eZWSN - Exploring Wireless Sensor Networking
Lab Version

Thomas Watteyne

Berkeley Sensor & Actuator Center, UC Berkeley, USA.
watteyne@eecs.berkeley.edu

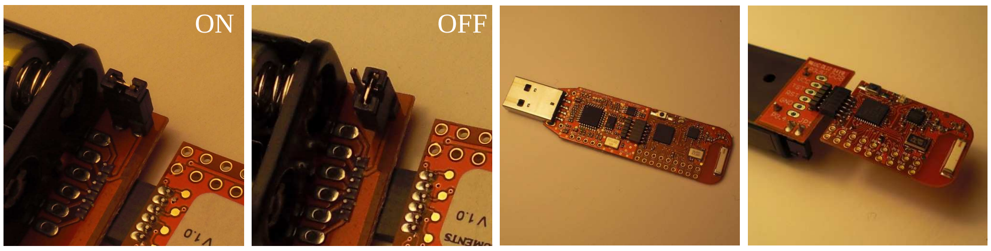
Goal

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. No prior knowledge is needed, other than (very) basic C and some wireless networking theory. This tutorial is meant to be fully hands-on.

Key words: Embedded Programming, Wireless Sensor Networks, RF measurements, Medium Access.

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 !**



16 April 2009

1 Prerequisites & Timeline

For this tutorial, you need:

- an MSP430 eZ430-RF2500 Development Tool kit, i.e. two target boards, a USB programmer and a battery unit [1] ;
- 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 [2] ;
- ideally, an oscilloscope like the Tektronix TDS 210 with a 1Ω resistor mounted in an open jumper.

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

- **hour 1.** Discover the development environment, run a basic demo (p. 3).
- **hour 2.** Discover the eZ430-RF2500 board and its components (p. 6).
- **hours 3-4.** Simple examples for the MSP430 (p. 12).
- **hour 5.** Simple examples for the CC2500 (p. 19).
- **hours 6-7.** RF measurements (p. 25).
- **hour 8.** Implementing a preamble sampling MAC protocol (p. 31).

Keep 30 minutes at the end of the lab to build a complete WSN presented in Section 8. **Before starting the lab, read Section 3 completely, and have a look at the structure of the document.**

[1] This set of 2 motes is manufactured and sold by Texas Instruments for $49.

[2] If this software is not installed, refer to the Installation Instructions provided in a separate document

2 The Development Environment (hour 1)

This section details the environment you will use throughout this tutorial. You will use this environment to run a simple demo, to measure current values using the oscilloscope and to read serial output using the host computer.

2.1 eZ430-RF2500 Sensor Monitor Demo

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.

2.1.1 Running the Code

- Download the source code of the lab[3] on your Desktop, unzip, and create folder `lab_ezwsn` on your folder. You will only use/change code from that folder; you can remove it at the end of the tutorial.
- Enter the `tidemo/` folder.
- 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.

2.1.2 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.

[3] `http://www.eecs.berkeley.edu/˜watteyne/290Q/source_code.zip`

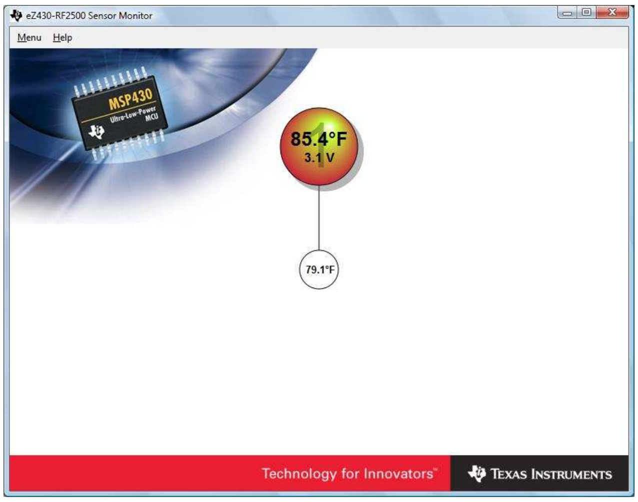


Fig. 1. Section 2.1 is done when you have a similar window on your screen.

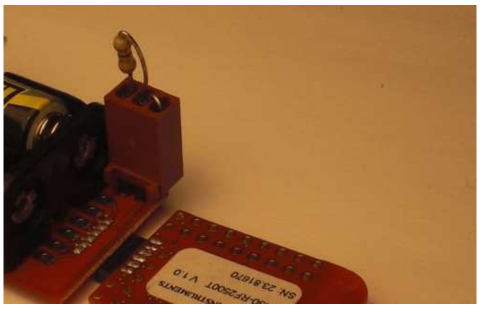


Fig. 2. Use the 1Ω jumper to switch on the target board...

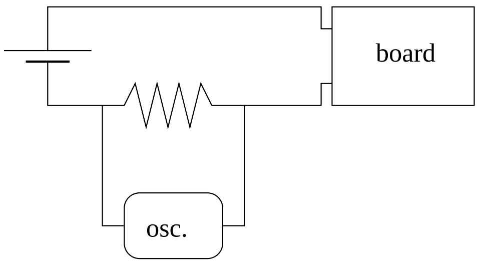


Fig. 3. ...and read the current consumption of the board.

- You see the temperature of the access point. Switch on your end Device, which appears on the screen (see Fig. 1).

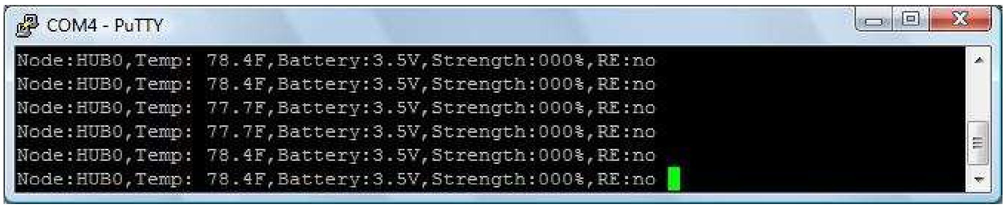
2.2 *Oscilloscope: Read the Current Consumption*

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 1Ω 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 1Ω resistor (Fig. 2).
- Connect the oscilloscope onto the resistor as in Fig. 3. What you read is the current consumption in amps of the board ($U = R \cdot I$, with $R = 1\Omega$).
- Zoom onto a single wake-up period. Use the averaging function of the oscilloscope to obtain a clear reading. Fill in Table 1 identifying the different phases.
- We can assume the boards are powered by two AAA batteries with a capacity

Phase	duration	av. current
sleep		
radio is switched on		
IDLE mode		
RX mode		
TX mode		
Average current consumption		
Lifetime with a 1000mA*hr battery		

Table 1
Duration and average current consumption of the different phases observed.



```
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
```

Fig. 4. Section 2.3 is done when you see a screen similar to this one.

of 1000 mA*hr, under the hypothetical condition in which the batteries hold their voltage ideally until their capacity is exhausted. Calculate the approximate lifetime in Table. 1.

2.3 Read Directly From COM Port

So far, the Sensor Monitor 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 & 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 one depicted in Fig. 4.

3 eZ430-RF2500 Board and its Components (hour 2)

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.

3.1 Crash Course on the MSP430f2274

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.

3.1.1 Operation of the MSP430

Fig. 5 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 [4] . 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.

3.1.2 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.

[4] Strictly speaking, instructions can take a couple of CPU cycles to execute

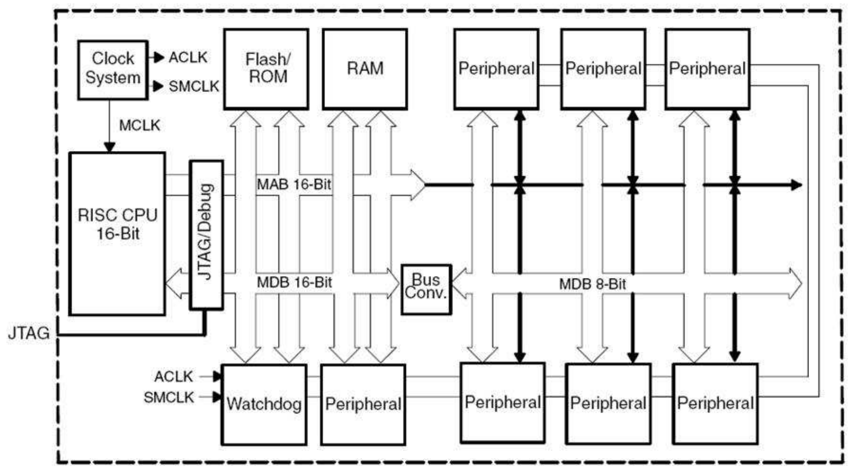


Fig. 5. The internal architecture of the MSP430.

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

3.1.3 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, the MSP430 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 section 4: when pushing the button (4.3), timer interrupts (4.4) and when receiving a packet at the radio (5.2).

3.1.4 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 [5], 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 4.4 to have a LED flash at a given rate.

3.1.5 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 P$x.y$, 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 P$x.y$; output if P$x.y$=1, input if P$x.y$=0;
- PxOUT.y sets the state of port P$x.y$ when set as output;
- PxIN.y reads the state of port P$x.y$ 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 [6] means that pins P1.1 through P1.4 are input, while P1.5 through P1.8 are outputs. To set/reset a specific pin, you need to use the binary operators presented in Fig. 6.

Note that most of the 32 digital pins can also be used for specific functions (SPI interface, input for Analog-to-Digital conversion, …), see [2] for details.

3.1.6 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 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 [1]).

[5] Strictly speaking, timers can be configured to count in up, down, or up/down modes, see [1]

[6] 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.

A	=	0b01101001		
~A	=	0b**10010110**		
A	|=	0b000000**1**0	⇒	A=0b011010**1**1
A	&=	~0b0000**1**000	⇒	A=0b0110**0**001
A	^=	0b**1**000**1**000	⇒	A=0b1110**0**001
A	<<	2	⇒	A=0b101001**00**
A	>>	2	⇒	A=0b**00**011010

Fig. 6. Binary operators used to set/reset individual bits.

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 you `main` function:

```
__bis_SR_register(LPM3_bits);
```

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 [1] and [2] (in that order).

3.2 Crash Course on the CC2500

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 [3].

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, which can be found online for free. We will use these off-the-shelf drivers in this tutorial.

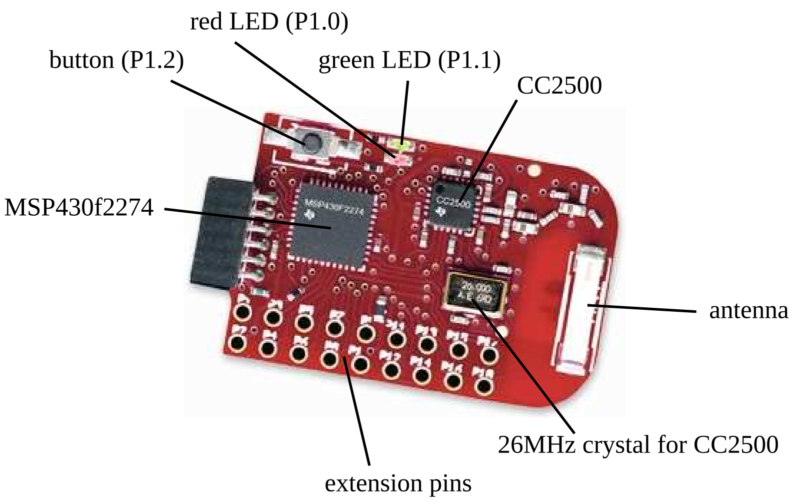


Fig. 7. The components of the eZ430-RF2500.

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 [3] (after having read the documents about the MSP430).

3.3 The eZ430-RF2500 Board

3.3.1 Overview

Fig. 7 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.

3.3.2 Interconnecting the MSP430 with the CC2500

As show in Fig. 8, 6 wires interconnect the MSP430 with the CC2500. 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.

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

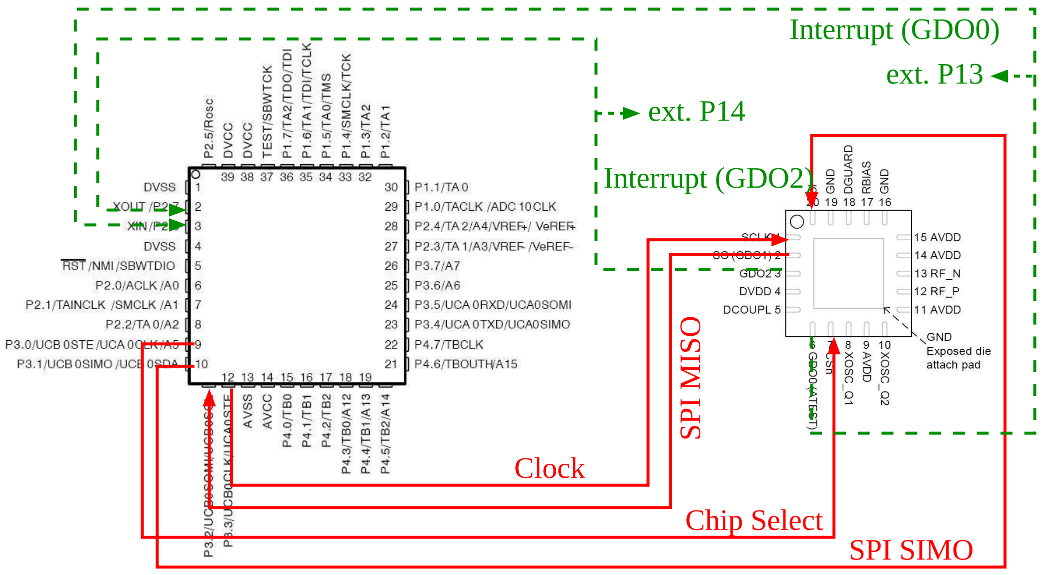


Fig. 8. MSP430 and CC2500 are interconnected by an SPI 4-wire link (plain) and two interrupt lines (dotted).

P14.

Refer to [4] for details about the eZ430-RF2500 platform.

4 Flashing LEDs: Simple Programming Examples for the MSP430 (hours 3-4)

This section will learn you how to use LEDs, interrupts and timers on the MSP430. These are the basic building blocks which you will use in section 5 for communicating with the CC2500. In this section, you will **not** use the CC2500.

4.1 A Steady LED

This example switches on both LEDs at node startup. It shows you how to use the port registers PxDIR and PxOUT. Moreover, you learn how to create a new project in IAR.

4.1.1 Creating a Project in IAR

You will need to repeat these steps each time you create a new project in IAR:

- 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`.

Note that you will find the complete source code by opening `lab_ezwsn.eww`. You will not need to create new projects.

4.1.2 Running the Code

- In IAR, right click on project `led_steady` in the workspace and chose `Set as Active`.
- Compile and download the code onto the board (Ctrl+D).
- Let the code execute (F5), you should now see both LEDs on.

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.

no lEDs (MSP430 running)	Current drawn
red+green LED	
red LED	
green LED	

Table 2
Current consumed by the LEDs (Section 4.1).

- **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 3V). This causes the LEDs, connected to those pins, to light.
- **Line 7** loops, leaving the board running.

4.1.3 Energy Consumption

The goal here is to fill in Table 2:

- 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.

4.2 Active Waiting Loop

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 4.4.

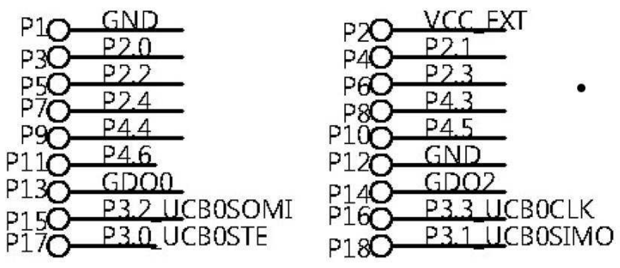


Fig. 9. The extension pins on the eZ430-RF2500 board, taken from [4]. Note that the pins with even number (shown on the right) are located on the edge of the board, and are thus accessible more easily.

4.2.1 Running the Code

- In IAR, right click on project `led_loops` in the workspace and chose `Set as Active`.
- Compile and download the code onto the board (Ctrl+D).
- Let the code execute (F5), both LEDs should blink.

Some keys for understanding the code:

- **Line 9:** the operator $\wedge =$ 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();` cause the MSP430 to do nothing for one cycle.

4.2.2 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 Fig. 9, which is connected to P2.3 on the MSP430.

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 [7]
- 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 and fill in Table 3.

[7] P12 is hard to reach, yet some oscilloscope will not require you to keep the ground all the time. Try un-grounding after a while and, if you're lucky, you'll still read a clear signal.

threshold value for i	**measured toggle duration**
1000	
10000	
20000	
30000	

Table 3
Duration when using an active waiting loop (Section 4.2).

4.3 Button-Driven Toggle Through Interrupts

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.

4.3.1 Running the Code

- In IAR, right click on project `led_button` in the workspace and chose `Set as Active`.
- 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.

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);
- **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).

Mode	current drawn
Active mode (active: CPU and all clock)	
LPM0 mode (active: SMCLK, ACLK; disabled: CPU, MCLK[8])	
LPM3 mode (active: ACLK; disabled: CPU, MCLK, SMCLK)	
LPM4 mode (disabled: CPU and all clock)	

Table 4
Current consumed by the LPM modes (Section 4.3).

4.3.2 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.

The goal here is to fill in Table 4:

- 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 12 which can never be reached.
- Measure the current now, make sure that LEDs are again off.
- repeat with LPM3 and LPM4.

Which mode is more energy-efficient? What prevents us to enter that mode all the time?

4.4 Timer-Driven Toggle Through Timer Interrupts

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.

TACCR0 value	**measured toggle duration**
500	
1000	
10000	
20000	

Table 5
Duration when using timers (Section 4.4).

4.4.1 Running the Code

- In IAR, right click on project `led_timer` in the workspace and chose `Set as Active`.
- Compile and download the code onto the board (Ctrl+D).
- Let the code execute (F5), both LEDs should blink.

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.

4.4.2 Measuring Time

As in Section 4.2, 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; fill in Table 5.

According to Table 5, at what speed does the VLO clock - which clocks ACLK - run [9] ?

[9] In theory, the VLO runs at 12kHz; the exact value depends on the voltage of the batteries and on the temperature.

5 Enabling Wireless Communication (hour 5)

In this section, you will learn how to use the CC2500 by sending packets on a given frequency and at a given transmission power. You will use the oscilloscope to visualize the current consumption under these different cases.

5.1 *Using the Texas Instruments Drivers*

Texas Instruments has developed a set of drivers for the eZ430-RF2500 as part of the simpliciTI project, freely available online. These drivers are also contained in the `source_code/drivers` folder for this tutorial. You will use them without changing the files.

- In IAR, create a new project as instructed in Section 4.1.
- 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:
  ```
    - 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`

Note that you will find the complete source code by opening `lab_ezwsn.eww`. You will not need to create new projects.

5.2 *Simple Tx/Rx*

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.

5.2.1 *Running the Code*

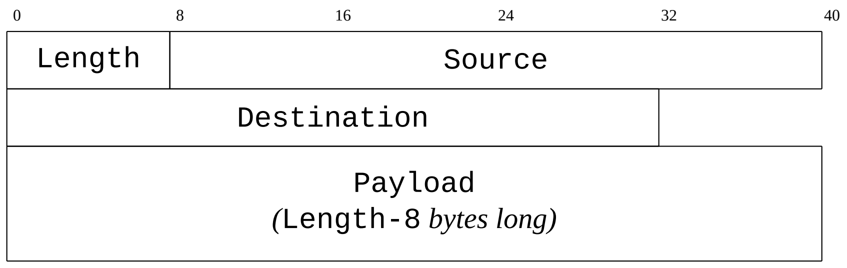
- In IAR, right click on project `txrx_simple` in the workspace and chose `Set as Active`.
- Compile and download the code onto both boards (Ctrl+D).
- Switch both boards on (one with the battery unit, the other with the USB slot of your PC); when you press one's button, the other's red LED should toggle.

The packet format is shown in Table 6 (p.21). A variable of type `mrfiPacket_t` is a structure containing two parts:

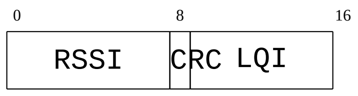
- `packet.frame` is the frame to be transmitted. The first byte is the `Length` of the payload together with source and destination `Address`. With the current driver implementation, addresses are coded on 4 bytes, and the maximum `Payload` length is 20 bytes. By default, the CC2500 does not perform 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



packet.frame



packet.rxMetrics

Table 6
Packet format.

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`[10] is called.

5.2.2 *Choosing a Frequency*

The CC2500 can transmit at any frequency on the ISM band 2400.0-2483.5MHz. 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.

In a lab environment, as you don't want to interfere with other groups, each group needs to pick a different channel. To avoid co-channel interference (a channel may "leak" into its neighboring channels, leading to interference), space frequencies as much as possible.

- add after line 1 the following line. 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 after line 7 the following line, replacing `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

[10] Note that the real interrupt function with the usual `pragma` declaration is contained in the drivers

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

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

You will need to add these lines for all subsequent exercises to isolate you from interference from other groups.

5.3 Continuous Tx/Rx

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.

5.3.1 Running the Code

- Start from the code from the previous Section.
- Comment out line 19; this means that once you push the button, the board will start sending an infinite number of messages. This is because you don't clear the interrupt flag, 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.

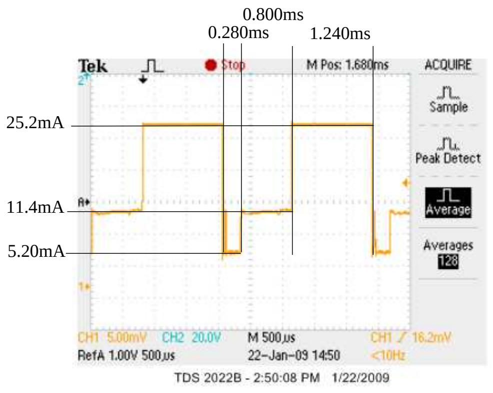
5.3.2 Energy Consumption

- Comment out lines 14 and 23 so as to be sure *not* to measure the energy consumption of the LEDs;
- Use the oscilloscope to measure the energy consumption;
- Make sure you obtain results which are close to Fig. 10.

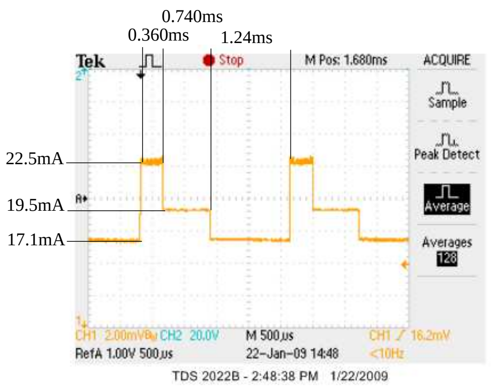
5.3.3 Impact of Transmission Power

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

- keep the following line:
  `#include "radios/family1/mrfi_spi.h"`



(a) Transmitter



(b) Receiver

Fig. 10. Energy consumption with continuous Tx/Rx.

register	**power**	**current drawn**
0x84	-24dBm	
0x55	-16dBm	
0x97	-10dBm	
0xA9	-4dBm	
0xFE	0dBm	
0xFF	1dBm	

Table 7
Impact of transmission power on current.

- add after line 7 the following line. This program the `PATABLE` register of the CC2500, which is responsible for the transmission power of the radio. You can put values between `0x00` and `0xFF`; these values map to a transmission power, as shown in Fig. 7.
  `mrfiSpiWriteReg(PATABLE,0xFF);`
- leave lines 14, 19 and 23 commented. Use your oscilloscope to visualize the maximum energy consumed. Repeat this for different values of PATABLE. Fill in Table 7 (p.23).

5.4 Wireless Chat

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, building a wireless chat. Note that you need two board and two USB programmers.

5.4.1 Running the Code

- In IAR, right click on project `txrx_chat` in the workspace and chose `Set as Active`.
- Compile and download the code onto both boards (Ctrl+D).
- Connect each board to a computer using the USB programmer and open PuTTY. When hitting enter, the message should appear on the other side's screen.

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.

6 RF Measurements (hours 6-7)

Because seeing is believing, the goal of this section is for you to see what a RF environment looks like. You will first build a spectrum analyzer to visualize to RF noise; you will then investigate on the relationship between RSSI and distance between sender and receiver, and on the relationship between RSSI and link probability.

6.1 *The importance of the CRC*

Each packet is appended with a 16-bit Cyclic-Redundancy-Check which is able to detect accidental alteration 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.

6.1.1 *Running the Code*

- In IAR, right click on project `txrx_crc` in the workspace and chose `Set as Active`.
- 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 multiple times this 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.
- You may wish to speed things up by commentling line 37 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.

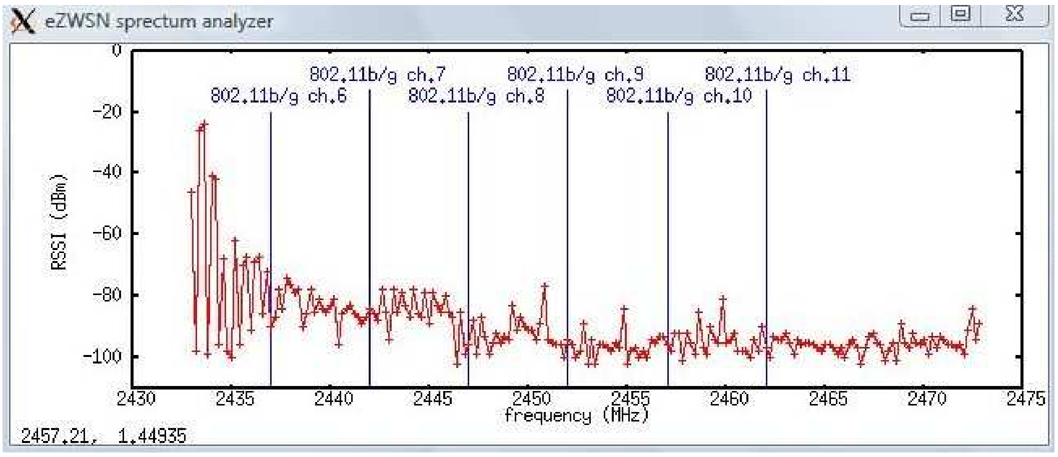


Fig. 11. Output of the spectrum analyzer in Section 6.2; one transmission is going on on channel 0.

6.2 Creating a Spectrum Analyzer to Measure Noise

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.

6.2.1 Running the Code

- Program a single board using the project `txrx_noise` in IAR; close IAR.
- Connect that board to the host computer, and find the COMx port it is connected to, as instructed in Section 2.3. Use PuTTY to read from that COMx port; you should see a continuous series of 200 numbers appearing.
- Start an X server onto the host computer using Xlaunch.
- Start a Cygwin Bash Shell and go into the folder containing `./txrx_noise.py`.
- `export DISPLAY=127.0.0.1:0.0`
- `cat /dev/ttyS`y where $y = x - 1$. You should see the same content appearing as when using PuTTY. If not, read the COMx port with PuTTY and try again.
- `cat /dev/ttyS`y`| ./txrx_noise.py`
  A window appears which looks like the one depicted in Fig. 11.

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.

6.2.2 Refresh Rate of the Spectrum Analyzer

- in the code, add after line 25
  `P2DIR |= 0x08;`
- in the code, add after line 28
  `P2DIR ^= 0x08;`
- using the oscilloscope, measure on extension port P6 the refresh rate of the frequency analyzer.
- What is the refresh rate ?
- move the last added line just after line 29. You now measure the time it takes for the mote to sample the noise level on one channel, and move to the other.
- What value do you measure ?

6.2.3 Testing the Spectrum Analyzer

- Reprogram a second board using the code described in Section 5.3, so that it continuously sends data. Start the continuous sending and visualize this transmission on the spectrum analyzer (at channel 0 by default).
- Using Section 5.2.2, change the operating channel, and see the implications on the spectrum analyzer.
- You can also see the impact of a running microwave oven.

6.3 RSSI vs. Distance

In this Section, we want to draw experimentally 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 straightforward to predict. It should be clear that repeating these measurements under different conditions yields very different results.

You will modify the code for the mote taken from Section 6.2 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:

- In the code used in the previous section, comment out line 29. 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 Section 6.2). We are interested in calculating the average value of each 200 point line.
- run the python script
  ```
  cat /dev/ttySy| ./txrx_rssi_dist.py
  ```
  where $y = x - 1$. This script outputs the average value of the RSSI 200 readings.
- Reprogram a second board using the code described in Section 5.3, so that it continuously sends data, on channel 0 and with a transmission power of 0dBm (see Section 5.3.3).
- Start the continuous sending and see how the RSSI decreases as the transmitter is moved away from the receiver.

Fig 12 plot 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.

6.4 Channel Success Probability

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 it's 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.

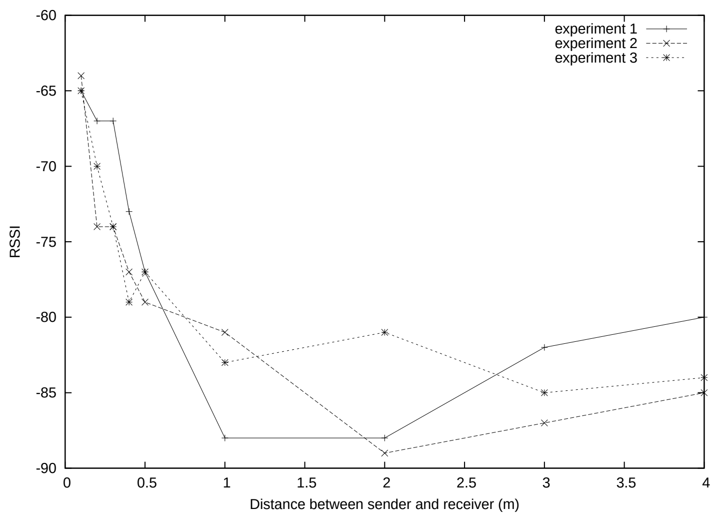


Fig. 12. Evolution of RSSI with distance.

The experimental setup goes as follows. We will use to boards, a sender and a receiver. The sender will 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 project `txrx_probability`.
- 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 not read the statistics; close PuTTY when done;
- in cygwin, enter the following command
  `cat /dev/ttyS`y`> probability.txt`
  where $y = x - 1$. This logs the output in a file; move the transmitter away from the receiver to record data for low RSSI values;
- in cygwin, enter the following commands
  ```
  gnuplot
  plot "probability.txt" using 1:2
  ```

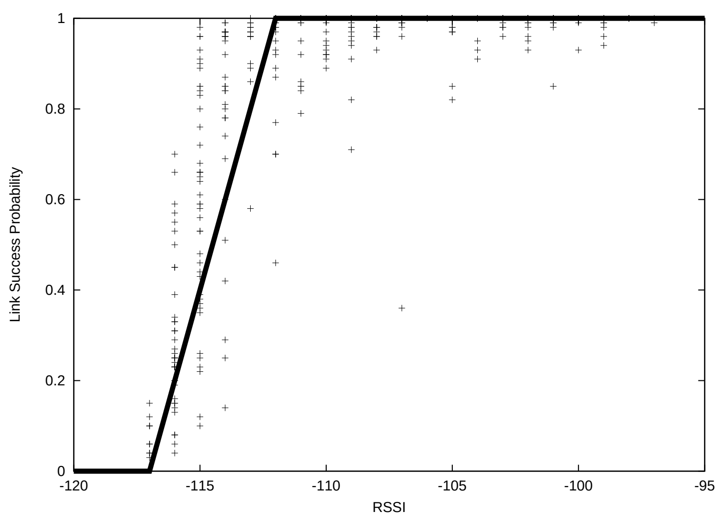


Fig. 13. Probability of Success as a function of RSS.

This plots the probability as a function of the RSSI, make sure to obtain a graph similar to the one in Fig. 13.

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. Note that line 59 is commented out, so this function is continuously repeated.
- **Line 61.** The format of a packet is depicted in Table 6. `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 Fig. 13, 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 Fig. 13.

7 Implementing A Preamble Sampling MAC Protocol (hour 8)

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 implement a variant of preamble sampling called MFP, 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.

7.1 An Introduction to Preamble Sampling

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.

Fig.14 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 D_{cca} seconds every CI seconds.

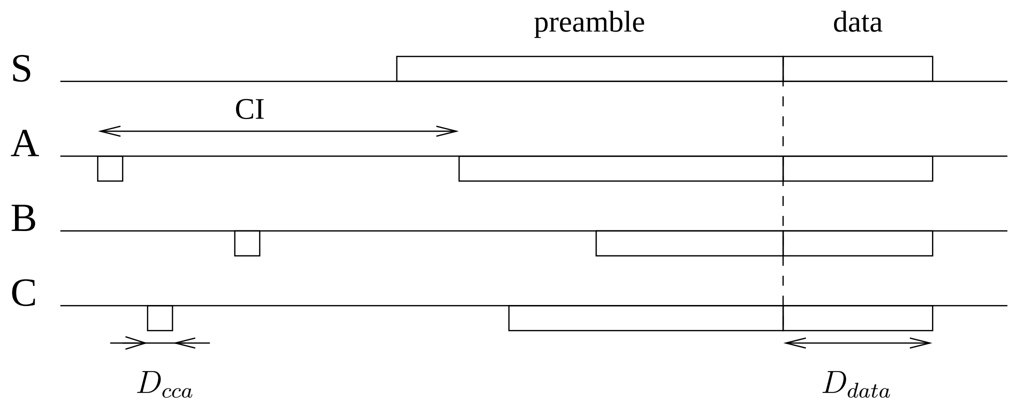


Fig. 14. Basic preamble-sampling.

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 T_{mf} seconds, and lasts for D_{mf}.

7.2 Running the code

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 a 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 boards with project `txrx_preample_msp`; 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 `01` on your screen.

Some keys for understanding the code:

- The microcontroller handles two timeouts, one for measuring CI, the other for D_{cca}. Those timeouts are sourced by two different clocks: a fast and accurate clock for D_{cca}; 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 (D_{cca}); and stops it when that expires. The radio is on only during D_{cca}.
- **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.

7.3 Timing issues: length of the preamble

- Use the oscilloscope to visualize the energy consumed by a board, you can clearly see the periodic wakeup of the CC2500 (see Fig. 15, lower part).
- What is the time between two successive wake-ups?

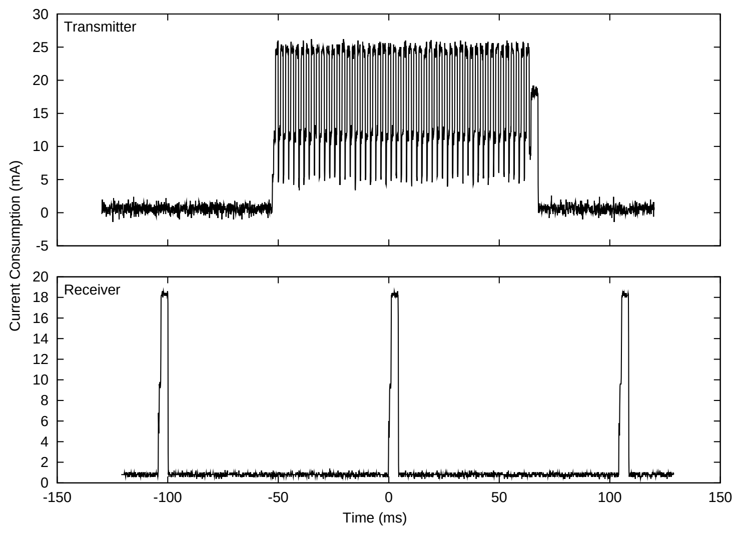


Fig. 15. Energy consumed by the transmitter and the receiver in preamble sampling. To function, the length of the preamble needs to be larger than the check interval CI.

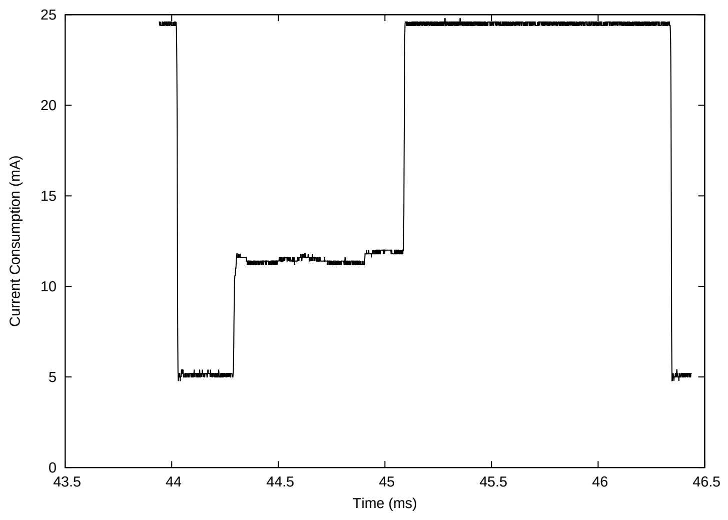


Fig. 16. A singled-out micro-frame.

- Push the button and capture the energy consumed when the board is in transmit mode (see Fig. 15, upper part). You can see the series on 50 microframes sent.
- By zooming in, what is the duration of one micro-frame? What is the duration of the preamble? Is that long enough? Why?

7.4 Timing issues: first micro-frame heard

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:

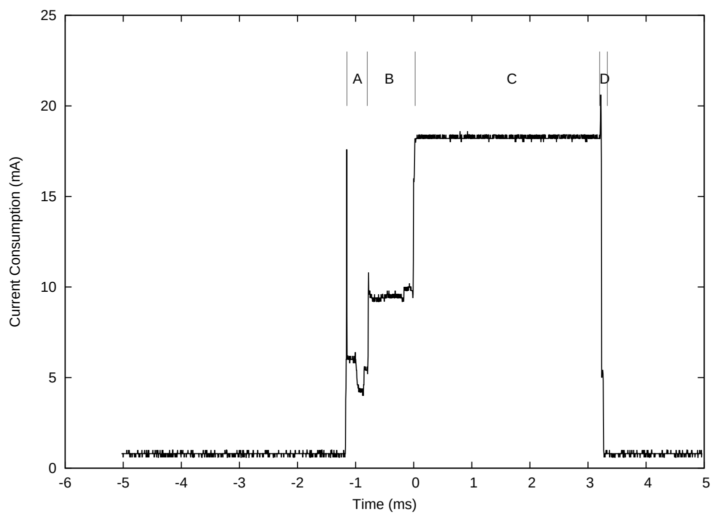


Fig. 17. Energy consumption measured when sampling the channel.

- move line 51 up two line, so that the counter value gets printed for every micro-frame 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`
  The first number you read is the first micro-frame received (here `30`), which is not necessarily the first one sent (here `50`).
- the sequence may not be continuous as in the example given above. Why?

7.5 Measuring the Energy Consumption

- 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 Fig. 17.
- You easily see the different phases when the CC2500 is turned on, the current drawn during these phases and their length. Fill Table 8.
- In Table 8, calculate the average current consumption of the board when it is listening, and the expected lifetime.

Phase		duration	av. current
idle			
A	μcontroller startup		
B	radio frequency calibration		
C	reception mode		
D	entering sleep mode		
average current consumption			
Lifetime with 1000mAh			

Table 8
Duration and average current consumption of the different phases observed in Fig. 17. Measurements averaged over 128 samples.

8 A full WSN example

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 Fig. 18 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.

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.

The source code is contained in project `txrx_wsn`. 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.

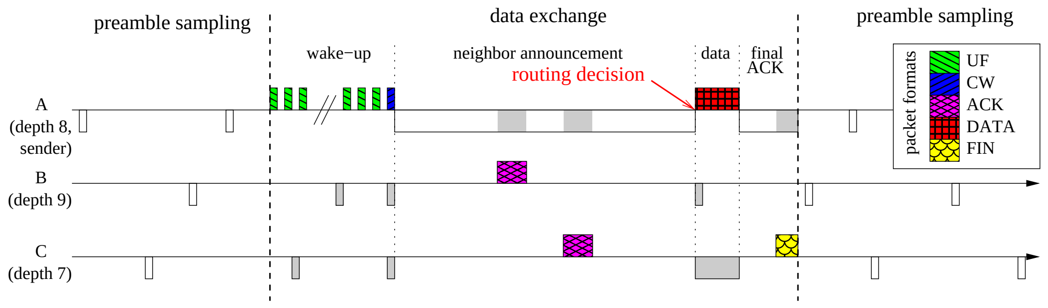


Fig. 18. Timeline illustrating the execution of the protocol stack. The x-axis represents time; a box above the line indicates that the radio is transmitting; a gray/white box under the axes means that the radio in receiving/idle listening, resp.; no box means the radio is turned off.

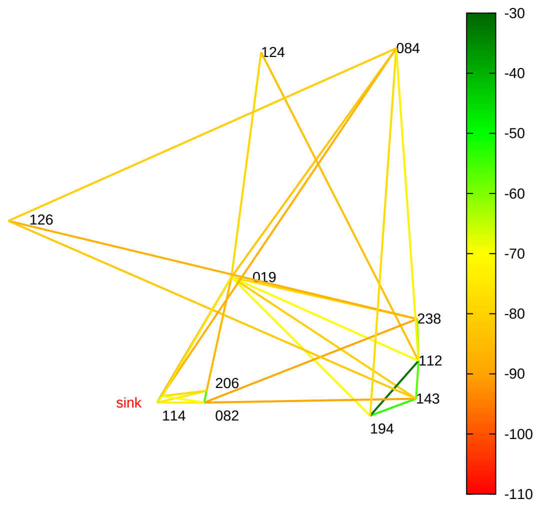


Fig. 19. An example outputted graph with 12 nodes. Links interconnect neighbors; colors indicate link quality in dBm.

8.0.1 *Running the Code*

Unfortunately, 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 [11] .

- 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/ttyS`x` | ./txrx_wsn.py` on the computer hosting the sink node. You should see a graph similar to Fig. 19.
- You may move nodes away from each other to obtain a multi-hop graph.

[11] 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.

References

[1] *MSP430x2xx Family User's Guide*, 2008, SLAU144E [available online].

[2] *MSP430x22x2, MSP430x22x4 Mixed Signal Microcontroller*, July 2006, SLAS504B [available online].

[3] *CC2500, Low-Cost Low-Power 2.4 GHz RF Transceiver*, 2007, SWRS040B [available online].

[4] *eZ430-RF2500 Development Tool User's Guide*, Texas Instruments, June 2008, SLAU227C [available online].