti.com/lit/ug/slau144e/slau144e.pdf
[6]http://focus.ti.com/lit/ds/symlink/msp430f2274.pdf
[7]This content is available online at <http://cnx.org/content/m21584/1.2/>.
[8]http://focus.ti.com/lit/ds/symlink/cc2500.pdf
[9]http://focus.ti.com/docs/toolsw/folders/print/simpliciti.html
[10]http://focus.ti.com/lit/ds/symlink/cc2500.pdf
[11]This content is available online at <http://cnx.org/content/m21585/1.2/>.

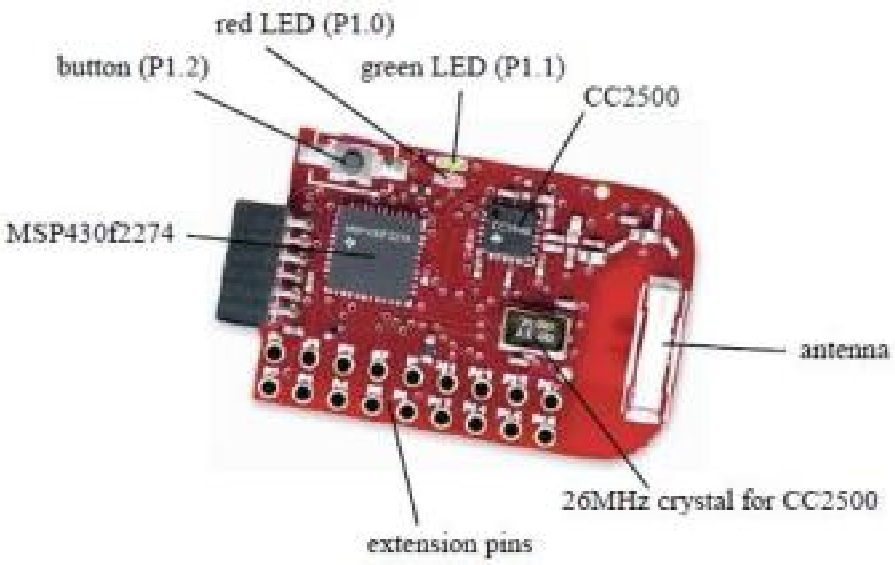


Figure 2.2: The components of the eZ430-RF2500.

2.3.2 Interconnecting the MSP430 with the CC2500

As show in the following figure, **6 wires** interconnect the **MSP430** (micro-controller) with the **CC2500** (radio). 4 of them form the **SPI** link, a serial link which enables digital communication. There is a hardware SPI modem on each side of the link, the configuration of which is handled by the drivers.

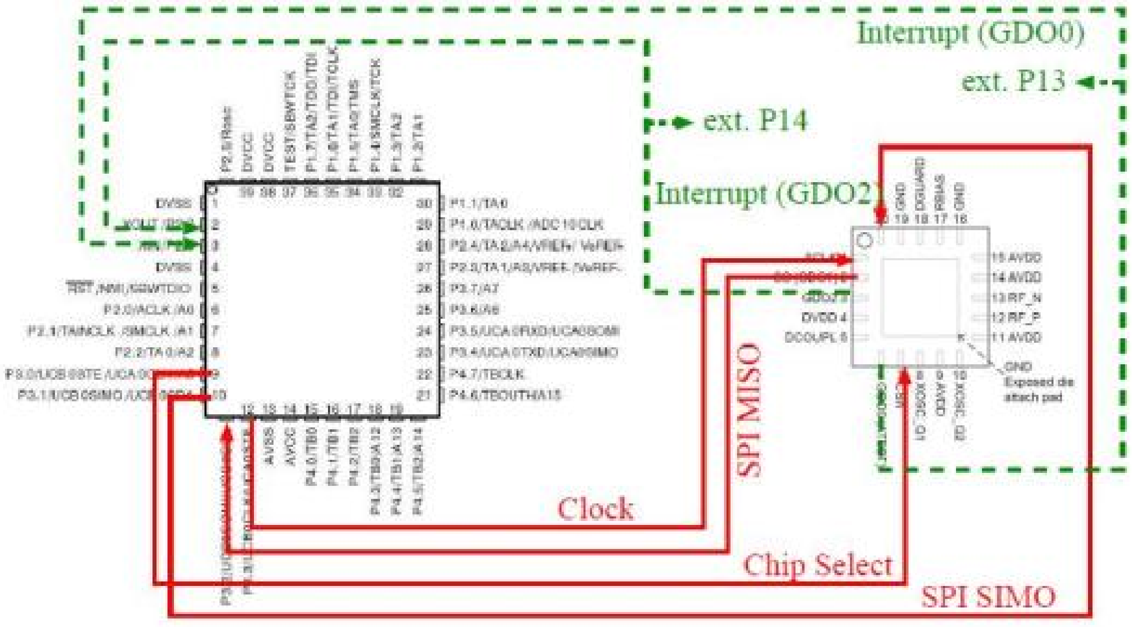


Figure 2.3: MSP430 and CC2500 are interconnected by an SPI 4-wire link (plain) and two interrupt lines (dotted).

The remaining two links are wires used by the CC2500 to wake-up the MSP430. That is, the MSP430 configures its ports `P2.6` and `P2.7` as input, with interrupts. It then configures the CC2500 to trigger `GDO0` or `GDO2` on a given event (typically, when receiving a packet). This enables the MSP430 to enter LPM3. Note that these wires are also routed to extension port pins `P13` and `P14`. The drivers are used in such a way that only `GDO0` is used for proper operation. You can thus configure `GDO2` as you wish, and monitor its state with an oscilloscope on extension port pin `P14`.

Refer to the eZ430-RF2500 Development Tool User's Guide[12] for details about the eZ430-RF2500 platform.

[12]http://cnx.org/content/m21585/latest/focus.ti.com.cn/cn/lit/ug/slau227d/slau227d.pdf

Chapter 3

Lab 3: Flashing LEDs, Simple Programming Examples for the MSP430

3.1 3.1. A Steady LED[1]

3.1.1 Creating a Project in IAR

If you want to create your own project in IAR, you need to repeat these steps:

- Connect the eZ430-RF2500 programming board to the computer and open IAR;
- Choose `Create new project in current workspace`;
- Leave Tool Chain to MSP430; as project template, choose `C > main`; click `OK`;
- Create a directory on your desktop in which you save your project;
- Go to `Project > Option` *(Alt+F7)*, in General Option, choose `Device=MSP430F2274`; in Debugger, choose `Driver=FET Debugger`.

For your convenience, you can download the entire source code[2] for the labs in this collection, organized in IAR projects. Open `source_code/iar_v4.11.lab_ezwsn.eww` and browse through the different projects in the `Workspace` panel on the left.

3.1.2 Running the Code

- Copy the following code into IAR [3].
- Compile and download the code onto the board *(Ctrl+D)*.
- Let the code execute *(F5)*, you should now see both LEDs on.

```
    #include "io430.h"
int main( void )
{
  WDTCTL = WDTPW + WDTHOLD;
  P1DIR |= 0x03;
  P1OUT |= 0x03;
  while(1);
}Switching on both LEDs
```

[1]This content is available online at <http://cnx.org/content/m21586/1.4/>.

[2]http://cnx.org/content/m22470/latest/watteyne_ezwsn_sourcecode.zip

[3]Alternatively, this code is available in the downloadable source code (<http://cnx.org/content/m22470/latest/watteyne_ezwsn_sourcecode.zip
Open `source_code/iar_v4.11/lab_ezwsn.eww` with IAR. The project corresponding to this section is called `led_steady`.

Some keys for understanding the code:

- **Line 1**: `io430.h` contains all the macros used for translating human readable values (e.g. `P1DIR`) into actual memory location. Right click on `io430.h` and choose `Open "io430.h"` to see its content.
- **Line 4**: The MSP430 has a watchdog timer which resets the board if it is not reset before it elapses. This way, if you code hangs, the board restarts and continues functioning. For our simple examples, we disactivate this function by writing the correct values into register `WDTCTL`.
- **Line 5** declares `P1.0` and `P1.1` as output pins. This is done by turning bits 0 and 1 to 1 in register `P1DIR`;
- **Line 6** sets the output state of pins `P1.0` and `P1.1` to logic 1 (physically somewhere between 2.2V an 3.5V). This causes the LEDs, connected to those pins, to light.
- **Line 7** loops, leaving the board running.

3.1.3 Energy Consumption

- Comment out line 6 and run the board from the battery unit. Use the resistor jumper and the oscilloscope to read out the default energy consumption (i.e. MSP430 running, no LEDs, no CC2500);
- Repeat this by leaving line 6. By subtracting the results, you can measure the energy consumption of the LEDs;
- replace line 6 by `P1OUT |= 0x01` and `P1OUT |= 0x02` will leave on the red and green LEDs only, respectively. You can now measure the consumption of the LEDs independently.

Make sure you obtain results close to the ones presented in the following table.

no lEDs (MSP430 running)	3.12mA
red+green LED	9.12-3.12=6.00mA
red LED	7.04-3.12=3.92mA
green LED	5.20-3.12=2.08mA

Table 3.1: Current consumed by the LEDs

3.2 3.2. Active Waiting Loop[4]

NOTE: This example shows a first way of measuring time. `__no_operation()` instructs the MSP430 to do nothing during one cycle; by repeating this many times, time can be measured. As this is neither accurate nor energy-efficient, a more elegant technique will be shown in section 3.4 (Section 3.4).

3.2.1 Running the Code

- Copy the code presented below [5].
- Compile and download the code onto the board *(Ctrl+D)*.
- Let the code execute *(F5)*, both LEDs should blink.

[4]This content is available online at <http://cnx.org/content/m21587/1.3/>.

[5]Alternatively, this code is available in the downloadable source code (<http://cnx.org/content/m22470/latest/watteyne_ezwsn_sourcecode.zip
Open `source_code/iar_v4.11/lab_ezwsn.eww` with IAR. The project corresponding to this section is called `led_loop`.

```
    #include "io430.h"
#include "in430.h"
int main( void )
{
  WDTCTL = WDTPW + WDTHOLD;
  int i;
  P1DIR |= 0x03;
  while (1) {
    P1OUT ^= 0x03;
    for (i=0;i<10000;i++) {
      __no_operation();
    }
  }
}Blinking LEDs using an active waiting loop
```

Some keys for understanding the code:

- **Line 9**: the operator `^=` causes 1s to become 0s and vice-versa (aka toggling).In case of our LEDs, it causes their on/off state to change;
- **Line 10**: `__no_operation();` causes the MSP430 to do nothing for one cycle.

3.2.2 Measuring Time

We want to measure time precisely with the oscilloscope. This can, in theory, be done by measuring the voltage at the LEDs, but it is hard to hold the probes right. We will therefore use extension pin `P6` represented in the figure below, which is connected to `P2.3` on the MSP430.

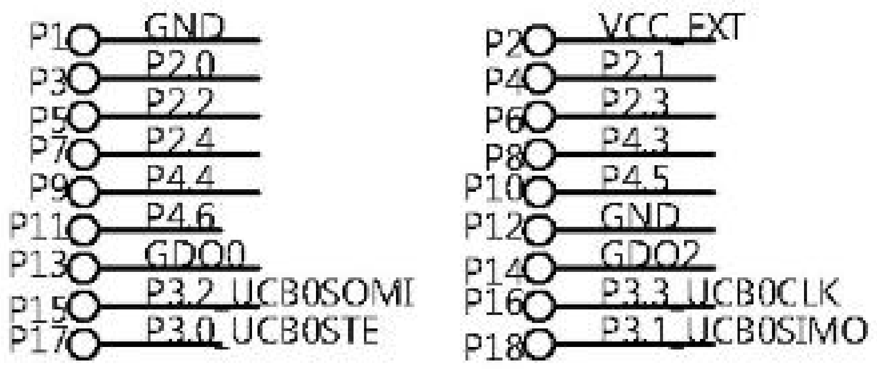


Figure 3.1: The extension pins on the eZ430-RF2500 board. Note that the pins with even number (shown on the right) are located on the edge of the board, and are thus accessible more easily.

We will configure `P2.3` to be output and toggle its state together with the state of the LEDs. Therefore:

- add line `P2DIR |= 0x08;` after line 7 to declare `P2.3` as output;
- add line `P2OUT ^= 0x08;` after line 9 to toggle `P2.3` after toggling the LEDs;
- connect a probe of your oscilloscope to extension port `P6`, ground on extension port `P12`;
- power on the board, you're now able to read the duration between two toggles;

- reprogram your board with waiting values between 1000 and 30000; verify that the times obtained are close to the ones presented the following table.

threshold value for `i`	measured toggle duration
1000	6.72ms
10000	67.2ms
20000	134.0ms
30000	201.0ms

Table 3.2: Duration when using an active waiting loop

3.3 3.3. Button-Driven Toggle Through Interrupts[6]

NOTE: The goal of this section is to start using interrupts through the button on the board. You will program the board so that the LEDs change state when the button is pressed. You will also measure the energy consumed when the board sits idle, and when it enters a low-power mode.

3.3.1 Running the Code

- Copy the code presented in the following listing [7].
- Compile and download the code onto the board *(Ctrl+D)*.
- Let the code execute *(F5)*, press the button on the board, the LEDs' state should change.

```
    #include "io430.h"
#include "in430.h"
int main( void )
{
  WDTCTL = WDTPW + WDTHOLD;
  P1DIR |=  0x03;
  P1DIR &= ~0x04;
  P1REN |= 0x04;
  P1IE |= 0x04;
  __bis_SR_register(GIE);
  while(1);
}
#pragma vector=PORT1_VECTOR
__interrupt void Port_1 (void)
{
  P1IFG &= ~0x04;
  P1OUT ^= 0x03;
}Toggle LEDs state using the button
```

Some keys for understanding the code:

- **Lines 6 and 7** declare `P1.0` and `P1.1` as outputs (for the LEDs), and `P1.2` as input (for the button);

[6]This content is available online at <http://cnx.org/content/m21588/1.4/>.
[7]Alternatively, this code is available in the downloadable source code (<http://cnx.org/content/m22470/latest/watteyne_ezwsn_sourcecode.zip
Open `source_code/iar_v4.11/lab_ezwsn.eww` with IAR. The project corresponding to this section is called `led_button`.

- **Line 8** activates an internal resistor on `P1.2`. This is needed for a button for the signal to be cleaner when the button is pressed; i.e. otherwise, the `P1.2` constantly floats between high and low state. This is only needed for buttons.
- **Line 9** enables interrupts on `P1.2`.
- **Line 10** enables interrupts globally.
- **Line 13 and 14** declare that this function should be called when an interrupt of type `PORT1_VECTOR` happens; the name `Port_1` chosen has no importance.
- **Line 16** resets the interrupt flag generated (mandatory otherwise the function will be called again right after it finishes execution).

3.3.2 Low-Power Modes

As such, the board sits idle while waiting for an interrupt to happen with the MSP430 on (which continuously executed line 11). After measuring this current,you will change the code so as to enter low-power mode instead of sitting idle.

- Using the battery unit, the resistor jumper and the oscilloscope, measure the current consumed in this mode (make sure that the LEDs are off when you measure).
- Change line 10 by `__bis_SR_register(GIE+LPM0_bits);`. This instructs the MSP430 to enable the interrupts globally, and to enter LPM0 mode immediately. Only an interrupt can wake the MSP430.
- Remove line 11 which can never be reached.
- Measure the current now, make sure that LEDs are again off.
- repeat with LPM3 and LPM4.

You should obtain results close to:

Active mode *(active: CPU and all clock)*	3.28mA
LPM0 mode *(active: SMCLK, ACLK; disabled: CPU, MCLK)*	2.88mA
LPM3 mode *(active: ACLK; disabled: CPU, MCLK, SMCLK)*	2.72mA
LPM4 mode *(disabled: CPU and all clock)*	2.72mA

Table 3.3: Current consumed by the LPM modes

Note that, because we are not using any clock in this example, we should use LPM4 as it is the most energy-efficient.

3.4 3.4. Timer-Driven Toggle Through Timer Interrupts[8]

NOTE: We will explore a energy-efficient way of measuring time by using timers. A timer is a register which counts up to a certain number at each clock tick. During this, the MSP430 can switch to a low-power mode. Each time the timer threshold is reached, a timer interrupt wakes the MSP430, which toggles the two LEDs.

3.4.1 Running the Code

- Copy the code presented below [9].

[8]This content is available online at <http://cnx.org/content/m21589/1.3/>.
[9]Alternatively, this code is available in the downloadable source code (<http://cnx.org/content/m22470/latest/watteyne_ezwsn_sourcecode.zip
Open `source_code/iar_v4.11/lab_ezwsn.eww` with IAR. The project corresponding to this section is called `led_timer`.

- Compile and download the code onto the board *(Ctrl+D)*.
- Let the code execute *(F5)*, both LEDs should blink.

```
    #include "io430.h"
#include "in430.h"
int main( void )
{
  WDTCTL = WDTPW + WDTHOLD;
  P1DIR |=  0x03;
  BCSCTL3 |= LFXT1S_2;
  TACCTL0 = CCIE;
  TACCR0 = 1000;
  TACTL = MC_1+TASSEL_1;
  __bis_SR_register(GIE+LPM3_bits);
}
#pragma vector=TIMERA0_VECTOR
__interrupt void Timer_A (void)
{
  P1OUT ^= 0x03;
}Blink LEDs using timer interrupts
```

Some keys for understanding the code:

- **Line 7** switches on the ACLK by sourcing it to the **VLO**, a very low power crystal oscillator inside the MSP430, different from the more accurate and energy hungry **DCO** (Digitally Controlled Oscillator). The DCO drives the MCLK and the VLO the ACLK. ACLK is used for the timer; MCLK for executing code. When there is no code to be executed (i.e. when waiting for interrupts), a low power mode (in which DCO is switched off) can be used.
- **Line 8** enables interrupts for `Timer_A`.
- **Line 9** sets the value up to which `Timer_A` will count.
- **Line 10** tells `Timer_A` to count up (`MC_1`) each time ACLK ticks (`TASSEL_1`).
- **Line 11** enables interrupts globally and enters LPM3. Note that LPM3 leave only ACLK running, which is exactly what we need because our `Time_A` runs off ACLK.

3.4.2 Measuring Time

We use extension pin `P6` to measure time exactly. Therefore:

- add line `P2DIR |= 0x08;` after line 6 to declare `P2.3` as output;
- add line `P2OUT ^= 0x08;` after line 16 to toggle `P2.3` after toggling the LEDs;
- connect a probe of your oscilloscope to extension port `P6`, ground on extension port `P1`. Power on the board, you're now able to read the duration between two toggles.
- reprogram your board with `TACCR0` values between 1000 and 30000; verify that the times obtained are close to the following:

TACCR0 value	measured toggle duration
500	47.40ms
1000	95.00ms
10000	950.0ms
20000	1890ms

Table 3.4: Duration when using timers

This table informs you that 1000 ACLK cycles take 95.00ms, one ACLK cycle thus takes 95μs, the VLO on the MSP430 thus runs at 10.5kHz. In theory, the VLO runs at 12kHz; the exact value depends on the voltage of the batteries and on the temperature.

Chapter 4

Lab 4: Enabling Wireless Communication

4.1 4.1. Using the Texas Instruments Drivers[1]

Texas Instruments has developed a set of drivers for the eZ430-RF2500 as part of the simpliciTI project, freely available online[2] . You will use them without changing the files.

If you want to create your own project, you need to take the following steps to include the drivers:

- In IAR, create a new project as instructed in Section 3.1 (Section 3.1).
- Go to `Project > Options` *(Alt+F7)*, then to C/C++ compiler. In the `Preprocessor` tab, add the following lines in the `Additional include directories` text field. This tells IAR to look into these folders when you include files in your source code.

```
    $PROJ_DIR$\..\drivers\bsp
$PROJ_DIR$\..\drivers\bsp\drivers
$PROJ_DIR$\..\drivers\bsp\boards\EZ430RF
$PROJ_DIR$\..\drivers\mrfi
```

- In the `defined symbols` text field, add the following line. This tells the drivers you have a CC2500 radio chip on your board.

```
MRFI_CC2500
```

- in the Workspace panel, right click on the name of your project, and use `Add, Add Group...` to make the following group structure (where groups `bsp` and `mrfi` are inside group `Components`):

```
Application
Components
- bsp
- mrfi
Output
```

- use `Add, Add Files...` to add files `bsp.c`, `bsp.h` and `bsp_macros.h` under group `bsp`. Similarly, add files `mrfi.c`, `mrfi.h` and `mrfi_defs.h` under group `mrfi`. Add your C code as a file under group `Application`.

[1]This content is available online at <http://cnx.org/content/m21590/1.3/>.

[2]http://focus.ti.com/docs/toolsw/folders/print/simpliciti.html

For your convenience, you can download the entire source code[3] for the labs in this collection, organized in IAR projects. Open `source_code/iar_v4.11.lab_ezwsn.eww` and browse through the different projects in the `Workspace` panel on the left.

4.2 4.2. Simple Tx/Rx[4]

> NOTE: This first example involves two boards which stay in Rx mode by default. When you press a button on either one, it sends a message and toggles its green LED; the board which receives the message toggles its red LED. Once this is functional, you will play with the Rx/Tx frequency.

4.2.1 Running the Code

- Copy the code presented below [5].
- Compile and download the code onto both boards *(Ctrl+D)*.
- Switch both boards on (one with the battery unit, the other with the USB slot of your PC); when you press one's button, the other's red LED should toggle.

```
    #include "mrfi.h"
#include "radios/family1/mrfi_spi.h"
int main(void)
{
  BSP_Init();
  P1REN |= 0x04;
  P1IE |= 0x04;
  MRFI_Init();
  MRFI_WakeUp();
  MRFI_RxOn();
  __bis_SR_register(GIE+LPM4_bits);
}
void MRFI_RxCompleteISR()
{
  P1OUT ^= 0x02;
}
#pragma vector=PORT1_VECTOR
__interrupt void Port_1 (void)
{
  P1IFG &= ~0x04;
  mrfiPacket_t packet;
  packet.frame[0]=8+20;
  MRFI_Transmit(&packet, MRFI_TX_TYPE_FORCED);
  P1OUT ^= 0x01;
}The simplest Tx/Rx example possible
```

The packet format is:

Length (1B)	Source (4B)	Destination (4B)	Payload (Length-8 bytes long)

[3]http://cnx.org/content/m22470/latest/watteyne_ezwsn_sourcecode.zip

[4]This content is available online at <http://cnx.org/content/m21594/1.3/>.

[5]Alternatively, this code is available in the downloadable source code (<http://cnx.org/content/m22470/latest/watteyne_ezwsn_sourcecode.zip
Open `source_code/iar_v4.11/lab_ezwsn.eww` with IAR. The project corresponding to this section is called `txrx_simple`.

Table 4.1: packet.frame

RSSI (1B)	CRC (1b)	LQI (1b)

Table 4.2: packet.rxMetrics

A variable of type `mrfiPacket_t` is a structure containing two parts:

- `packet.frame` is the frame to be transmitted. The first byte is the `Length` of the payload together with source and destination Address. With the current driver implementation, addresses are coded on 4 bytes, and the maximum Payload length is 20 bytes. By default, the CC2500 does not perform address filtering, so in practice we will not care about the values of the address fields.
- `packet.rxMetrics` are statistics on the last received packet, i.e. it only makes sense on received packet. The first byte is the Received Signal Strength Indicator (**RSSI**) at sync word detection. This is the signal level in dBm. The next bit indicates whether the Cyclic Redundancy Check (**CRC**) was successful (by default, the CC2500 is configured to reject packets with unsuccessful CRC check, so in practice this field will always be 1). The last 7 bits are the Link Quality Indicator (**LQI**). The LQI gives an estimate of how easily a received signal can be demodulated by accumulating the magnitude of the error between ideal constellations and the received signal over the 64 symbols immediately following the sync word.

Some keys for understanding the code:

- **Line 4** is a function from the drivers (right-click on it, and choose `Go to definition of "BSP_Init()"` if you want to know) which disables the watchdog, initializes the MCLK at 8MHz, sets LED ports as outputs and the button port as input. Note that it does neither enables the internal resistor of the button, nor enables interrupts. This is done on lines 5 and 6.
- **Line 7.** MRFI stands for Minimal Radio-Frequency Interface; functions starting with MRFI are used to drive the CC2500 radio chip. `MRFI_Init()` initializes the 6 wires between the MSP430 and the CC2500, powers-up the CC2500 and configures the CC2500 47 registers and turns on interrupts from the CC2500;
- **Line 8** wakes up the radio, i.e. it turns on the 26MHz crystal attach to it without entering Rx or Tx mode;
- **Line 9** switches the radio to Rx mode; from this line on, it can receive packets, in which case the interrupt function `MRFI_RxCompleteISR` is called [6].

4.2.2 Choosing a Frequency

The CC2500 can transmit at any frequency on the **ISM** band **2400.0-2483.5 MHz**. The chip divides the 2400.0-2483.5 MHz spectrum into **channels** separated by a tunable **channel spacing**. By default, channel spacing is 200kHz; with default configuration, channel 0 is 2433.0Mhz, channel 1 is 2433.2Mhz, and so forth. The channel is configured through the `CHANNR` register of the CC2500.

- add 1 the following line at the very top of the code. This way, you have access to low level driver functions which enable you to write directly the CC2500 registers:

  ```
  #include "radios/family1/mrfi_spi.h"
  ```

- add the following line right after `MRFI_Init();`. You may replace `0x10` by the frequency you have chosen. This programs the `CHANNR` register of the CC2500. Be aware that values above `0xFC` are prohibited because the corresponding frequency is above 2483.5 MHz:

  ```
  mrfiSpiWriteReg(CHANNR,0x10);
  ```

When you have reprogrammed both boards, you should be able to communicate without interference.

[6]Note that the real interrupt function with the usual `#pragma` declaration is hidden in the drivers

4.3 4.3. Continuous Tx/Rx[7]

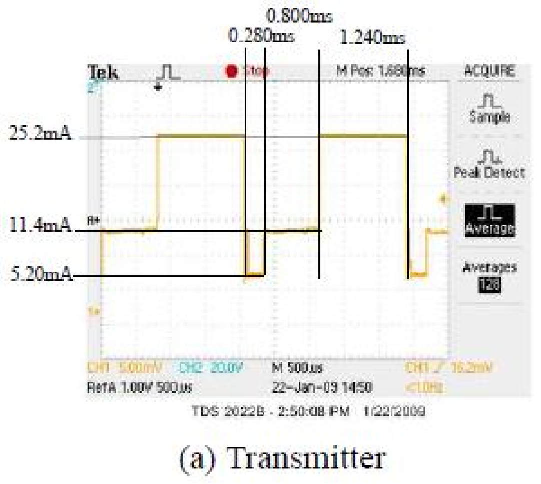
NOTE: We abandon the push button, and ask one board to sent messages continuously, while the other receives. Once this constant flow of data is established, you will be able to see the energy consumption of the transmitting and receiving boards, for a number of settings of transmission power.

4.3.1 Running the Code

- Start from the code used in 4.2. Simple Tx/Rx (Section 4.2).
- Replace line `P1IFG &= ~0x04;` by `P1IFG |= 0x04;`. This means that once you push the button, the board will start sending an infinite number of messages. This is because you force the interrupt flag to remain active, i.e. when your code leave the Interrupt Service Routine, it immediately re-enters because, according to the interrupt flag, there is still and interrupt pending.
- Reprogram both boards, switch them on, and push one button.

4.3.2 Energy Consumption

- You may wish to command out lines `P1OUT ^= 0x02;` and `P1OUT ^= 0x01;` so that your LEDs remain off not to measure their energy consumption;
- Use the oscilloscope to measure the energy consumption;
- Make sure you obtain results which are close to the following figures.



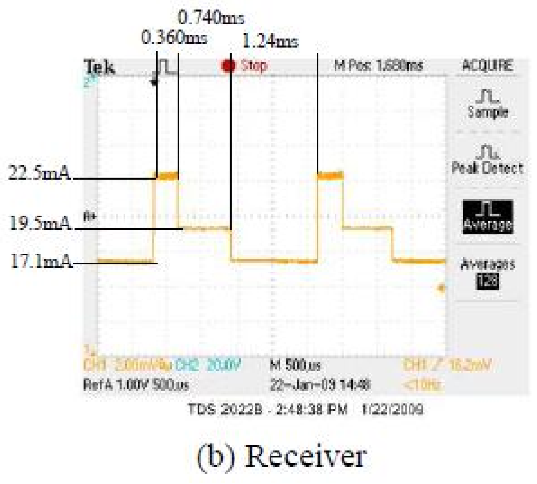


Figure 4.1: Energy consumption with continuous Tx/Rx.

IMPORTANT: As you can see, the radio accounts for most of the energy consumed by the mote. Additionally, as a rule of thumb you may safely consider that transmitting, receiving and idle listening cost about the same energy.

[7]This content is available online at <http://cnx.org/content/m21595/1.2/>.

4.3.3 Impact of Transmission Power

We want to measure the impact of the transmission power on the current consumption of the board. Therefore:

- if not there yet, add the following line at the beginning of your file:

  ```
  #include "radios/family1/mrfi_spi.h"
  ```

- right after line `MRFI_Init();`, add

  ```
  mrfiSpiWriteReg(PATABLE,0xFF);
  ```

  . This programs the `PATABLE` register of the CC2500, which is responsible for the transmission power of the radio. You can put any value between `0x00` and `0xFF;`, which is then mapped to a transmission power as shown in the next table.
- Use your oscilloscope to visualize the maximum energy consumed. Repeat this for different values of `PATABLE`. Check that you have results close to the ones presented in the next figure.

register	power	current
0x84	-24dBm	16.1mA
0x55	-16dBm	16.5mA
0x97	-10dBm	18.3mA
0xA9	-4dBm	22.6mA
0xFE	0dBm	27.6mA
0xFF	1dBm	27.9mA

Table 4.3: Impact of transmission power on current

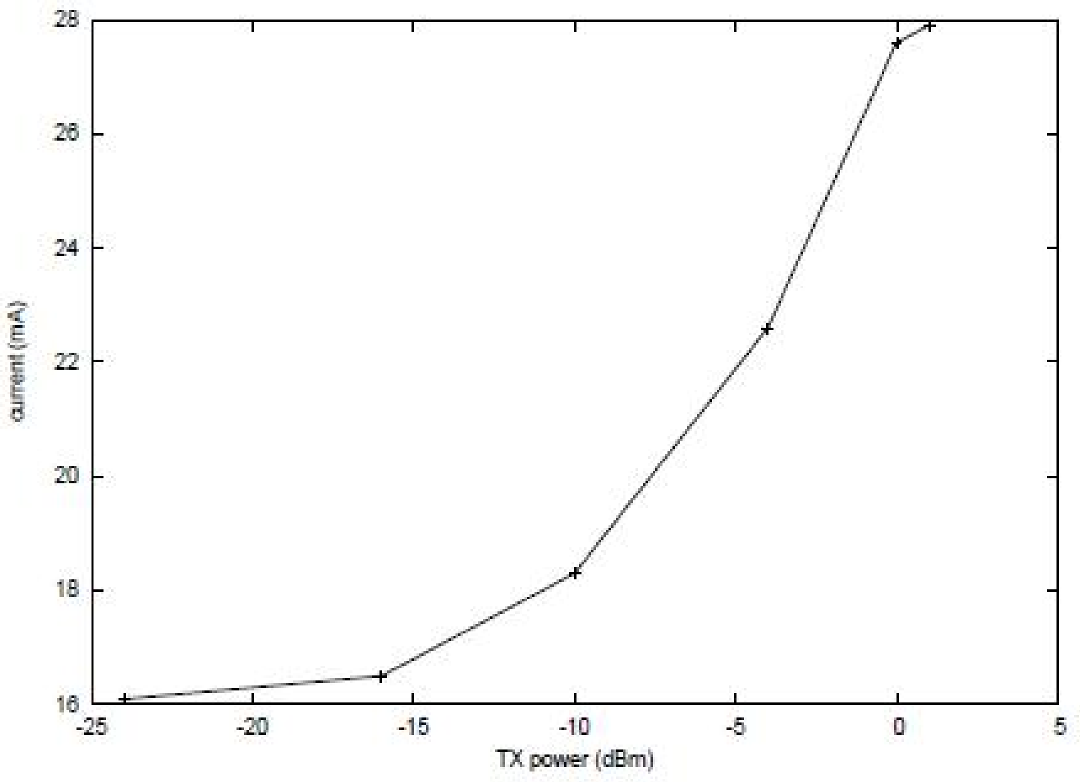


Figure 4.2: Impact of transmission power on current

IMPORTANT: Varying the transmission power from -20dBm to 0dBm (i.e. 200 times more power) only doubles the motes current consumption. This is because the electronics account for the largest amount of energy: at 0dBm, 1mW is sent over the antenna, while the mote consumes 27.6mA*3V=82.8mW.

4.4 4.4. Wireless Chat[8]

NOTE: Now that you can send packets between nodes, you will put content into those packets. Data is entered through the keyboard and sent over the air when pressing enter. The receiver prints the received data on the screen. You are thus building a wireless chat program.

IMPORTANT: Because you want to type to and read from both motes, you need to have two USB programmers.

4.4.1 Running the Code

- Copy the code presented below [9].

[8]This content is available online at <http://cnx.org/content/m21596/1.3/>.

[9]Alternatively, this code is available in the downloadable source code (<http://cnx.org/content/m22470/latest/watteyne_ezwsn_sourcecode.zip
Open `source_code/iar_v4.11/lab_ezwsn.eww` with IAR. The project corresponding to this section is called `txrx_chat`.

- Compile and download the code onto both boards *(Ctrl+D)*.
- Connect each board to a computer using the USB programmer and open PuTTY. Type something and hit enter, the message should appear on the other side's screen.

```
    #include "radios/family1/mrfi_spi.h"
#include "mrfi.h"
uint8_t index_output = 9;
mrfiPacket_t packetToSend;
int main(void)
{
  BSP_Init();
  MRFI_Init();
  P3SEL    |= 0x30;     // P3.4,5 = USCI_A0 TXD/RXD
  UCA0CTL1  = UCSSEL_2; // SMCLK
  UCA0BR0   = 0x41;     // 9600 from 8Mhz
  UCA0BR1   = 0x3;
  UCA0MCTL  = UCBRS_2;
  UCA0CTL1 &= ~UCSWRST; // Initialize USCI state machine
  IE2      |= UCA0RXIE; // Enable USCI_A0 RX interrupt
  MRFI_WakeUp();
  MRFI_RxOn();
  index_output=0;
  __bis_SR_register(GIE+LPM4_bits);
}
void MRFI_RxCompleteISR()
{
  uint8_t i;
  P1OUT ^= 0x02;
  mrfiPacket_t packet;
  MRFI_Receive(&packet);
  char output[] = {"                    "};
  for (i=9;i<29;i++) {
    output[i-9]=packet.frame[i];
    if (packet.frame[i]=='\r') {
      output[i-9]='\n';
      output[i-8]='\r';
    }
  }
  TXString(output, (sizeof output));
}
#pragma vector=USCIAB0RX_VECTOR
__interrupt void USCI0RX_ISR(void)
{
  char rx = UCA0RXBUF;
  uint8_t i;
  packetToSend.frame[index_output]=rx;
  index_output++;
  if (rx=='\r' || index_output==29) {
     packetToSend.frame[0]=28;
     MRFI_Transmit(&packetToSend, MRFI_TX_TYPE_FORCED);
     index_output = 9;
    for(i=9;i<29;i++) {
      packetToSend.frame[i]=' ';
    }
    P1OUT ^= 0x01;
  }
  P1OUT ^= 0x02;
  TXString(&rx, 1);
}
```

Some keys for understanding the code:

- Lines 9-15 configure the UART module used to communicate over the serial port. In particular, line 17 enables interrupts for incoming traffic, i.e. when you write onto PuTTY.
- Lines 21-36. Function MRFI_RxCompleteISR is called when a packet is received. The red LED is switched on (Line 26), and an empty output string is modified with the received characters before being sent.
- Lines 37-55. Function USCI0RX_ISR is called when you enter a character on PuTTY. This 8-bit character is stored at the right byte in the outgoing packet (line 45). When you hit enter or you have type 29 consecutive characters (44), the frame is sent and the output buffer is initialized for subsequent text.

IMPORTANT: If you have multiple motes and multiple computers, all messages typed on one computer will appear on all screen. To reduce the broadcast group, you may wish to switch to another frequency channel using the code described in 4.2. Simple Tx/Rx (Section 4.2).

Chapter 5

Lab 5: RF Measurements

5.1 5.1. The importance of CRC[1]

NOTE: Each packet is appended with a 16-bit Cyclic-Redundancy- Check (**CRC**) which is used to detect accidental alterations of data during transmission. CRC is computed over the data portion of the frame. Upon receiving a packet, the CC2500 recomputes a CRC over the received data and compares that values with the received CRC. A mismatch indicates a corrupted packet; by default, the CC2500 silently drops such a packet. You will disable CRC and see how often corrupted packets occur.

5.1.1 Running the Code

- Copy code presented below [2].
- Compile and download the code onto both boards *(Ctrl+D)*.
- Connect one board to a computer using the USB programmer, and read its output using PuTTY.
- Power the other with the battery packet and press the button. You should read 'abcdefghijklmnopqrstu' on PuTTY.
- Repeat his multiple times by moving away the sending node; you should see some alterations to the output text. If CRC was enabled, these packet would have been dropped by the CC2500 and you wouldn't see them on the screen.
- You may wish to speed things up by changing `P1IFG &= ~0x04;` to `P1IFG |= 0x04;` on the sender's side so one press of the button generates an infinite stream of packets. Similarly, you may wish to reduce its transmission power as explained in 4.3. Continuous Tx/Rx (Section 4.3).

```
    #include "radios/family1/mrfi_spi.h"
#include "mrfi.h"
mrfiPacket_t packetToSend;
int main(void)
{
  BSP_Init();
  P1REN |= 0x04;
  P1IE |= 0x04;
  MRFI_Init();
```

[1]This content is available online at <http://cnx.org/content/m21598/1.4/>.

[2]Alternatively, this code is available in the downloadable source code (<http://cnx.org/content/m22470/latest/watteyne_ezwsn_sourcecode.zip
Open `source_code/iar_v4.11/lab_ezwsn.eww` with IAR. The project corresponding to this section is called `txrx_crc`.

```
  mrfiSpiWriteReg(PATABLE,0xBB);
  mrfiSpiWriteReg(PKTCTRL0,0x41);
  P3SEL    |= 0x30;
  UCA0CTL1  = UCSSEL_2;
  UCA0BR0   = 0x41;
  UCA0BR1   = 0x3;
  UCA0MCTL  = UCBRS_2;
  UCA0CTL1 &= ~UCSWRST;
  MRFI_WakeUp();
  MRFI_RxOn();
  __bis_SR_register(GIE+LPM4_bits);
}
void MRFI_RxCompleteISR()
{
  uint8_t i;
  P1OUT ^= 0x02;
  mrfiPacket_t packet;
  MRFI_Receive(&packet);
  char output[] = {"                        \r\n"};
  for (i=9;i<packet.frame[0];i++) {
    output[i-9]=packet.frame[i];
  }
  TXString(output, (sizeof output));
}
#pragma vector=PORT1_VECTOR
__interrupt void Port_1 (void)
{
  P1IFG &= ~0x04;
  char character = 'a';
  uint8_t index;
  mrfiPacket_t packet;
  for (index=9;index<30;index++) {
    packet.frame[index]=character++;
  }
  packet.frame[0]=++index;
  MRFI_Transmit(&packet, MRFI_TX_TYPE_FORCED);
  P1OUT ^= 0x01;
}
```

5.2 5.2. Creating a Spectrum Analyzer to Measure Noise[3]

NOTE: The CC2500 can easily change its operating frequency, through the use of channels. For each of these channels, the CC2500 can sense the level of electro-magnetic noise, in dBm. The goal of this section is to build a spectrum analyzer by continuously plotting noise vs. frequency. We therefore use a Python script running on the host computer.

IMPORTANT: For this section, you will need to have installed all the software. If not, please follow the Installation Instructions (Section).

[3]This content is available online at <http://cnx.org/content/m21599/1.3/>.

5.2.1 Running the Code

- Copy-paste the code presented below into IAR [4], and program a single board; close IAR.
- Copy-paste the python code below into a new file called `txrx_noise.py` in some directory on your computer [5].
- Connect your USB programmer board to the host computer, and find the COMx port it is connected to. Use PuTTY to read from that *x* port; you should see a continuous series of 200 numbers appearing.
- Start an X server onto the host computer using `Xlaunch` you have installed on your computer.
- Start `Cygwin` and go into the folder containing `./txrx_noise.py` using command `cd` ***`path to your folder`*** [6].
- type `export DISPLAY=127.0.0.1:0.0`
- type `cat /dev/comx`. You should see the same content appearing as when using PuTTY. If not, read the COMxport with PuTTY and try again.
- type `cat /dev/comx | ./txrx_noise.py` A window appears which looks like the one depicted below.

```
    #include "mrfi.h"
#include "radios/family1/mrfi_spi.h"
void print_rssi(int8_t rssi)
{
  char output[] = {" 000 "};
  if (rssi<0) {output[0]='-';rssi=-rssi;}
  output[1] = '0'+((rssi/100)%10);
  output[2] = '0'+((rssi/10)%10);
  output[3] = '0'+ (rssi%10);
  TXString(output, (sizeof output)-1);
}
int main(void)
{
  int8_t rssi;
  uint8_t channel;
  BSP_Init();
  MRFI_Init();
  P3SEL    |= 0x30;
  UCA0CTL1  = UCSSEL_2;
  UCA0BR0   = 0x41;
  UCA0BR1   = 0x3;
  UCA0MCTL  = UCBRS_2;
  UCA0CTL1 &= ~UCSWRST;
  MRFI_WakeUp();
  __bis_SR_register(GIE);
  while(1) {
    for (channel=0;channel<200;channel++) {
      MRFI_RxIdle();
      mrfiSpiWriteReg(CHANNR,channel);
      MRFI_RxOn();
      rssi=MRFI_Rssi();
```

[4] Alternatively, this code is available in the downloadable source code (<http://cnx.org/content/m22470/latest/watteyne_ezwsn_sourcecode.zip
Open `source_code/iar_v4.11/lab_ezwsn.eww` with IAR. The project corresponding to this section is called `txrx_noise`.

[5] Alternatively, this file is already present in the downloadable source code (<http://cnx.org/content/m22470/latest/watteyne_ezwsn_sourcecod
in the directory `source_code/c_files/`.

[6] In cygwin, your Windows drive C:\ is called `/cygdrive/c`. Also, use forward slashes. If your folder is located at `C:\Users\Thomas\Desktop\source_code`, in Cygwin, type `cd /c/cygdrive/Users/Thomas/Desktop/source_code`.

```
      print_rssi(rssi);
    }
    TXString("\n",1);
  }
}
void MRFI_RxCompleteISR()
{
}
C code
```

```
    #!/usr/bin/env python
import re, Gnuplot, sys
if __name__ == '__main__':
    print 'display started...'
    g = Gnuplot.Gnuplot(debug=1)
    g('set term x11 title "eZWSN sprectum analyzer"')
    g('set yrange [-110:0]')
    g('set data style linespoints')
    g.xlabel('frequency (MHz)')
    g.ylabel('RSSI (dBm)')
    char = ''
    line = ''
    while 1:
char = sys.stdin.read(1)
        if char == '\n':
    sline = line.split()
    data = []
    for i in range(len(sline)):
try:
sline[i] = int(sline[i])
except:
sline[i] = 0
frequency = 2433.0+(0.2*i)
                data.append([frequency,sline[i]])
    g.plot(data)
    line = ''
        else:
    line = line + charPython code
```

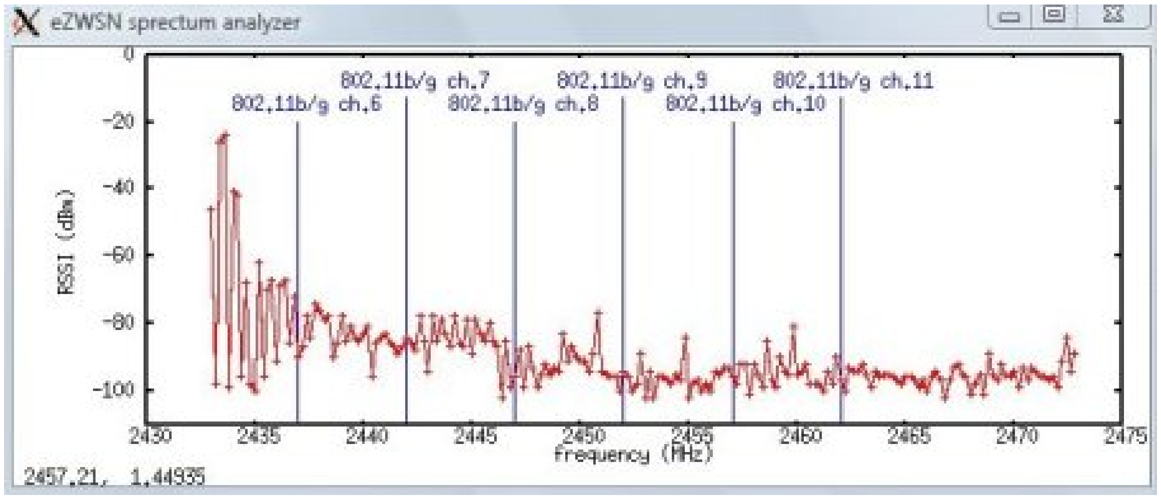


Figure 5.1: Output of the spectrum analyzer in Section 6.2; one transmission is going on on channel 0.

Some keys for understanding the code running on the mote:

- **Line 6.** `print_rssi()` is a function which prints the RSSI value read from the CC2500 onto the serial port which is initialized between lines 18 and 25. `TXString()` is a function provided in `bsp_board.c`.
- **Lines 18-25** initialize the serial port, which enable your code to output lines of text using the `TXString()` function. These lines can then be read using PuTTY.
- **Line 27** instructs the MSP430 to stay in active mode all the time, i.e. not to enter any low power mode.
- **Lines 28-37** is a loop which continuously scans through channels 0-200, recording and outputting the RSSI value. `MRFI_Rssi()` is a function declared in the drivers.

The python script continuously feeds the GNUplot environment with data received from the standard interface. Note that the content of the right COMx port if piped to this python script.

5.2.2 Refresh Rate of the Spectrum Analyzer

- in the code, after line `BSP_Init();`, add line `P2DIR |= 0x08;`
- in the code, after line `TXString("\n",1);`, add line `P2DIR ^= 0x08;`. This toggles extension port `P6` every time the screen gets refreshed.
- using the oscilloscope, measure on extension port `P6` the refresh rate of the frequency analyzer. Make sure to refresh rate is around 1.1s.
- move line `P2DIR ^= 0x08;` right after line `print_rssi(rssi);`. You now measure the time it takes for the mote to sample the noise level on one channel, and move to the other.
- Make sure you measure a value close to 5.5ms.

5.2.3 Testing the Spectrum Analyzer

- Reprogram a second board using the code described in 4.3. Continuous Tx/Rx (Section 4.3), so that it continuously sends data. Start the continuous sending and visualize this transmission on the spectrum analyzer (at channel 0 by default).

- Using the technique described in 4.2. Simple Tx/Rx (Section 4.2), change the operating channel, and see the implications on the spectrum analyzer.
- You can also see the impact of a running microwave oven.

IMPORTANT: When you bring the transmitting mode close to the receiver, you see on the spectrum analyzer that the signal 'leaks' over multiple channels. This is why a standard such as IEEE802.15.4[7] specifies the channel spacing to be 5MHz. We have set the spacing to a much lower value (200kHz) on purpose to see the effect of adjacent channel leakage. In practice, you should use frequencies which are far away from each other.

5.3 5.3. RSSI vs. Distance[8]

NOTE: You will draw the RSSI received as a function of distance between sender and receiver. Because of the random nature of propagation, especially in a closed room, you will see that this relationship is not straightforword to predict. It should be clear that repeating these measurements under different conditions yields very different results.

IMPORTANT: For this section, you will need to have installed all the software. If not, please follow the Installation Instructions (Section).

You will modify the code for the mote taken from 5.2. Creating a Spectrum Analyzer to Measure Noise (Section 5.2) in order to read 200 times the RSSI value *from the same channel*. A python script will then average those read values. To this end:

- Use the code from 5.2. Creating a Spectrum Analyzer to Measure Noise (Section 5.2), but comment out line `mrfiSpiWriteReg(CHANNR,channel);`. The mote will now read the RSSI 200 times on channel 0.
- Reprogram the receiver board and visualize the values outputted on the COMx port using PuTTY (see 1.3. Read directly from COM Port (Section 1.3)). We are interested in calculating the average value of each 200 point line.
- Copy-paste the code presented below to a new file called `txrx_rssi_dist.py` [9];
- in Cygwin, run the python script `cat /dev/comx | ./txrx_rssi_dist.py` [10]. This script outputs the average value of the RSSI 200 readings.
- Reprogram a second board using the code described in 4.3. Continuous Tx/Rx (Section 4.3), so that it continuously sends data, on channel 0 and with a transmission power of 0dBm.
- Start the continuous sending and see how the RSSI decreases as the transmitter is moved away from the receiver.

```
    #!/usr/bin/env python
import sys
if __name__ == '__main__':
    char = ''
    line = ''
    while 1:
```

[7]http://ieeexplore.ieee.org/xpl/standardstoc.jsp?isnumber=35824
[8]This content is available online at <http://cnx.org/content/m21600/1.5/>.
[9]Alternatively, this file is already present in the downloadable source code (<http://cnx.org/content/m22470/latest/watteyne_ezwsn_sourcecod
in the directory `source_code/c_files/`.
[10]In Cygwin, you should first go to the folder containing `txrx_rssi_dist.py`, see 5.2. Creating a Spectrum Analyzer to Measure Noise (Section 5.2).

```
char = sys.stdin.read(1)
        if char == '\n':
    sline = line.split()
    sumsum = 0
    counter = 0
    for i in range(len(sline)):
try:
sline[i] = int(sline[i])
sumsum += sline[i]
counter += 1
except:
sline[i] = 0
    result = sumsum/counter
    print result
    line = ''
        else:
    line = line + char
```

The figure below plots the evolution of RSSI as transmitter and receiver boards are parted away in three different directions. Because these measurements are not done in an infinite open space, shadowing and fading effects introduce randomness into the relationship between RSSI and distance.

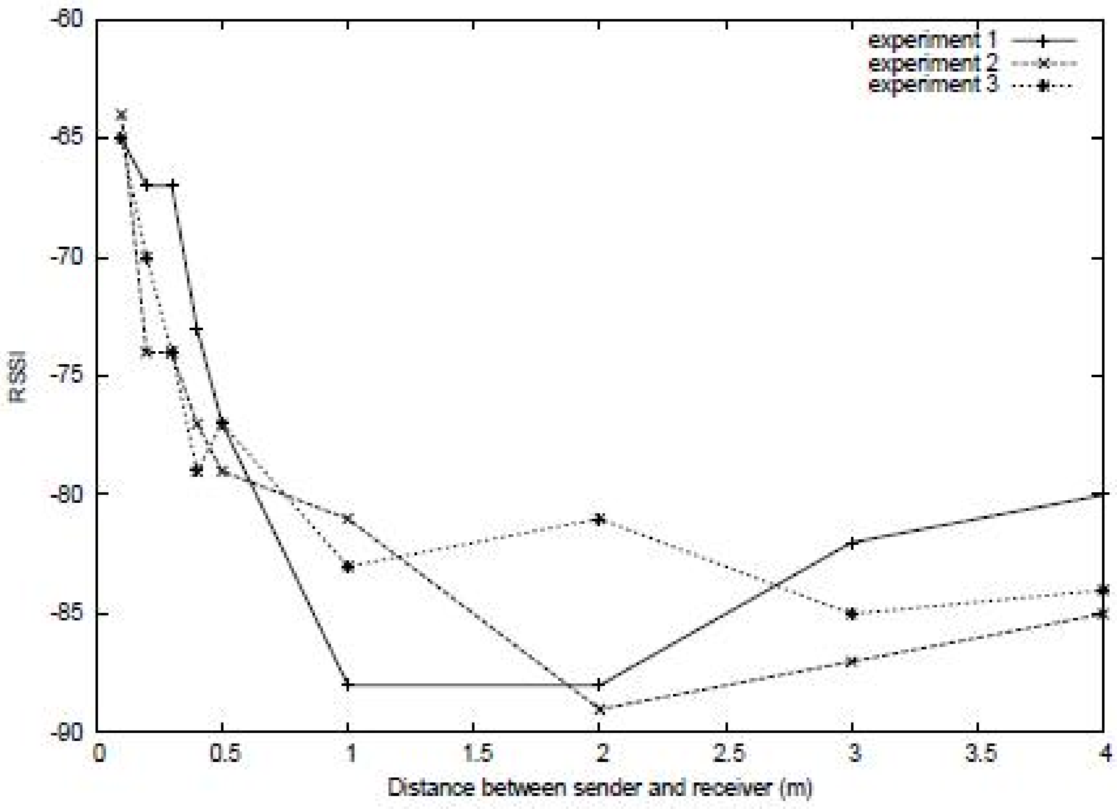


Figure 5.2: Evolution of RSSI with distance.

IMPORTANT: RSSI can not be predicted knowing the distance. This means that you can not know for sure two nodes are able to communicate by just the distance to one another. Similarly, distance can not be predicted knowing the RSSI.

5.4 5.4. Channel Success Probability[11]

NOTE: You have seen that the strength of a link can not be easily predicted as a function of the distance between sender and receiver, especially indoors. You will now discover the relationship between the RSSI of a link and its probability. We define link probability as the portion of the messages sent by the sender which are successfully received at the receiver. You will see that link probability is strongly correlated to the RSSI.

The experimental setup goes as follows. You will use to boards, a sender and a receiver. The sender continuously sends bursts of 100 messages, each containing a counter which increases from 1 to 100. Out of those 100 sent messages, the receiver may only receive 40. It will count the number of received messages until the counter reaches 100, or until the counter loops back to a smaller value. When this happens, the receiver outputs the number of received messages, i.e. the probability of the link.

At the same time as the receiver counts the number of the received messages, it calculates the average RSSI over those 100 messages, which it outputs together with the link probability. Finally, to allow for the receiver to output these statistics (which takes some time), after each burst of 100 messages, the sender sends 20 messages with counter set to 101.

To implement this:

- Reprogram two boards with the code taken from the listing below [12].
- Attach one board to the host computer, and read the output from COMx using PuTTY;
- Power the second board from the battery unit and press the button; this will cause that board to transmit the bursts of 100 messages. You can read the statistics using PuTTY.
- Reprogram the sender node by changing line `P1IFG &= ~0x04;` by `P1IFG |= 0x04;`. This causes a continuous stream of messages. Power the newly programmed node on the battery pack and press the button; the red LED flashes.
- On the receiver side, verify with PuTTY that you are receiving continuously. Close PuTTY.
- Open Cygwin, and enter `cat /dev/comx > probability.txt`. This logs the output in a file.
- Walk away with either node to record data for low RSSI values. Press *(Ctrl+C)* in Cygwin to stop the logging process.
- Start `XLaunch`. In Cygwin, type `export DISPLAY=127.0.0.1:0.0`.
- In Cygwin, enter `gnuplot`, then `plot "probability.txt" using 1:2`. This plots the probability as a function of the RSSI, make sure to obtain a graph similar to the one in the figure below.

```
    #include "mrfi.h"
uint8_t counter, num_received, bool_counting;
int16_t cumulative_rssi;
mrfiPacket_t packet;
void print_probability(int16_t cumulative_rssi, uint8_t number)
{
  char output[] = {" 000 0.00\n"};
  if (cumulative_rssi<0) {
    output[0]='-';
    cumulative_rssi=-cumulative_rssi;
```

[11]This content is available online at <http://cnx.org/content/m21601/1.4/>.

[12]Alternatively, this code is available in the downloadable source code (<http://cnx.org/content/m22470/latest/watteyne_ezwsn_sourcecode.zip
Open `source_code/iar_v4.11/lab_ezwsn.eww` with IAR. The project corresponding to this section is called `txrx_probability`.

```
  }
  output[1] = '0'+((cumulative_rssi/100)%10);
  output[2] = '0'+((cumulative_rssi/10)%10);
  output[3] = '0'+ (cumulative_rssi%10);
  output[5] = '0'+((number/100)%10);
  output[7] = '0'+((number/10)%10);
  output[8] = '0'+ (number%10);
  TXString(output, (sizeof output)-1);
}
int main(void)
{
  BSP_Init();
  P1REN |= 0x04;
  P1IE  |= 0x04;
  MRFI_Init();
  P3SEL    |= 0x30;
  UCA0CTL1  = UCSSEL_2;
  UCA0BR0   = 0x41;
  UCA0BR1   = 0x3;
  UCA0MCTL  = UCBRS_2;
  UCA0CTL1 &= ~UCSWRST;
  MRFI_WakeUp();
  MRFI_RxOn();
  __bis_SR_register(GIE+LPM4_bits);
}
void MRFI_RxCompleteISR()
{
  P1OUT ^= 0x02;
  MRFI_Receive(&packet);
  counter = packet.frame[9];
  if (counter==101) {
    if (bool_counting == 1) {
      print_probability(cumulative_rssi/num_received,num_received);
    }
    bool_counting=0;
    num_received=0;
    cumulative_rssi=0;
  } else {
    bool_counting=1;
    num_received++;
    cumulative_rssi+=Mrfi_CalculateRssi(packet.rxMetrics[0]);
  }
}
#pragma vector=PORT1_VECTOR
__interrupt void interrupt_button (void)
{
  P1IFG &= ~0x04;
  P1OUT ^= 0x01;
  mrfiPacket_t packet;
  packet.frame[0]=8+3;
  for (counter=1;counter<101;counter++){
      packet.frame[9]=counter;
      MRFI_Transmit(&packet, MRFI_TX_TYPE_FORCED);
   }
   for (counter=0;counter<20;counter++){
      packet.frame[9]=101;
      MRFI_Transmit(&packet, MRFI_TX_TYPE_FORCED);
   }
}
```

Some keys to understand the code:

- **Line 55.** When the button is pressed, the board continuously sends bursts of 100 messages followed by 20 "guard" messages.
- **Line 61.** The format of a packet is presented in 4.2. Simple Tx/Rx (Section 4.2). `packet.frame[0]` is the length field, which sets the payload length at 3 bytes (minimum accepted value).
- **Line 63.** `packet.frame[9]` is the first byte of the payload.
- **Line 43.** `bool_counting` indicates whether the statistics have already been printed out (without this semaphore, as there are 20 guard messages, the statistics would be printed out 20 times)

As shown in hte figure below, link probability is closely correlated to RSSI. The theory tells us that, at very low RSSI, link probability is very close to 0; at very high RSSI it is close to 1. Between those extremes, there is a linear slope, shown as an overlay in the figure below.

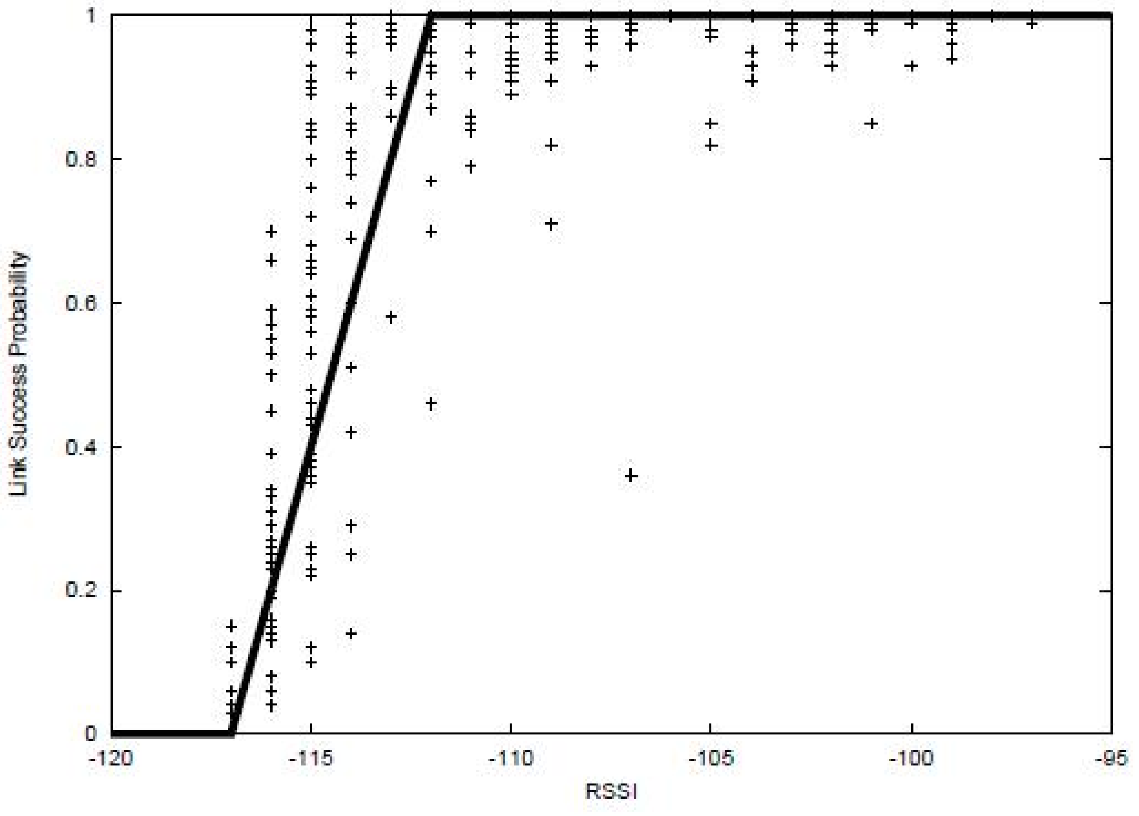


Figure 5.3

Chapter 6

Lab 6: Implementing A Preamble Sampling MAC Protocol

6.1 6.1. An Introduction to Preamble Sampling[1]

> NOTE: Preamble-sampling is a technique used to reduce the energy consumption of the MAC protocol in a wireless node. The goal of this section is to measure the energy consumption of the transmitting and receiving nodes, and to predict the lifetime of a node when powered on two AAA/LR03 batteries.

Nodes using preamble-sampling are not synchronized. Instead, nodes periodically listen for a very short time (called Clear-Channel-Assessment, or **CCA**) to decide whether a transmission is ongoing. We call check interval (**CI**) the amount of time a node waits between two successive CCAs. The sender needs to make sure the receiver node is awake before sending data; it prepends a (long) preamble to its data. By having the preamble at least as long as the wake-up period, the sender is certain that the receiver will hear it and be awake for receiving the data.

The figure below is a chronograph depicting the radio state of node S and its three neighbors `A`, `B` and `C`. A box above/under a vertical line means the node's radio is transmitting/ receiving, respectively. No box means the radio is off. All nodes sample the channel for Dcca seconds every CI seconds.

[1]This content is available online at <http://cnx.org/content/m21602/1.2/>.

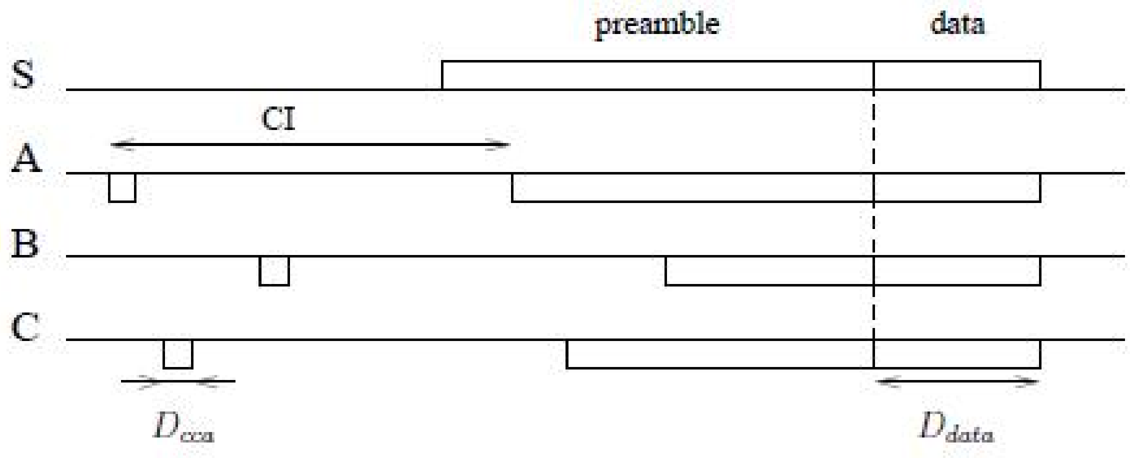


Figure 6.1: Basic preamble-sampling.

In this Section, you will implement a simplified version of preamble sampling:

- there is no data;
- the preamble is cut into a series of micro-frames. Each micro-frame contains a counter indicating how many micro-frames still remain. A micro-frame is sent every `Tmf` seconds, and lasts for `Dmf`.

6.2 6.2. Running the code[2]

The experimental setting goes as follows. You will have two boards, both sampling periodically sampling the channel. When they are not sampling the channel, the CC2500 is switched off and the MSP enters low-power mode. When you press a button on one board, it sends a preamble cut into 50 micro-frames; the receiver hears a micro-frame and keeps listening until it hears the last one.

To this end:

- Program two board with the code taken from listing below [3]; one will be the transmitter, the other the receiver.
- Plug in one of the board into the computer and use PuTTY to read from its COMx port; this will be the receiver.
- press on the transmitter's button, you should read `03 02 01` on your screen.

```
    #include "mrfi.h"
#include "radios/family1/mrfi_spi.h"
void start_slow_timeout()
{
  TACTL|=TACLR; TACCTL0=CCIE; TACTL=TASSEL_1+MC_1;
}
void stop_slow_timeout()
```

[2]This content is available online at <http://cnx.org/content/m21603/1.4/>.

[3]Alternatively, this code is available in the downloadable source code (<http://cnx.org/content/m22470/latest/watteyne_ezwsn_sourcecode.zip Open `source_code/iar_v4.11/lab_ezwsn.eww` with IAR. The project corresponding to this section is called `txrx_preamble_msp`.

```
{
  TACTL=MC_0; TACCTL0=0;
}
void start_fast_timeout()
{
  TBCTL|=TBCLR; TBCCTL0=CCIE; TBCTL=TBSSEL_2+MC_1;
}
void stop_fast_timeout()
{
  TBCTL=MC_0; TBCCTL0=0;
}
void print_counter(int8_t counter)
{
  char output[] = {"   "};
  output[0] = '0'+((counter/10)%10);
  output[1] = '0'+ (counter%10);
  TXString(output, (sizeof output)-1);
}
int main(void)
{
  BSP_Init();
  P1REN |= 0x04;
  P1IE |= 0x04;
  MRFI_Init();
  P3SEL    |= 0x30;     // P3.4,5 = USCI_A0 TXD/RXD
  UCA0CTL1  = UCSSEL_2; // SMCLK
  UCA0BR0   = 0x41;     // 9600 from 8Mhz
  UCA0BR1   = 0x3;
  UCA0MCTL  = UCBRS_2;
  UCA0CTL1 &= ~UCSWRST; // Initialize USCI state machine
  IE2      |= UCA0RXIE; // Enable USCI_A0 RX interrupt
  BCSCTL3 |= LFXT1S_2; TACTL=MC_0; TACCTL0=0; TACCR0=1060;  //slow timeout
  TBCTL=MC_0; TBCCTL0=0; TBCCR0=31781;                      //fast timeout
  start_slow_timeout();
  __bis_SR_register(GIE+LPM3_bits);
}
void MRFI_RxCompleteISR()
{
  mrfiPacket_t packet;
  stop_fast_timeout();
  stop_slow_timeout();
  MRFI_Receive(&packet);
  if (packet.frame[9]<4) {
    print_counter(packet.frame[9]);
    start_slow_timeout();
  } else {
    MRFI_WakeUp();
    MRFI_RxOn();
  }
}
#pragma vector=PORT1_VECTOR
```

```
__interrupt void interrupt_button (void)
{
  P1IFG &= ~0x04;
  uint8_t counter;
  mrfiPacket_t packet;
  packet.frame[0]=8+20;
  MRFI_WakeUp();
  for (counter=50;counter>=1;counter--) {
     packet.frame[9]=counter;
     MRFI_Transmit(&packet, MRFI_TX_TYPE_FORCED);
  }
}
#pragma vector=TIMERA0_VECTOR
__interrupt void interrupt_slow_timeout (void)
{
  MRFI_WakeUp();
  MRFI_RxOn();
  start_fast_timeout();
  __bic_SR_register_on_exit(SCG1+SCG0);
}
#pragma vector=TIMERB0_VECTOR
__interrupt void interrupt_fast_timeout (void)
{
  stop_fast_timeout();
  MRFI_Sleep();
  __bis_SR_register_on_exit(LPM3_bits);
}
```

Some keys for understanding the code:

- The microcontroller handles two timeouts, one for measuring `CI`, the other for `Dcca`. Those timeouts are sourced by two different clocks: a fast and accurate clock for `Dcca`; a slower, less accurate but extremely energy-efficient clock for `CI`. The fast clock is the Digitally Controlled Oscillator (**DCO** on Timer A) while the very-low-power, low-frequency oscillator (**VLO** on Timer B) is the slow clock. Because `CI` is triggered by the slow clock, that clock stays on all the time. Only when the slow timeout expires does the microcontroller start the fast clock to clock the fast timeout (`Dcca`); and stops it when that expires. The radio is on only during `Dcca`.
- **Line 38** initializes the slow timeout on Timer A
- **Line 39** initializes the slow timeout on Timer B
- **Line 42.** Because the slow clock runs all the time, the board can only enter `LPM3` which leaves the VLO clock running.
- **Line 71.** Every time the slow timeout triggers, the CC2500 is switched on in Rx mode (lines 74-75); the fast timeout is started (line 76), and because it is clocked by the DCO, `LPM0` mode is entered which leaves the DCO running (line 77).
- **Line 79.** When the fast timeout expires, this timeout is stopped (line 82), the CC2500 is put to sleep (line 83) and the `LPM3` mode is resumed (line 84).
- **Line 58.** When the button is pressed, the board transmits 50 micro-frames, each containing a decrementing counter.

6.3 6.3. Timing issues: length of the preamble[4]

- Use the oscilloscope to visualize the energy consumed by a board, you can clearly see the periodic wakeup of the CC2500 (see figure below, lower part).

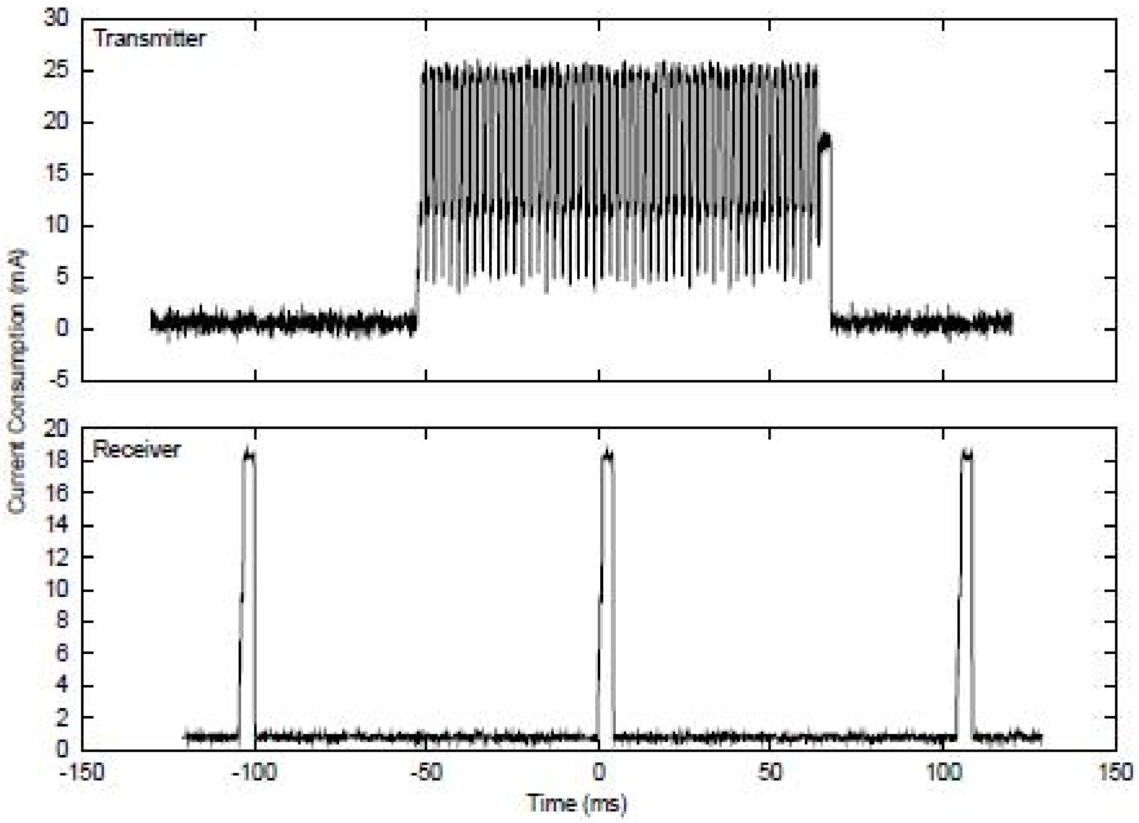


Figure 6.2: Energy consumed by the transmitter and the receiver in preamble sampling. To function, the length of the preamble needs to be larger than the check interval CI.

- You can see that the time between two successive wake-ups is `CI=104ms`. Push the button and capture the energy consumed when the board is in transmit mode (see figure, upper part). You can see the series on 50 microframes sent.
- By zooming in, you can see that one micro-frame is sent every `Dcca=3.24ms`. Hence, the length of the whole preamble is `50*3.24=162ms`, which is larger that the check interval.

6.4 6.4. Timing issues: first micro-frame heard[5]

Because the receiver periodically samples the channel, the micro-frame of a preamble it hears first can be any of the 50. You will now visualize which micro-frame is heard first. To this end:

[4]This content is available online at <http://cnx.org/content/m21604/1.2/>.
[5]This content is available online at <http://cnx.org/content/m21605/1.2/>.

- move line `print_counter(packet.frame[9]);` up two lines (right after `MRFI_Receive(&packet);`), so that the counter value gets printed for every microframe received;
- after reprogramming a board, connect it to the host computer and read its COMx port with PuTTY;
- press on the second boards button, you should now read a series of decrementing numbers such as:

  `30 29 28 25 24 23 20 19 18 15 14 13 10 09 08 05 04 03` [6]

  The first number you read is the first micro-frame received (here 30), which is not necessarily the first one sent (here 50).

6.5 6.5. Measuring the Energy Consumption[7]

- Use the oscilloscope to visualize the current drawn by a receiver board.
- Zoom in onto a wake-up period and use the averaging function on your oscilloscope (on the TDS2022B press acquire, average) to average over as many samples as possible.
- Freeze the screen once this is done, you should obtain a screen close to the following one:

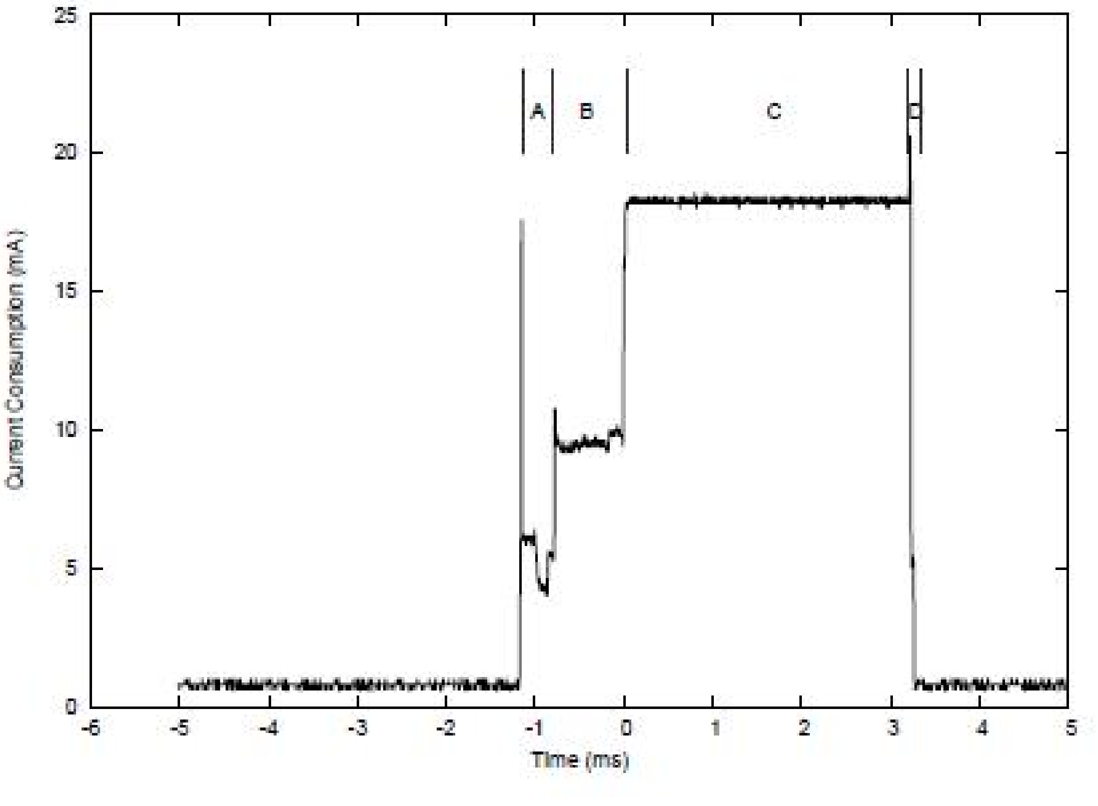


Figure 6.3: Energy consumption measured when sampling the channel.

- You easily see the different phases when the CC2500 is turned on, the current drawn during these phases and their length (see Table below).

[6]The sequence may not be continuous because the board spends some time writing on its serial port, and thus misses a few micro-frames.
[7]This content is available online at <http://cnx.org/content/m21606/1.2/>.

- As shown in the following table, you can calculate the average current consumption of the board when it is listening, and the expected lifetime.

	Phase	duration	av. current
	idle	99.6ms	0.788mA
A	microcontroller startup	0.396ms	5.45mA
B	radio frequency calibration	0.772ms	9.55mA
C	reception mode	3.24ms	18.3mA
D	entering sleep mode	0.032ms	5.00mA

Table 6.1: Duration and average current consumption of the different phases observed in the figure. Measurements averaged over 128 samples.

- One can calculate that the average average current consumption is 1.450mA, hence an expected lifetime of about one month on 1000mAh.

IMPORTANT: By carefully tuning the value of `CI` and `Dcca` (which was not done here for simplicity), the energy consumption of a preamble-sampling MAC protocol can be brought down below .1%.

Chapter 7

Lab 7: A Complete WSN Example

7.1 7.1. Communication Stack[1]

> IMPORTANT: The source code is too long to be put on this web page. You will have to download[2] it. Open `source_code/iar_v4.11/lab_ezwsn.eww` with IAR. The project corresponding to this section is called `txrx_wsn`.

Project `txrx_wsn` is a complete WSN example which implements a complete communication stack for WSNs, using gradient multi-hop routing. A gradient routing protocol assigns a scalar value to each node, which we call its *height*. Heights are assigned in such a way that they increase with distance to a central node. Distance is calculated using a cumulative cost function based here on hop count. The forwarding process selects the next hop as the neighbor which offers the largest gradient, i.e.the neighbor with lowest height.

In the implemented protocol stack, the application layer generates sensed data to be sent to a sink node, by using on-board Analog-to-Digital Conversion. The routing layer is responsible for updating the node's `myHeight`; the MAC layer performs on-demand neighbor discovery and uses preamble sampling for energy-efficiency.

The execution timeline of the implemented protocol is presented in the figure below, for an example topology of 3 nodes. By default, nodes perform preamble sampling. When a node wants to send a message (here `A`), it starts by sending a preamble as long as the check interval (`CI`) to make sure all neighbors hear that preamble. For efficient handling by a packet radio, the preamble is cut into a series of micro-frames `UF`, each containing a counter indicating the number of `UF` still to come. Upon hearing a `UF`, a receiving node turns its radio off and sets a timer to switch into receive mode after the last `UF`. At that moment, the sender indicates the duration of the neighbor announcement window to follow in a `CW` packet.

[1]This content is available online at <http://cnx.org/content/m22473/1.3/>.
[2]http://cnx.org/content/m22470/latest/watteyne_ezwsn_sourcecode.zip

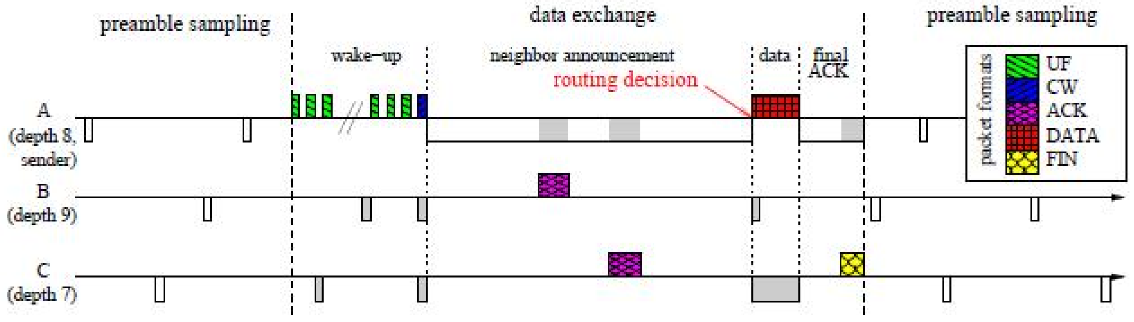


Figure 7.1: Timeline illustrating the execution of the protocol stack. The x-axis represents time; a box above the line indicates that the radio is transmitting; a gray/white box under the axes means that the radio in receiving/idle listening, resp.; no box means the radio is turned off.

Receivers choose a random backoff for sending an `ACK` message inside the neighbor announcement window and sleep the rest of the time; the sender listens for the complete announcement window and populates the initially empty neighbor table as it receives `ACK` messages.

After the neighbor announcement window, the sender updates its `myHeight` by the minimum value of its neighbors', incremented by one, and select its neighbor with smallest `Height`. It inserts this information into the `DATA` packet header which it transmits. The destination node receives the whole packet while the non-destination neighbor switches to sleep after the header. The destination replies with a final acknowledgment `FIN`; all nodes resume preamble sampling.

7.2 7.2. Run a Multi-Hop WSN[3]

The source code is contained in project `txrx_wsn` [4]. By default, each node transmits `every 5+rand(5)` seconds at `-16dBm`, on channel `0`. The sink node prints the content of the received packets.

Project `txrx_wsn` can not be compiled with the free edition of IAR because code size it larger than 4kB. Projects `txrx_wsn_node` and `txrx_wsn_sink` contain already compiled binary code for nodes and sink, respectively [5].

- Program only one board for the whole network using project `txrx_wsn_sink`, the others using project `txrx_wsn_node`.
- Use PuTTY to read from the sink node and switch on the other motes.
- enter command `cat /dev/comx | ./txrx_wsn.py` on the computer hosting the sink node. You should see a graph similar to the one below.

[3]This content is available online at <http://cnx.org/content/m22476/1.2/>.
[4]This code is available in the downloadable source code (<http://cnx.org/content/m22470/latest/watteyne_ezwsn_sourcecode.zip>). Open `source_code/iar_v4.11/lab_ezwsn.eww` with IAR. The project corresponding to this section is called `txrx_wsn`.
[5]If you have a full edition of IAR, you can use project `txrx_wsn` directly. You will need to modify boolean `IS_SINK_NODE` in source file `onehopmac.h` to program either nodes or sink.

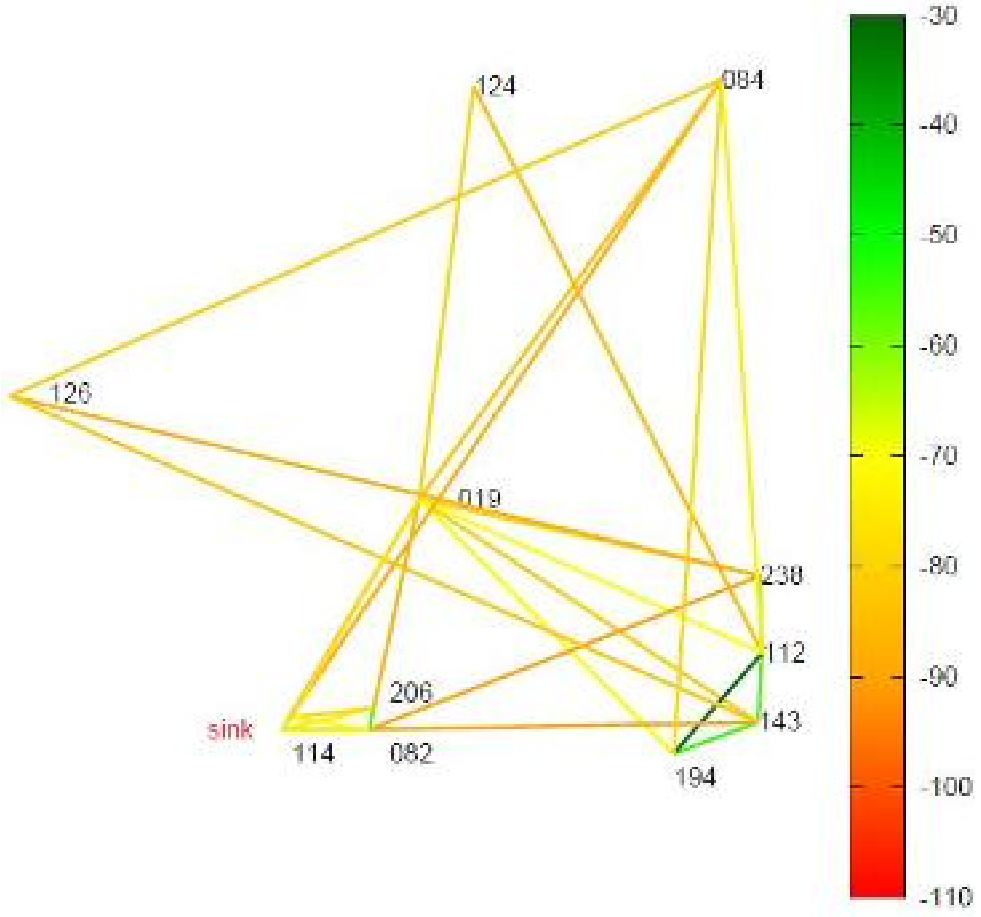


Figure 7.2: An example outputted graph with 12 nodes. Links interconnect neighbors; colors indicate link quality in dBm.

- You may move nodes away from each other to obtain a multi-hop graph.

Attributions

Module: "4.2. Simple Tx/Rx"
By: Thomas Watteyne
URL: http://cnx.org/content/m21594/1.3/
Pages: 30-31
Copyright: Thomas Watteyne
License: http://creativecommons.org/licenses/by/3.0/

Module: "4.3. Continuous Tx/Rx"
By: Thomas Watteyne
URL: http://cnx.org/content/m21595/1.2/
Pages: 32-34
Copyright: Thomas Watteyne
License: http://creativecommons.org/licenses/by/3.0/

Module: "4.4. Wireless Chat"
By: Thomas Watteyne
URL: http://cnx.org/content/m21596/1.3/
Pages: 34-36
Copyright: Thomas Watteyne
License: http://creativecommons.org/licenses/by/3.0/

Module: "5.1. The importance of CRC"
By: Thomas Watteyne
URL: http://cnx.org/content/m21598/1.4/
Pages: 37-38
Copyright: Thomas Watteyne
License: http://creativecommons.org/licenses/by/3.0/

Module: "5.2. Creating a Spectrum Analyzer to Measure Noise"
By: Thomas Watteyne
URL: http://cnx.org/content/m21599/1.3/
Pages: 38-42
Copyright: Thomas Watteyne
License: http://creativecommons.org/licenses/by/3.0/

Module: "5.3. RSSI vs. Distance"
By: Thomas Watteyne
URL: http://cnx.org/content/m21600/1.5/
Pages: 42-44
Copyright: Thomas Watteyne
License: http://creativecommons.org/licenses/by/3.0/

Module: "5.4. Channel Success Probability"
By: Thomas Watteyne
URL: http://cnx.org/content/m21601/1.4/
Pages: 44-47
Copyright: Thomas Watteyne
License: http://creativecommons.org/licenses/by/3.0/

eZWSN: Experimenting with Wireless Sensor Networks using the eZ430-RF2500
Through this series of fully hands-on labs, you will explore Wireless Sensor Networks from an experimental point of view. No prior knowledge is needed, other than (very) basic C and some wireless networking theory. After this tutorial, you will be able to program a microcontroller (in this case an MSP430) and a radio chip (in this case a CC2500), and to implement state-of-the-art communication protocols for Wireless Sensor Networks.