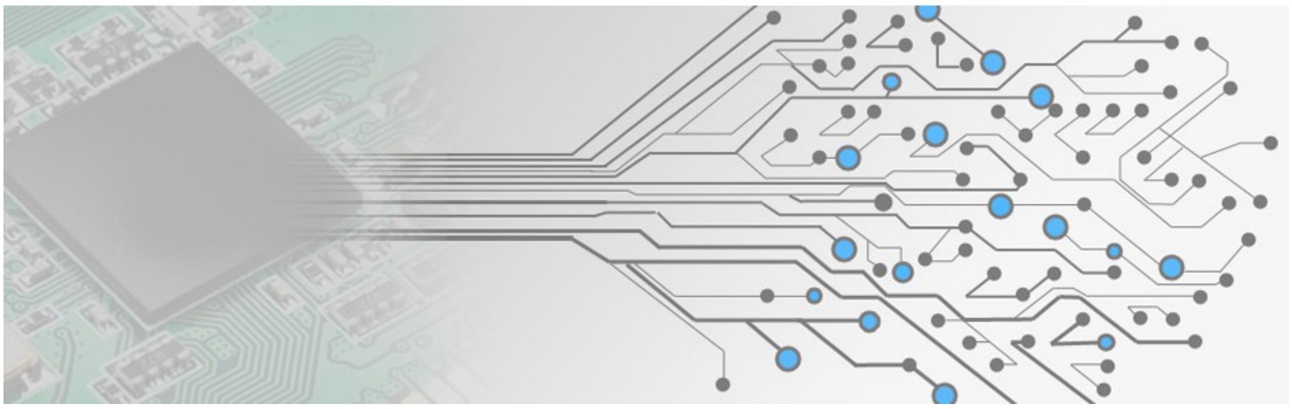


University of Thessaly

Department of Electrical and Computer Engineering

Thesis:

Development of a Wireless Sensor mote based on MSP430 micro-controller

Submitted by
Toutziaris Angelos

Supervisor
Dr. Athanasios Korakis

Advisor
Ioannis Kazdaridis

Volos, Greece October 2014

Acknowledgements

I would like to express my deep gratitude to Dr. Athanasios Korakis, my supervisor. I am particularly grateful for the guidance and assistance given by Ioannis Kazdaridis, whose contribution was crucial for the completion of this project.

Finally, I wish to thank my parents and friends for their support and encouragement throughout my study.

Περίληψη

Στόχος αυτής της διπλωματικής εργασίας είναι να μελετηθεί και να υλοποιηθεί η κατασκευή μίας ασύρματης πλατφόρμας αισθητήρων. Για να το πετύχουμε αυτό, μελετήσαμε τις βασικές αρχές της ασύρματης πλατφόρμας αισθητήρων και τις ιδιότητες και λειτουργικότητες τεσσάρων συσκευών. Του MSP430F5529 launchpad, του φωτοαντιστάτη, του αισθητήρα θερμοκρασίας και υγρασίας SHT11 και της συσκευής ασύρματης επικοινωνίας XBee ZB ZigBee.
Αρχικά, συνδέσαμε το launchpad με ένα windows 8.1 PC. Έπειτα εγκαταστήσαμε το απαραίτητο λογισμικό, συμπεριλαμβανομένων κάποιων drivers συσκευών και της εφαρμογής περιβάλλοντος προγραμματισμού Code Composer Studio IDE(CCS). Στη συνέχεια συνδέθηκαν οι αισθητήρες με την πλατφόρμα. Προγραμματίσαμε τη μονάδα επεξεργασίας μας, τον MSP430F5529 μικρο-ελεγκτή, να επικοινωνεί με τους αισθητήρες να δέχεται και να υπολογίζει τιμές θερμοκρασίας, υγρασίας και φωτεινότητας.
Έπειτα προσθέσαμε στην υλοποίηση κάποιων χαρακτηριστικών ασφάλειας και βελτιστοποίησης, όπως το watchdog και τη λειτουργία χαμηλής κατανάλωσης. Προγραμματίσαμε τον μικρο-ελεγκτή να επανεκκινεί σε περίπτωση που συμβεί κάποιο σφάλμα λογισμικού, και να εισέρχεται σε κατάσταση χαμηλής ενέργειας όταν είναι αδρανής.
Το τελικό βήμα ήταν να ρυθμίσουμε δύο Xbee συσκευές ασύρματης επικοινωνίας. Ρυθμίσαμε τις συσκευές αυτές να συνδέονται μεταξύ τους. Μία συσκευή συνδέθηκε στον υπολογιστή και μία στην πλατφόρμα.
Δοκιμάσαμε δύο διαφορετικές υλοποιήσεις. Στην πρώτη, προγραμματίσαμε τον μικρο-ελεγκτή να συλλέγει μετρήσεις, να ενεργοποιεί το Xbee, να μεταδίδει τις τιμές μετρήσεων στον υπολογιστή, έπειτα να απενεργοποιεί ξανά το Xbee και να εισέρχεται σε κατάσταση χαμηλής κατανάλωσης ενέργειας. Στην δεύτερη υλοποίηση, ρυθμίσαμε το Xbee έτσι ώστε να ενεργοποιείται περιοδικά, μετά από μία κατάσταση απενεργοποίησης. Όταν συμβεί αυτό, να ενεργοποιεί τον μικρο-ελεγκτή έτσι ώστε να του στείλει τα δεδομένα μετρήσεων για να τα μεταδώσει ασύρματα στο έτερο Xbee. Από την πλευρά του, ο μικρο-ελεγκτής λειτουργεί σε κατάσταση χαμηλής κατανάλωσης ενέργειας, ενεργοποιείται ανά περιόδους, συλλέγει μετρήσεις από τους αισθητήρες και επιστρέφει σε κατάσταση χαμηλής κατανάλωσης.

Abstract

The goal of this thesis is to study the development of a wireless sensor mote and develop a simple mote. To achieve this, we studied the basic principles of a wireless sensor mote, the attributes and functionality of four devices. The MSP430F5529 launchpad, the photoresistor, the SHT11 temperature and relative humidity sensor and the XBee ZB ZigBee Mesh Module.

Initially, we connected the launchpad with a PC running windows 8.1. After that, we installed the necessary software including device drivers, programming/debugging application Code Composer Studio IDE(CCS). Next, the sensor modules were connected with the launchpad. We programmed the MSP430F5529 micro-controller to communicate with the sensors and obtain data and calculate environmental information, such as temperature, relative humidity and light density levels.

Next, we added some safety and optimization features to the implementation, such as watchdog and low power mode. We programmed the micro-controller to reset when an error occurs and operate in low power mode when it is idle.

The final step was to set up two Xbee wireless modules. We configured the modules to connect to each other. One module was connected with the host PC and the other with the launchpad.

We tried two different implementations. In the first one, we programmed the micro-controller to take measurements, cause the Xbee to turn ON, transmit these measurements to the host PC, turn Xbee to sleep state again and then operate in low power mode. In the second one, we programmed the Xbee to turn ON periodically after operating in Sleep state for a specific duration of time. When this happens, it causes the micro-controller to send measured data via the wireless module to the host PC. The micro-controller operates in low power mode, wakes up after a period of time, takes measurements and return to low power mode.

Table of contents

Abbreviations........8
Introduction........9
Wireless sensor mote........9
Controller........9
Transceiver........10
External memory........10
Power source........10
Sensors........11
Project Description........11
Chapter 1. LaunchPad........12
Texas Instruments MSP430F5529 launchpad........12
USB interface........13
eZ-FET lite Onboard Emulator........14
Emulator and Target Isolation Jumper Block........15
3.3-V and 5-V Jumpers........16
Emulator Connection and Application UART........16
Programming the MSP430F5529 with external emulator..17
MSP430F5529 micro-controller........19
MSP430F5529 Key Features:........20
Oscillator and System Clock........20
Configuring Clocks........21
Central Processing Unit (CPU)........22
Chapter 2. Sensors........23
Light intensity sensor / Photoresistor........23
Circuitry........24
Source code........25
Integrated temperature sensor........26
Source code........26
Sensirion SHT11........28
Memory........29
Circuitry........30

Communication....................................31
Communication process....................................31
Functionality....................................32
Measurement of R. Humidity & Temperature....................................32
Connection reset sequence....................................32
Read-Write status register....................................32
Power Consumption....................................33
Source code....................................33
Chapter 3. Wireless Communication....................................37
Xbee Series 2 Radio module....................................37
Configuration....................................38
1st implementation: End Device with pin wake-up. 41
2nd implementation: End Device with cyclic sleep..42
Port/Pin Interrupt....................................44
Source Code....................................44
UART Interface....................................45
Chapter 4. Implementation Features....................................47
Watchdog....................................47
Watchdog Mode....................................48
Watchdog Timer Interrupts....................................48
Clock Fail-Safe Feature....................................48
Watchdog time intervals and clock sources....................................49
Watchdog Control Registers and Source Code....................................49
Low Power Modes....................................50
Operating Modes....................................50
Status Register....................................51
Exiting Low-Power Modes LPM0 Through LPM4.....52
Timers....................................54
Timer_A registers....................................54
Configuration....................................57
Source code....................................57
Conclusion - Summary....................................58
Final Results....................................58

Future Work....58
Appendix....59
Source Code....59
Definitions and Functions....59
1st Implementation: Xbee pin wake-up....64
2nd Implementation: Xbee cyclic sleep....67
References....70

Abbreviations

WSN:	wireless sensor network
MCU:	micro-controller unit
UART:	universal asynchronous receiver/transmitter
SBW:	Spy-Bi-Wire
UCS:	unified clock system
CRC:	cyclic redundancy check
LPM:	low power mode
USC:	universal serial communication
eUSCI:	enhanced universal serial communication interface
WDT:	watchdog timer
PUC:	power-up clear
SMCLK:	sub main clock
ACLK:	auxiliary clock
VLOCLK:	very low frequency oscillator
DCO:	digitally controlled oscillator
DLL:	dynamic link library
SR:	stare register

Introduction

Wireless sensor mote

A **sensor node**, also known as a mote, is a node in a wireless sensor network that is capable of performing some processing, gathering sensory information and communicating with other connected nodes in the network. Wireless sensor nodes have existed for decades and used for applications as diverse as earthquake measurements to warfare. Nowadays, motes focus on providing the **longest wireless range** (dozens of km), the **lowest energy consumption** (some uA) and the **easiest development process** for the user.

The main components of a sensor node are a **controller**, **transceiver**, **external memory**, **power source** and one or more **sensors**.

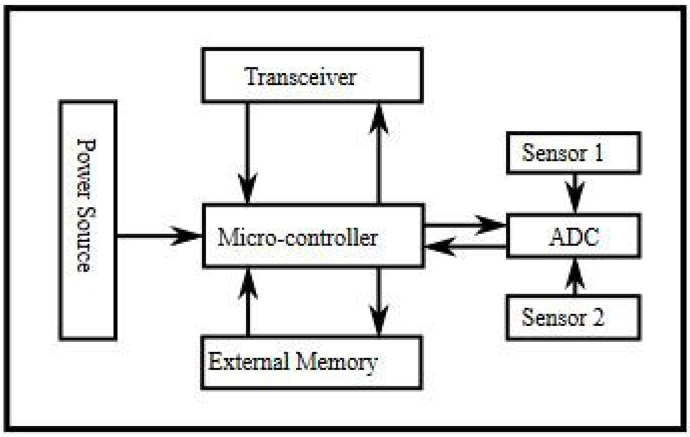


Fig 1: Wireless Sensor Node

Controller

The controller **performs tasks**, **processes data** and **controls** the **functionality** of other components in the sensor node. The most common controller is a **micro-controller**. Other alternatives that can be used as a controller are: a general purpose desktop microprocessor, digital signal processors, FPGAs and ASICs. A micro-controller is often used in many embedded systems such as sensor nodes because of its low cost, flexibility to connect to other devices, ease of programming, and low power consumption.

Transceiver

The possible choices of wireless transmission media are **radio frequency** (RF), **optical communication** (laser) and **infrared**. Radio frequency-based communication is the most relevant that fits most of the WSN applications. The functionality of both transmitter and receiver are combined into a single device known as a transceiver. The transceiver's operational states are **transmit**, **receive**, **idle**, and **sleep**.

Most transceivers operating in idle mode have a **power consumption** almost equal to the power consumed in receive mode. Thus, it is better to completely shut down the transceiver rather than leave it in the idle mode when it is not transmitting or receiving. A significant amount of power is consumed when switching from sleep mode to transmit mode in order to transmit a packet.

External memory

From an energy perspective, the most relevant kinds of memory are the **on-chip memory** of a micro-controller and **Flash memory**; off-chip RAM is rarely, if ever, used. Flash memories are used due to their cost and **storage capacity**. Memory requirements are very much application dependent. Two categories of memory based on the purpose of storage are: **user memory** used for storing application related or personal data, and **program memory** used for programming the device. Program memory also contains identification data of the device if present.

Power source

An important aspect in the development of a wireless sensor node is ensuring that there is always adequate energy available to power the system. The sensor node consumes power for sensing, communicating and data processing. Power is stored either in **batteries** or **capacitors**. Batteries, both rechargeable and non-rechargeable, are the main source of power supply for sensor nodes. As wireless sensor nodes are typically very small electronic devices, they can only be equipped with a limited power source of less than 0.5-2 ampere-hour and 1.2-3.7 volts. Current sensors are able to renew their energy from solar sources, temperature differences, or vibration.

Sensors

Sensors are hardware devices that produce a measurable response to a change in a physical condition like temperature or pressure. Sensors measure physical data of the parameter to be monitored. A sensor node should be **small** in size, consume extremely **low energy**, operate in **high volumetric densities**, be **autonomous** and operate **unattended**, and be **adaptive** to the environment.
Sensors are classified into three categories: passive, omni-directional sensors; passive, narrow-beam sensors; and active sensors. **Passive sensors** sense the data without actually manipulating the environment by active probing. They are self powered; that is, energy is needed only to amplify their analog signal. **Active sensors** actively probe the environment, for example, a sonar or radar sensor, and they require continuous energy from a power source. **Narrow-beam sensors** have a well-defined notion of direction of measurement, similar to a camera. **Omni-directional sensors** have no notion of direction involved in their measurements.

Project Description

The **purpose** of our project is the development of a wireless sensor node based on the **MSP430 micro-controller**. This is a widely applicable micro-controller in the area of wireless sensor prototypes since it features advanced characteristics described in the next sections.
The main principle is that the micro-controller is interfaced with several sensing modules that provide realistic measurements of physical magnitudes. Moreover, the developed node is able to communicate with a gateway node in order to transmit the acquired measurements.

The **sensors** used to measure temperature, relative humidity and brightness are the following:

- Light intensity sensor or **Photoresistor**: Detects the light level in the room in which it is located.
- Sensirion **SHT11** temperature and humidity sensor.
- The **internal** temperature sensor which is integrated in the platform.

The **wireless connection** and communication is implemented via a **Xbee Series2** radio transceiver.
Measuring values were stored in micro-controller's **on-chip memory**, and the mote was **powered** by a **USB cable**.

The mote was programmed/debugged using the integrated development environment(IDE) **Code Composer Studio**(CCS) v5.5.0

Below, there are some details given about the features and characteristics of the devices.

Chapter 1. LaunchPad

Texas Instruments MSP430F5529 launchpad

The MSP430F5529LP LaunchPad is a simple evaluation module for the MSP430F5529 micro-controller. It provides all the components and interfaces necessary for developing on the MSP430 MCU. Contains an on-board emulation for programming and debugging, as well as buttons and LEDs for simple user interface. The micro-controller will be the mote's controller. It will manage the procedure of measuring sensing values, extracting environmental information and sending this information to a host node, a Windows 8.1 machine.

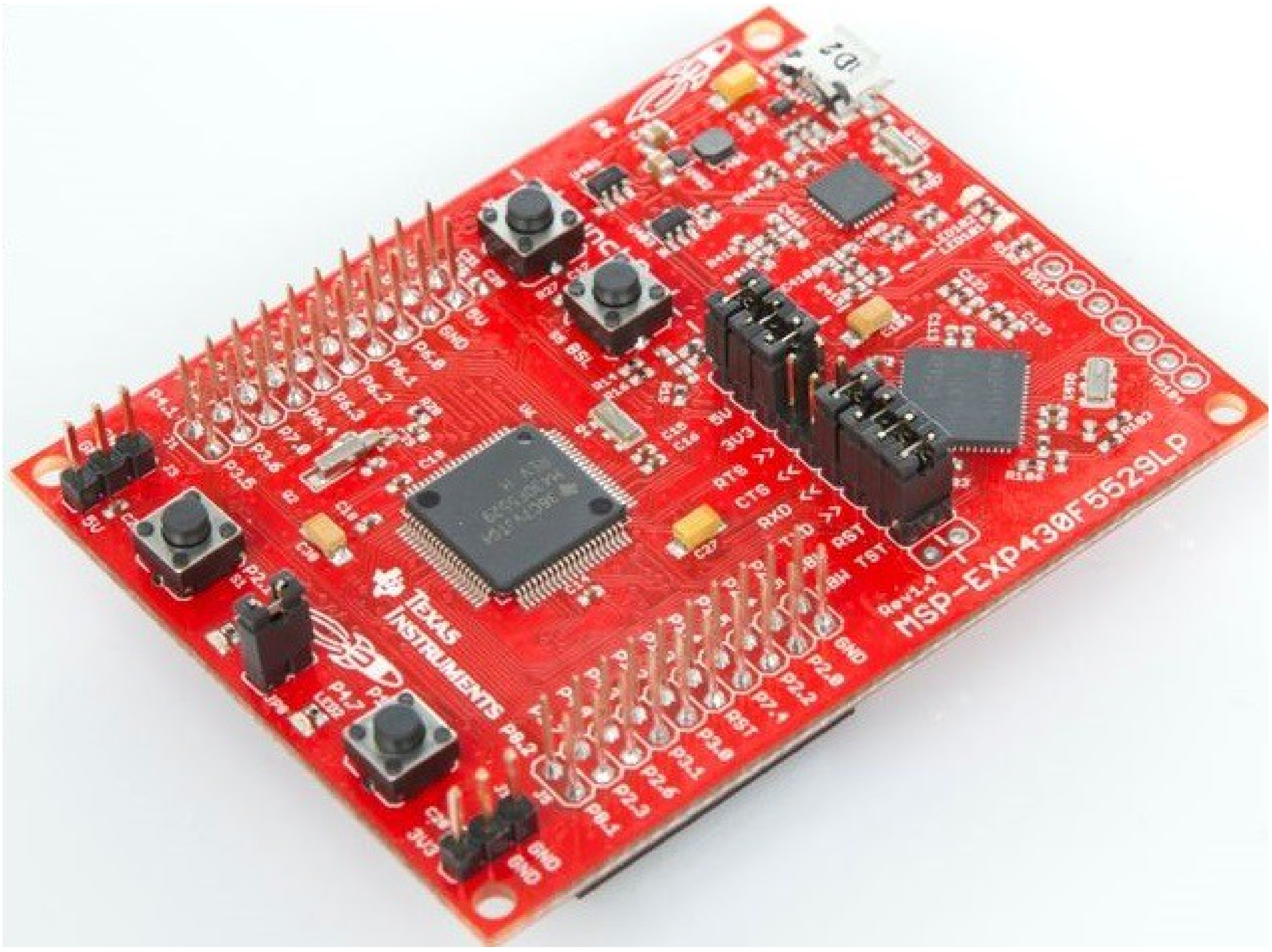


Fig 2: Texas Instruments MSP430F5529 Launchpad

MSP430F5529 launchpad consists of three main parts:

- USB hub and power supply
- eZ-FET Emulator
- MSP430F5529 micro-controller

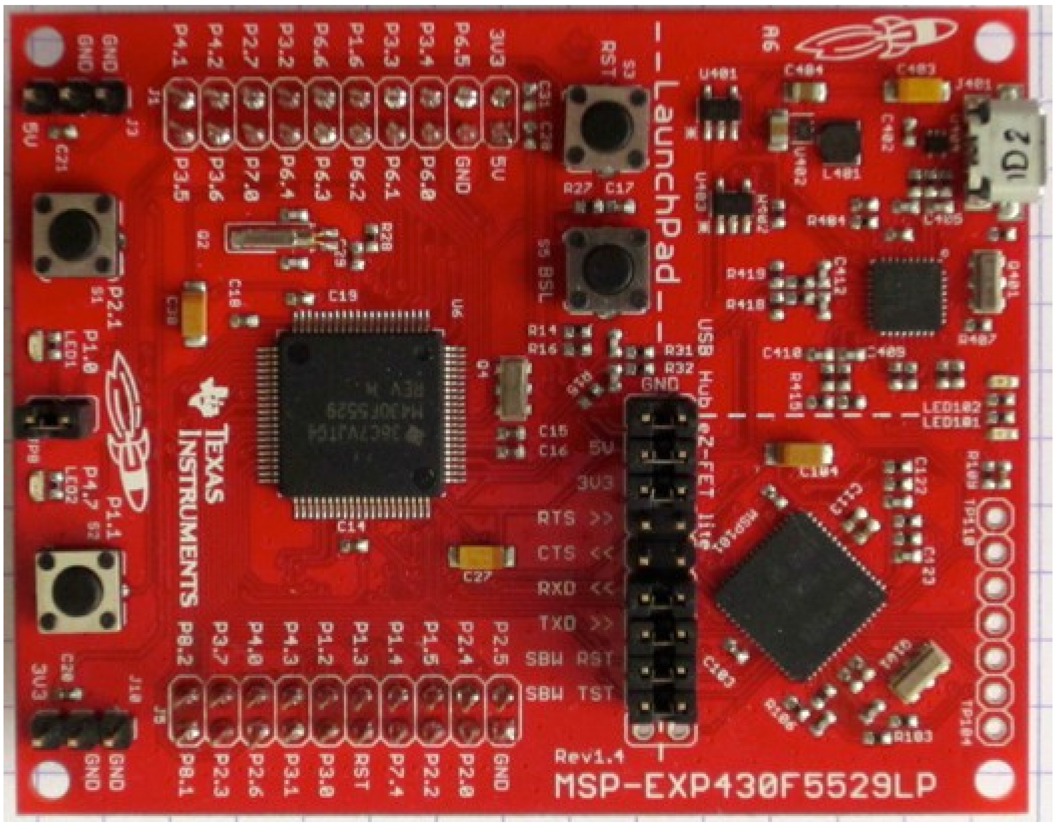


Fig 3: Texas Instruments MSP430F5529 Launchpad

USB interface

USB hub provides the power supply for all devices on the launchpad. The USB 5-V bus power is reduced to 3.3 V, by a dc-dc converter. It also gives the ability to interface with the USB-enabled micro-controller using MSC, HID or CDC, and communicate with the PC via the implemented back-channel UART.

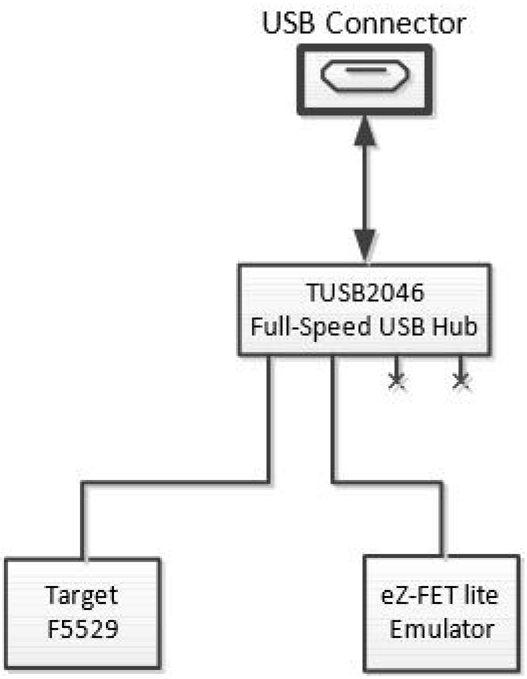


Fig 4: USB hub Connection

eZ-FET lite Onboard Emulator

eZ-FET Emulator is an onboard emulator/programmer that allows **programming/debugging** the MSP430F5529 micro-controller without connecting an external programmer.
The eZ-FET lite is a simple and **Open-source** emulator that supports almost all MSP430 MCUs. It also provides a "backchannel" UART-over-USB connection with the host, which can be very useful during debugging, and has LEDs for visual feedback.

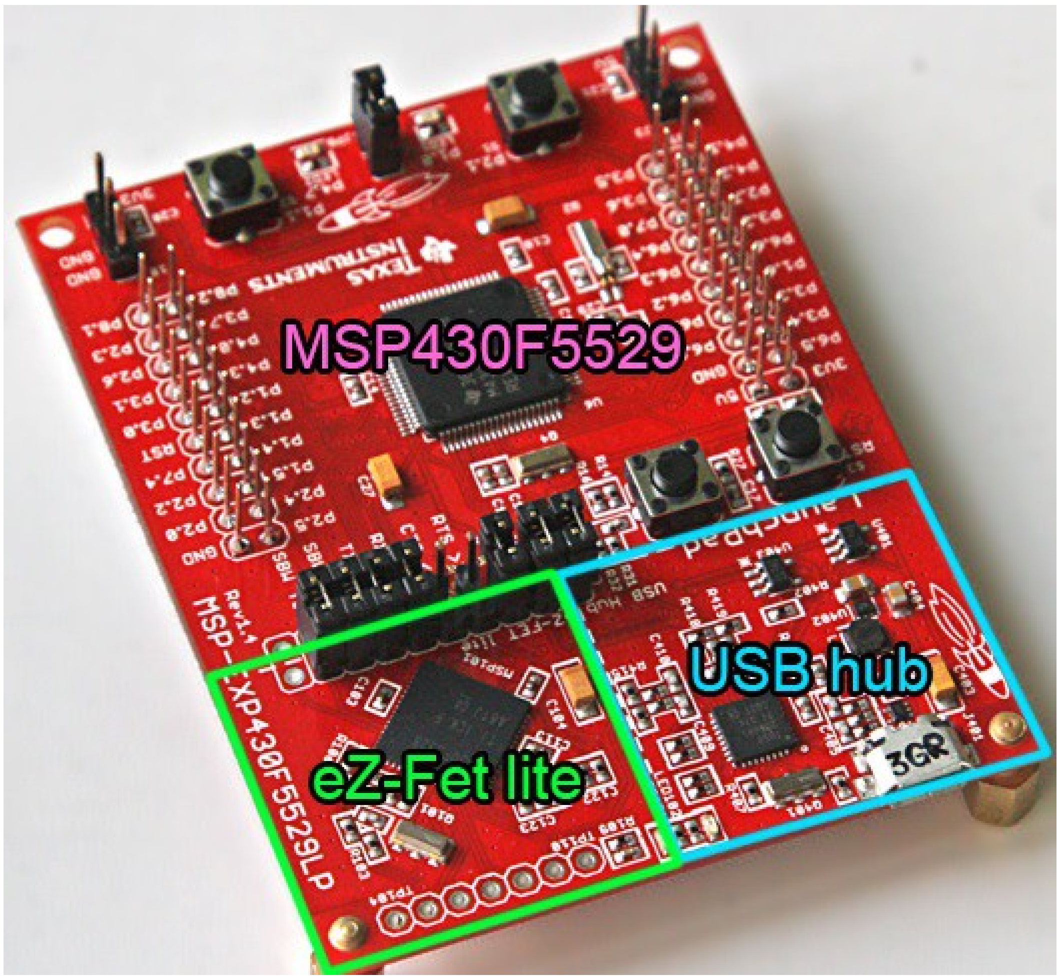


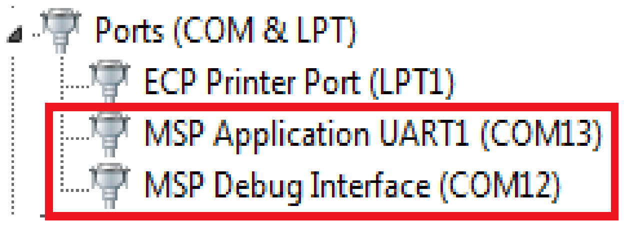
Fig 5: MSP430F5529 launchpad components

The emulator is supported by the following software development tools: TI's Eclipse based Code Composer Studio IDE (CCS), IAR Embedded Workbench IDE (IAR), and Energia open source code editor.
In the context of this project we used CCS IDE to program the micro-controller.

The eZ-FET lite emulator itself is a composite USB device, which means that it contains two USB interfaces:

- A CDC interface (virtual COM port) for the emulation function
- A CDC interface (virtual COM port) for the application UART

A computer communicates with the emulator through these two serial interfaces.



Ports (COM & LPT)
ECP Printer Port (LPT1)
MSP Application UART1 (COM13)
MSP Debug Interface (COM12)

Fig 6: Emulator Interfaces

Emulator and Target Isolation Jumper Block:

The emulator is connected to the micro-controller with nine wires.
A set of ten jumpers which is placed between the emulator and the F5529 target device allows the developer to disconnect these wires.

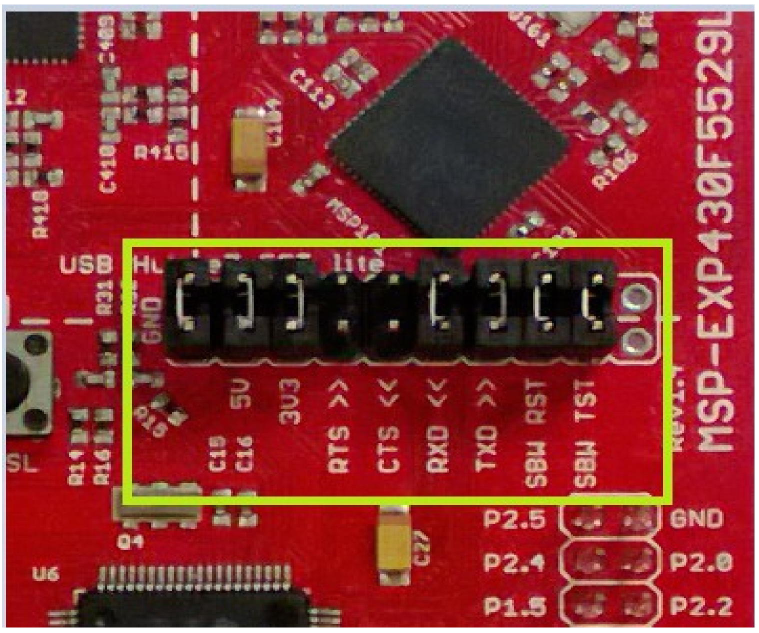
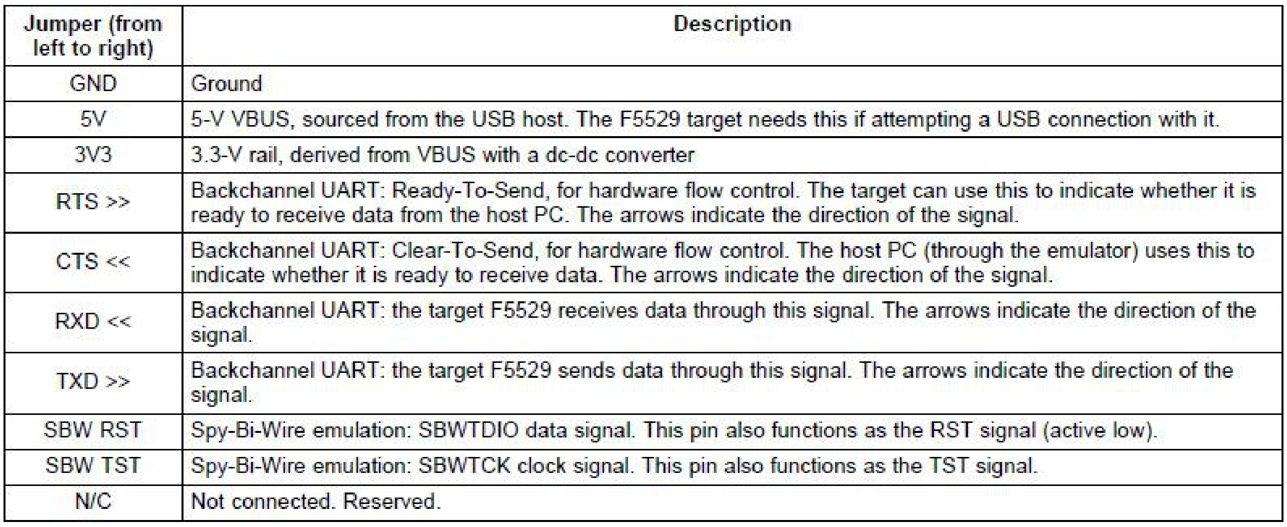


Fig 7: Emulator and Target Isolation Jumper Block

The connection wires as found on jumper block are shown below:



Jumper (from left to right)	Description
GND	Ground
5V	5-V VBUS, sourced from the USB host. The F5529 target needs this if attempting a USB connection with it.
3V3	3.3-V rail, derived from VBUS with a dc-dc converter
RTS >>	Backchannel UART: Ready-To-Send, for hardware flow control. The target can use this to indicate whether it is ready to receive data from the host PC. The arrows indicate the direction of the signal.
CTS <<	Backchannel UART: Clear-To-Send, for hardware flow control. The host PC (through the emulator) uses this to indicate whether it is ready to receive data. The arrows indicate the direction of the signal.
RXD <<	Backchannel UART: the target F5529 receives data through this signal. The arrows indicate the direction of the signal.
TXD >>	Backchannel UART: the target F5529 sends data through this signal. The arrows indicate the direction of the signal.
SBW RST	Spy-Bi-Wire emulation: SBWTDIO data signal. This pin also functions as the RST signal (active low).
SBW TST	Spy-Bi-Wire emulation: SBWTCK clock signal. This pin also functions as the TST signal.
N/C	Not connected. Reserved.

Fig 8: Emulator Jumper Description

3.3-V and 5-V Jumpers

The emulator provides two supply voltages to the micro-controller. The 5-V VBUS and 3.3-V power rails, travel through the isolation jumper block.

This routing serves various functions:

- Measurement of the target's **power consumption**
- Removing the emulator from the circuit when an external (non-USB) power source is used
- Removing the F5529 target from the circuit when a different external target board is attached to the emulator(If you want to use the on-board eZ-FET lite emulator with a **different target**, you can remove the jumpers and connect your target hardware to the jumper block)

Emulator Connection and Application UART

MSP430F5529 supports both standard **four-wire JTAG** and the **two-wire Spy-Bi-Wire** (SBW) standard.
The eZ-FET lite emulator on the F5529 LaunchPad supports SBW only. These two signals travel through jumpers in the isolation block, and can be disconnected if desired. They are labeled on the block as "SBW RST" and "SBW TEST".
These two wires are responsible for programming/debugging the target MCU.

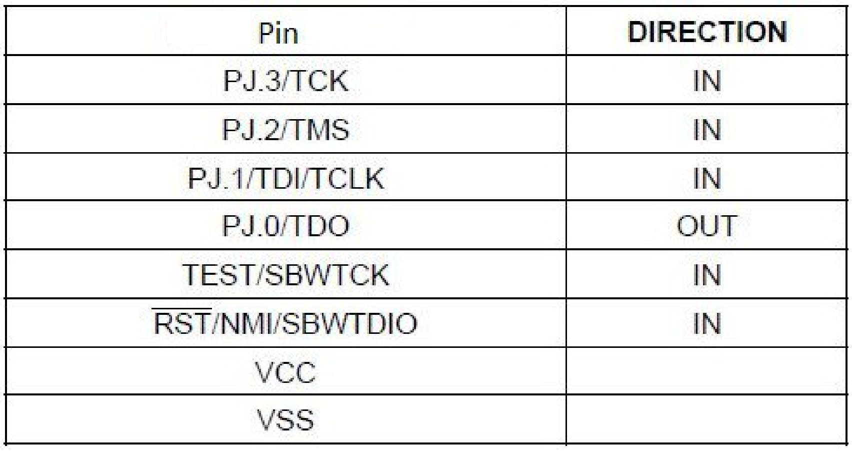
The backchannel UART consists of four signals: the data signals TXD and RXD, and the hardware flow control signals RTS and CTS. All four of these signals travel through the jumper block as well and can be disconnected.

The **backchannel UART** pins can be configured for other functionality instead of the backchannel UART, for example UART communication with an external device such as a sensor or a wireless module. If this is desired, these jumpers must be removed.

Programming the MSP430F5529 with external emulator

In order to develop a mote including a single MSP430F5529 micro-controller that is not connected to an emulator embedded within a launchpad, an external emulator must be used. The micro-controller's firmware must be programmed with this emulator.
This external emulator must be connected according to one of the following ways.

If the **four-wire JTAG** standard is used, the pins that need to be connected are shown below:



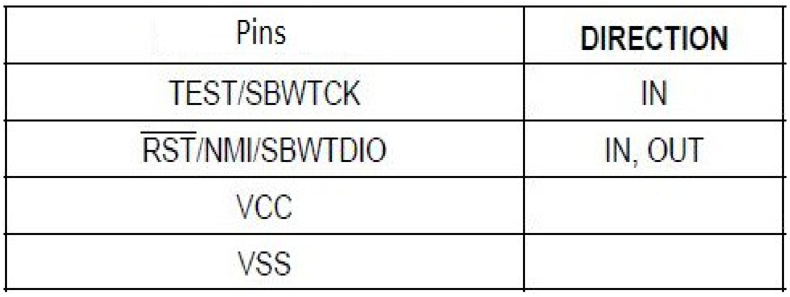
Pin	DIRECTION
PJ.3/TCK	IN
PJ.2/TMS	IN
PJ.1/TDI/TCLK	IN
PJ.0/TDO	OUT
TEST/SBWTCK	IN
$\overline{\text{RST}}$/NMI/SBWTDIO	IN
VCC	
VSS	

Fig 9: JTAG-4 pins

The emulator's JTAG clock pin must be connected with the micro-controller's pin 75(Pj.3/TCK).
JTAG state control pin must be connected with pin 74(Pj.2/TMS).
Emulator's JTAG data output/test clock pin must connect to pin 73(Pj.1/TDI/TCLK).
Emulator's data input pin must connect to pin 72(Pj.0/TDO).
The output pin that enables JTAG pins must connect to pin 71(TEST/SBWTCK).
Target reset pin must connect to pin 76(RST/NMI/SBWTDIO).

Finally, power supply and ground must connect to pins 11, 18, 50(Power Supply) and 14, 19, 49 pins (Ground).

If the slower two-wire **Spy-Bi-Wire** (SBW) standard is used, the pins that need to be connected are:



Pins	DIRECTION
TEST/SBWTCK	IN
$\overline{\text{RST}}$/NMI/SBWTDIO	IN, OUT
VCC	
VSS	

Fig 10: JTAG-2 Spy-Bi-Wire pins

The emulator's clock output must connect to the micro-controller's pin 71(TEST/SBWTCK).
The data input/output pin must connect to pin 76(RST/NMI/SBWTDIO).

Power supply and ground must connect to micro-controller's pins 11, 18, 50(Power Supply) and 14, 19, 49 pins (Ground).

MSP430F5529 micro-controller

The MSP430F5529 16-bit MCU has 128KB flash, 8KB RAM, 25-MHz CPU speed, integrated USB, and many peripherals. The micro-controller will be set to manage the functions of the mote: sensing, calculating variables, transmitting data, implementing special features.

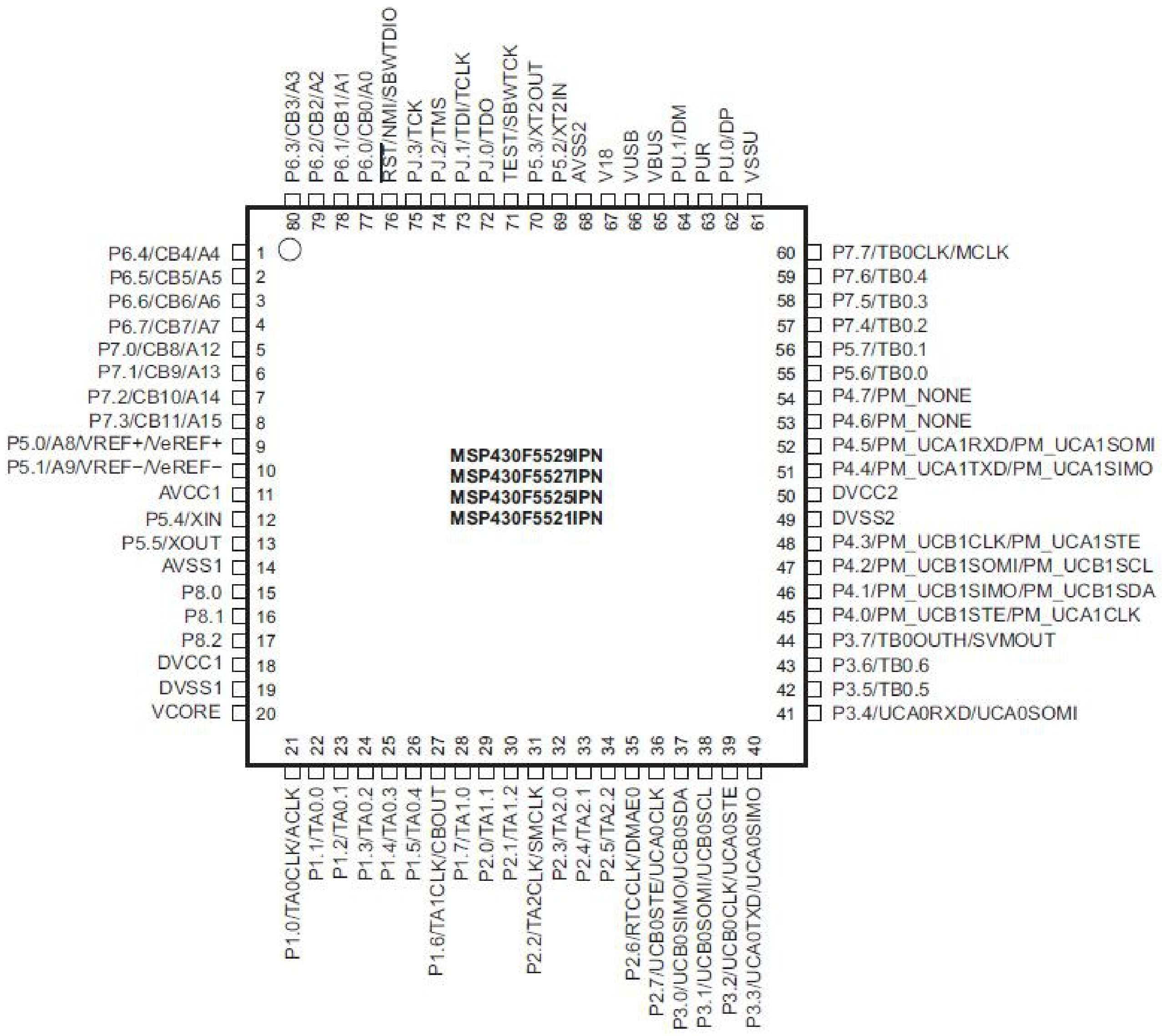


Fig 11: MSP430F5529 Pins

MSP430F5529 Key Features:

- USB-enabled MSP430F5529 16-bit MCU
  - Up to 25-MHz System Clock
  - 1.8-V to 3.6-V operation
  - 128KB flash, 8KB RAM
  - Five timers
  - Up to four serial interfaces (SPI, UART, I2C)
  - 12-bit analog-to-digital converter
  - Analog comparator
  - Integrated USB, with a complete set of USB tools, libraries, examples, and reference guides
- Two integrated buttons fully programmable Reset button, Bootstrap loader.
- Integrated temperature sensor.

Oscillator and System Clock

The clock system in the MSP430F5529 is supported by the Unified Clock System (UCS) module that includes support for:

- A 32-kHz watch crystal oscillator (XT1 LF mode) (XT1 HF mode is not supported)
- An internal very-low-frequency oscillator (VLO)
- An internal trimmed low-frequency oscillator (REFO)
- An integrated internal digitally controlled oscillator (DCO)
- A high-frequency crystal oscillator (XT2).

The UCS module is designed to meet the requirements of both low system cost and low power consumption. It features digital frequency locked loop (FLL) hardware, that in conjunction with a digital modulator, stabilizes the DCO frequency to a programmable multiple of the selected FLL reference frequency.
The internal DCO provides a fast turn-on clock source and stabilizes in 3.5 µs (typical).

The UCS module provides the following clock signals:

- Auxiliary clock (ACLK), sourced from a 32-kHz watch crystal (XT1), a high-frequency crystal (XT2), the internal low-frequency oscillator (VLO), the trimmed low-frequency oscillator (REFO), or the internal digitally controlled oscillator DCO.
- Main clock (MCLK), the system clock used by the CPU. MCLK can be sourced by same sources made available to ACLK.
- Sub-Main clock (SMCLK), the subsystem clock used by the peripheral modules. SMCLK can be sourced by same sources made available to ACLK.
- ACLK/n, the buffered output of ACLK, ACLK/2, ACLK/4, ACLK/8, ACLK/16, ACLK/32.

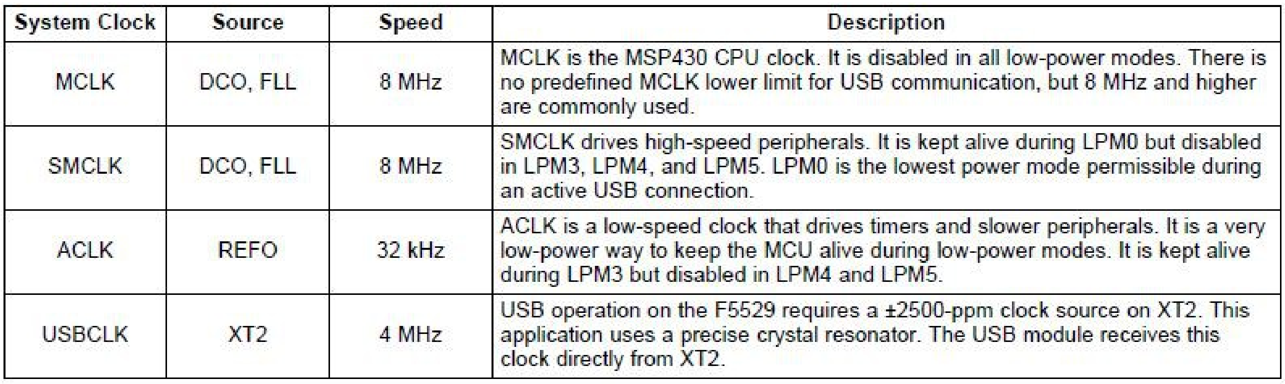
Configuring Clocks

MSP430 applications typically use a fast clock and a slow clock.
The fast clock (called **MCLK**) sources the CPU and peripherals in some cases, while the slow one keeps timers and peripherals operating during low power modes.
This approach reduces **power**: slow clocks consume less power, so the more often the fast clock can be disabled, the less power your application may consume.
Typically this fast clock is the MCU's integrated digitally controlled oscillator (DCO). The **DCO** itself is an important low-power tool, because unlike a crystal, it has a very fast start-up time, and thus can be quickly shut down and re-enabled. The DCO can be activated by an interrupt and stabilize fast enough to respond to it.

Many MSP430 devices, including the F5529, couple the DCO with a frequency-locked loop (FLL) module that keeps the DCO locked to a precise slower-frequency reference. This gives good control over the DCO frequency.

The F5529 has three slow clocks available:

- REFO: This is a modestly precise low-power on-chip oscillator that does not require a crystal. It operates at 32 kHz.
- LFXT1: This is a crystal oscillator. It is very precise and lower power than the REFO, but it requires a crystal. It, too, operates at 32 kHz.
- VLO: This oscillator is not very precise but does not require a crystal and has the lowest power of the three. It usually operates somewhere between 12 kHz and 20 kHz.



System Clock	Source	Speed	Description
MCLK	DCO, FLL	8 MHz	MCLK is the MSP430 CPU clock. It is disabled in all low-power modes. There is no predefined MCLK lower limit for USB communication, but 8 MHz and higher are commonly used.
SMCLK	DCO, FLL	8 MHz	SMCLK drives high-speed peripherals. It is kept alive during LPM0 but disabled in LPM3, LPM4, and LPM5. LPM0 is the lowest power mode permissible during an active USB connection.
ACLK	REFO	32 kHz	ACLK is a low-speed clock that drives timers and slower peripherals. It is a very low-power way to keep the MCU alive during low-power modes. It is kept alive during LPM3 but disabled in LPM4 and LPM5.
USBCLK	XT2	4 MHz	USB operation on the F5529 requires a ±2500-ppm clock source on XT2. This application uses a precise crystal resonator. The USB module receives this clock directly from XT2.

Fig 12: MSP430 Clocks

In the context of this project, we use the ACLK in order to achieve low power consumption. When the micro-controller is idle, we turn off high frequency clocks, forcing the device to operate in low power mode.

Central Processing Unit (CPU)

The MSP430 CPU has a 16-bit **RISC** architecture that is highly transparent to the application.
All **operations**, other than program-flow instructions, are performed as register operations in conjunction with seven addressing modes for source operand and four addressing modes for destination operand.
The CPU is integrated with 16 **registers** that provide reduced instruction execution time. The register-to-register operation execution time is one cycle of the CPU clock. Four of the registers, R0 to R3, are dedicated as program counter, stack pointer, status register, and constant generator, respectively. The remaining registers are general-purpose registers.

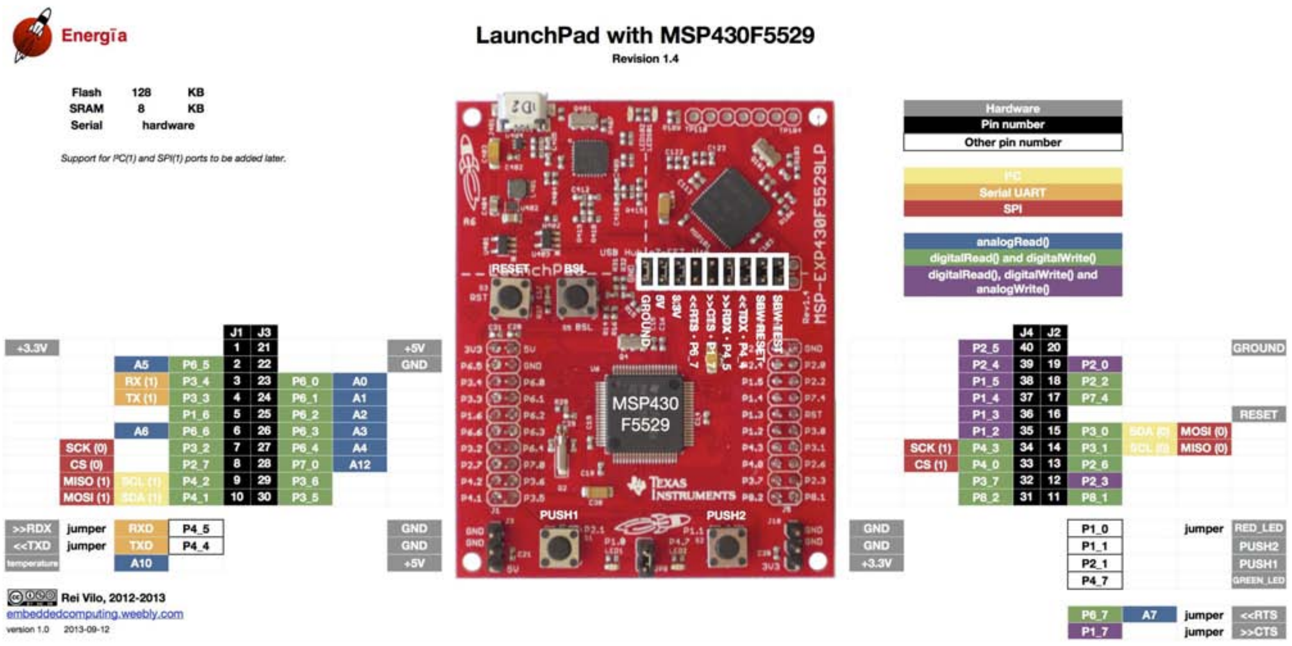

Energia
LaunchPad with MSP430F5529
Revision 1.4
Flash 128 KB
SRAM 8 KB
Serial hardware
Support for I²C(1) and SPI(1) ports to be added later.
Hardware
Pin number
Other pin number
I²C
Serial UART
SPI
analogRead()
digitalRead() and digitalWrite()
digitalRead(), digitalWrite() and analogWrite()
MSP430 F5529
PUSH1
PUSH2
Rei Vilo, 2012-2013
embeddedcomputing.weebly.com
version 1.0 2013-09-12

Fig 13: Launchpad Pinout

Peripherals are connected to the CPU using data, address, and control buses, and can be handled with all instructions.
The **instruction set** consists of the original 51 instructions with three formats and seven address modes and additional instructions for the expanded address range. Each instruction can operate on word and byte data.

Chapter 2. Sensors

Light intensity sensor / Photoresistor

The photoresistor(**photocell**) is a sensing module connected with the launchpad that helps measuring the level of ambient light.

A **photoresistor** or light-dependent resistor (LDR) or photocell is a light-controlled variable resistor. The resistance value of a photoresistor alters with changes of light intensity; in other words, it exhibits photoconductivity.
A photoresistor is made of a high resistance semiconductor. In the dark, a photoresistor can have a resistance as high as a few megaohms (MΩ), while in the light, a photoresistor can have a resistance as low as a few hundred ohms.

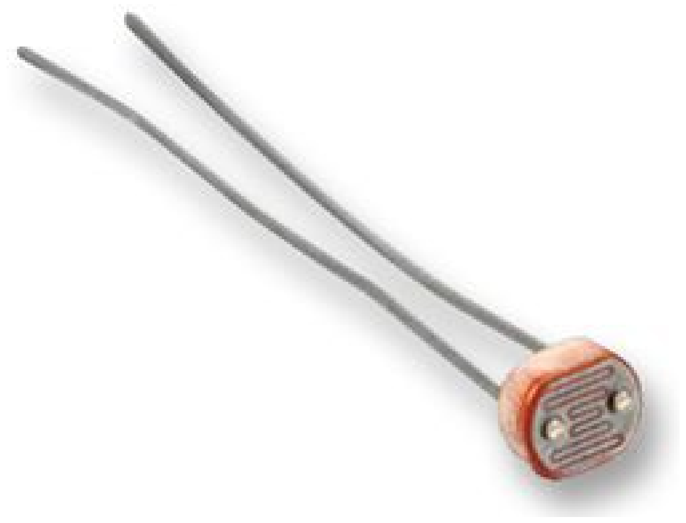


Fig 14: Photocell

When a light level of 1000 lux (bright light) is directed towards it, the resistance is 400ohms.
When a light level of 10 lux (very low light level) is directed towards it, the resistance has risen dramatically to 10.43Mohms.

Circuitry

Photocell is connected to a circuit in the following way. The one end connects to a 5V **supply voltage**, and the other one to **ground** through a **10K resistor**, forming a **voltage divider**.

The micro-controller is connected to the photoresistor with an analog pin, in order to acquire luminosity measurements by reading the **voltage** level.
This pin will be defined as **input** to our system in order to "read" the indicated value. This pin must be an **analog** one to be able to detect the entire range of values that come as an input. A strictly digital pin could only read two values, HIGH or LOW, which are useless in this case.

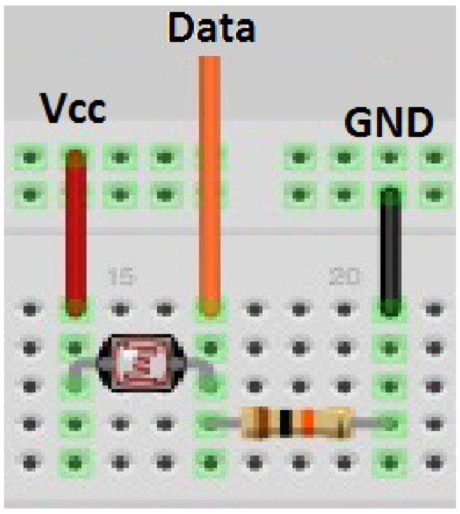


Fig 15: Photocell Connection

Of course the sensing module operates as a resistor and may be used directly on a circuit, for example to adjust the voltage level that a LED receives, according to light levels. However in our case the micro-controller reads the value provided by the photoresistor and detects the brightness in the room.

For this project we programmed the micro-controller to extract a value from a photocell using the analog-to-digital converter. Using this value we calculate the ambient light level. The **ADC(analog-to-digital converter)** compares two voltage values: a measured voltage and a specified reference voltage. Then it converts the difference to a numeric value and stores this value in a register. 0 to 5V difference is translated into 0 to 4095 output value.

Source code

The source code that implements this function is explained below:

```
unsigned long brightness=0.0;
```

The variable brightness that will keep the ambient light level percentage.

```
ADC12CTL0 = ADC12SHT02 + ADC12ON;         // Sampling time, ADC12 on
ADC12CTL1 = ADC12SHP;                     // Use sampling timer
ADC12IE = 0×01;                           // Enable interrupt
ADC12CTL0 |= ADC12ENC;
P6SEL |= 0×01;                            // P6.0 ADC option select
```

Here we set up the Analog-to-Digital Converter options, like sampling time, timer and input pin.
We also enable interrupts from ADC, in order to get the value when the conversion is complete.

```
 ADC12CTL0 |= ADC12SC;                    // Start sampling/conversion
_delay_cycles(15);

brightness=((long)ADC12MEM0/4095.0)*100;
```

At the end, a command given to the converter gets a sample and starts converting it. When the conversion is complete, the converted value is stored in ADC12MEM0 register. Based on the acquired value we calculate the percentage of the ambient light level.

Integrated temperature sensor

In the launchpad platform there is an **integrated temperature sensor**, located in the Reference module.
This **REF module** is a general purpose system that is used to generate reference voltages, and provide them to other embedded devices such as analog-to-digital converters, digital-to-analog converters, comparators, in order for them to be able to operate.

To make a temperature **measurement** with the sensor, the integrated analog-to-digital converter device is used.
This **converter** detects a voltage out of the sensor and stores it in the internal memory. Then we draw this value and we calculate the value of the temperature.

The temperature **calculating function** is a linear function. For the calculation we use some values for **calibration**, depending on what is the value of the reference voltage.
We get these values from specific addresses in the micro-controller's **memory**.

Typically the sensor is able to measure temperatures throughout the operating temperature range of the platform, which is from -40°C to +85°C.

Source code

For this project we programmed the micro-controller to detect temperature levels with the help of the integrated temperature sensor. The source code that implement this function is the following:

```
#define  CALADC12_15V_30C    *((unsigned  int  *)0x1A1A)      //  Temperature  Sensor
Calibration-30 C
#define  CALADC12_15V_85C    *((unsigned  int  *)0x1A1C)      //  Temperature  Sensor
Calibration-85 C
unsigned int temp;
volatile float temperatureDegC;
volatile float temperatureDegF;
```

Calibration values are set according to the **reference voltage** of 1.5V that we use. We also declare the **variables** that will keep the temperature values that are measured.

```
REFCTL0 &= ~REFMSTR;                      // Reset REFMSTR to hand over control to
                                            // ADC12_A ref control registers
ADC12CTL0 = ADC12SHT0_8 + ADC12REFON + ADC12ON;
                                            // Internal ref = 1.5V
ADC12CTL1 = ADC12SHP;                     // enable sample timer
ADC12MCTL0 = ADC12SREF_1 + ADC12INCH_10;  // ADC i/p ch A10 = temp sense i/p
ADC12IE = 0x001;                          // ADC_IFG upon conv result-ADCMEMO
__delay_cycles(100);                       // delay to allow Ref to settle
ADC12CTL0 |= ADC12ENC;
```

Reference voltage is set at 1.5V. The **converter** gets a **start** command. The **source** of the converter is the integrated temperature sensor, and the **register** which will store the measured value is ADC12MEM0. This is set by the ADC12CTL0 register.
Also, we enable an **interrupt** that indicates the completion of the **conversion** and write of the register ADC12MEM0.
Finally, a delay is added to give time to the reference module to settle, followed by a command that indicates the completion of the set up, and ability to start making measurements.

```
ADC12CTL0 |= ADC12SC;                   // Sampling and conversion start
_delay_cycles(60);
temp = ADC12MEM0;

// Temperature in Celcius
temperatureDegC = (float)(((long)temp - CALADC12_15V_30C) * (85 - 30)) /
            (CALADC12_15V_85C - CALADC12_15V_30C) + 30.0f;

// Temperature in Fahrenheit Tf = (9/5)*Tc + 32
temperatureDegF = temperatureDegC * 9.0f / 5.0f + 32.0f;
```

We give a **sampling** and **conversion** start command. The conversion is completed with the write of the ADC12MEM0. Then we get the measured value and calculate the temperature in Celsius and Fahrenheit using specific formulas.

The temperature values extracted using this sensor were not entirely correct. Usually there was a difference between the actual temperature and the temperature measured using the sensor. Very often, the measured temperature was 2 degrees higher.
The reason of this result is the fact that this sensor is integrated in the launchpad, and is effected by the launchpad's own temperature.

Sensirion SHT11

The Sensirion SHT11 sensor is connected to the platform and provides measurements of the environment **temperature** and **humidity** levels.
The sensor is known for its stability over time, low energy consumption and sturdiness.
This sensor integrates **signal processing elements** besides the data measurement, thereby it gives fully calibrated digital data. It also has an **analog-to-digital converter**, and **serial interface** for communication. These features result in increased signal quality output, fast response times and high tolerance to external interference.

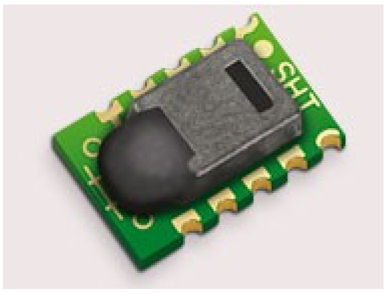


Fig 16: SHT11 sensor

It can measure **moisture** levels of from 0% to 100% accuracy ± 3.0%, and a **temperature** of from -40 ° C to 123,8 ° C and with accuracy ± 0,4 ° C.

The results of the measurements are converted either to 8bit humidity and 12bit temperature or 14bit humidity and 12bit temperature depending on a setting. The difference is the precision (resolution) of the measurement and of course the measuring time.
If the measurement has resolution 8bit, measuring time is 20ms, for 12bit resolution is 80ms and for 14bit resolution is 320ms. These times can be up to 30% smaller, depending on the power and measurement settings.

The supply voltage of the device must be between 2,4V and 5,5V. The recommended supply voltage is 3,3V.

The communication between the sensor and the micro-controller is done via an I/O data connection. Also the device gets an input clock that synchronizes the communication.

Memory

The sensor has an **OTP** (one-time programmable) **memory** which contains data for calibration of the device.
These calibration data are used after each measurement, so that the analog-to-digital module converts the data measured by sensing modules to values.
Then these values can be transmitted via the serial connection.
The results after each measurement and conversion are not stored in the memory, but kept in the communication circuit of the sensor, until the micro-controller restarts the clock. When this happens, the transmission of the results to the host starts.
The results are no longer kept in the sensor. In order for the host to get some more measurement results the process must start all over again.

Also, the device includes a 1Byte size status register, which contains several settings.

Specifically, the register's second bit (bit6) is a signal indicating **low voltage**. If the voltage on the sensor falls below 2,47V then after a measurement this bit becomes 1. This is useful when the platform is powered by a battery.

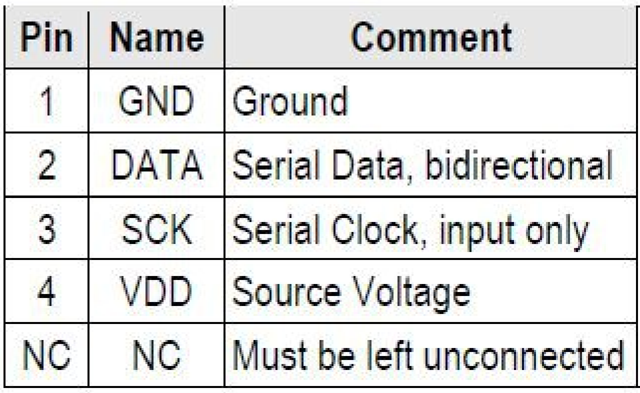
The 6th bit (bit2) shows whether the **embedded heater** that the sensor contains is **on**. This heater is an internal device that provides measurements up to 5 to 10 degrees Celsius higher than the ambient temperature. When enabled, the sensor measures its own temperature, which is used for analysis of the function of the sensor, for example measurements made before and after the heater is activated. The sensor is designed for non-continuous operation of this heater.

The seventh bit (bit1) adjusts the **measuring process**. When this bit is 0, the calibration values from the OTP memory, are loaded every time there is a measurement. This function can be disabled resulting in faster measurements by approximately 10ms.

The 8th bit (bit0) determines the **resolution** of the measurement. If it is 0 (which is the default value), then the measurements have 12bit resolution for humidity and 14bit for temperature . If the bit is altered to value 1, then the measurements are 8bit resolution for humidity and 12bit for temperature.

Circuitry

The sensor has 10 pins, but only 4 of them are used. One pin is connected to the supply voltage, another one is connected to the ground. There is one pin connecting the input clock which is responsible for the synchronization of the communication, and the fourth pin is an I/O pin for data send/receive between the sensor and the micro-controller. The pins are shown below:



Pin	Name	Comment
1	GND	Ground
2	DATA	Serial Data, bidirectional
3	SCK	Serial Clock, input only
4	VDD	Source Voltage
NC	NC	Must be left unconnected

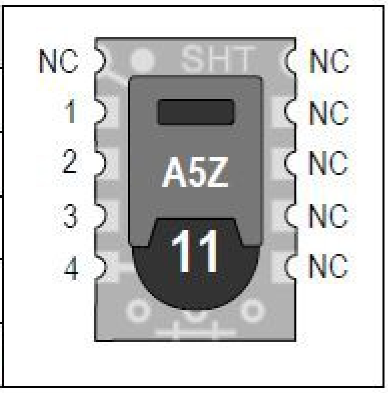


Fig 17: SHT11 pinout

The proper circuitry for the connection with the micro-controller is the following:

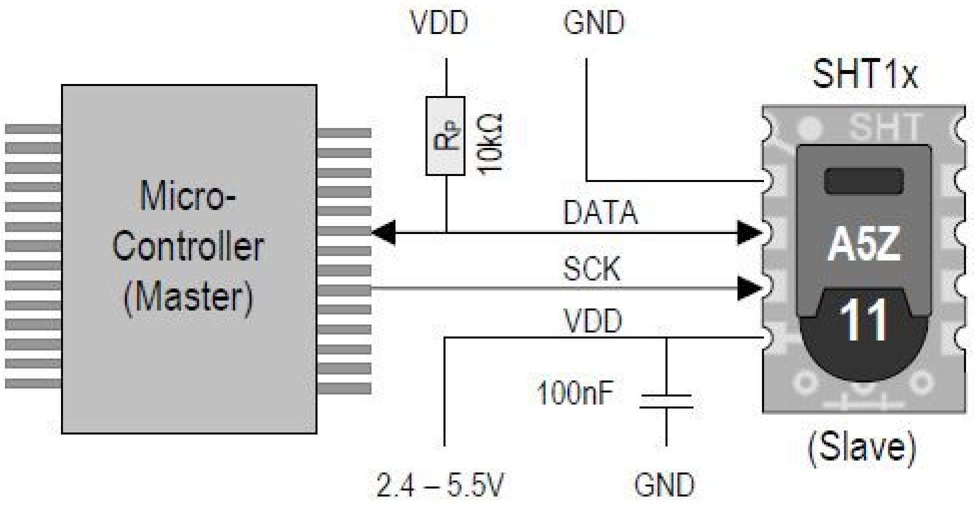


Fig 18: SHT11 Circuitry

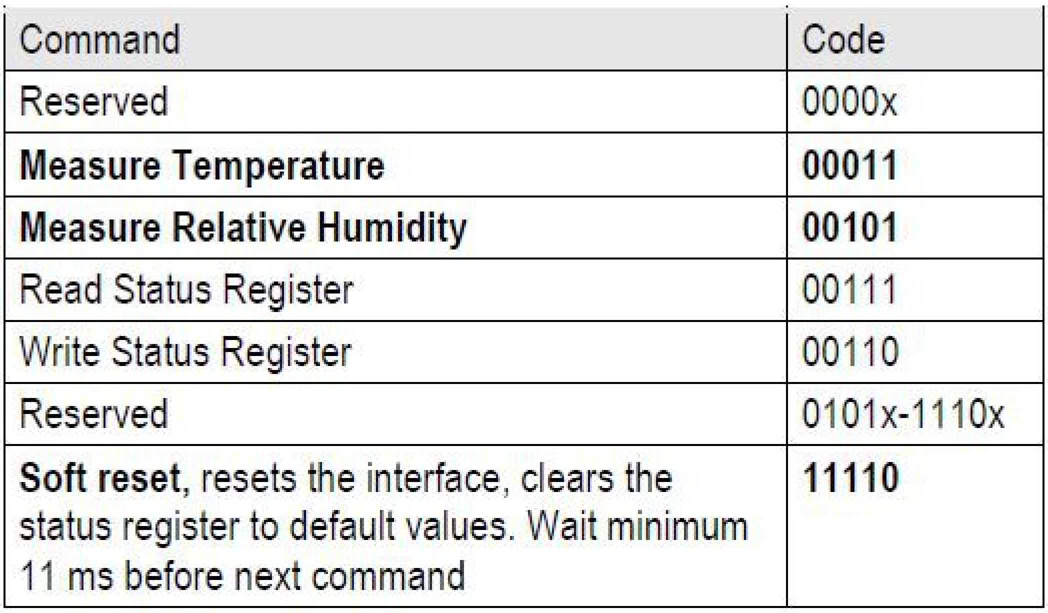
As seen, the data pin of the sensor is also connected to the supply voltage through a 10k resistor. This is a pull-up resistor, that brings the data pin to a HIGH state when the devices do not transmit a LOW signal. Also the supply pin is connected to the ground through a 100nF capacitor. The supply voltage that we used is the recommended 3.3V.

Communication

SCK is used to synchronize the communication between micro-controller and SHT11. Since the interface consists of fully static logic there is no minimum SCK frequency.
The DATA tri-state pin is used to transfer data in and out of the sensor. For sending a command to the sensor, DATA is valid on the rising edge of the serial clock (SCK) and must remain stable while SCK is high. After the falling edge of SCK the DATA value may be changed.
For reading data from the sensor, DATA is valid after SCK has gone low and remains valid until the next falling edge of SCK.
To avoid signal contention the micro-controller must only drive DATA LOW. The external pull-up resistor is required to pull the signal HIGH.

Communication process

The communication between the sensor and the micro-controller consists of some specific steps.
As a first step the sensor is powered up to chosen supply voltage VDD. The slew rate during power up shall not fall below 1V/ms. After power-up the sensor needs 11ms to get to Sleep State. No commands must be sent before that time.
To initiate a transmission, a Transmission Start sequence has to be issued. It consists of a lowering of the DATA line while SCK is HIGH, followed by a LOW pulse on SCK and raising DATA again while SCK is still HIGH.
After that the sensor is ready to read a command from the DATA line. The commands consist of three address bits and five command bits. The address bits are specified to 000. The command bits are shown below:



Command	Code
Reserved	0000x
Measure Temperature	**00011**
Measure Relative Humidity	**00101**
Read Status Register	00111
Write Status Register	00110
Reserved	0101x-1110x
Soft reset, resets the interface, clears the status register to default values. Wait minimum 11 ms before next command	**11110**

Fig 19: SHT11 Command Codes

The sensor indicates the proper reception of a command by pulling the DATA pin LOW (ACK bit) after the falling edge of the 8th SCK clock. The DATA line is released (and goes HIGH) after the falling edge of the 9th SCK clock.

Functionality

Measurement of Relative Humidity and Temperature:

After issuing a measurement command (‘00000101’ for relative humidity, ‘00000011’ for temperature) the controller has to wait for the measurement to complete. This takes a maximum of 20/80/320 ms for an 8/12/14bit measurement. The time varies with the speed of the internal oscillator and can be lower by up to 30%.
To signal the completion of a measurement, the SHT11 pulls data line LOW and enters Idle Mode. The controller must wait for this Data Ready signal before restarting SCK to read out the data.
Measurement data is stored until readout, therefore the controller can continue with other tasks and read out at its convenience. Two bytes of measurement data and one byte of CRC checksum (optional) will then be transmitted. The micro-controller must acknowledge each byte by pulling the DATA line low. All values are MSB first, right justified (e.g. the 5th SCK is MSB for a 12bit value, for an 8bit result the first byte is not used).
Communication terminates after the acknowledge bit of the CRC data. If CRC-8 checksum is not used the controller may terminate the communication after the measurement data LSB by keeping ACK high. The device automatically returns to Sleep Mode after measurement and communication are completed.
Important: To keep self heating below 0.1°C, SHT1x should not be active for more than 10% of the time – e.g. maximum one measurement per second at 12bit accuracy shall be made.

Connection reset sequence:

If communication with the device is lost the following signal sequence will reset the serial interface: While leaving DATA HIGH, toggle SCK nine or more times. This must be followed by a Transmission Start sequence preceding the next command. This sequence resets the interface only. The status register preserves its content.

Read-Write status register:

After the command Status Register Read or Status Register Write the content of 8 bits of the status register may be read out or written.
After each byte is transmitted, an acknowledgment is sent from the receiver.
In case of write status register, micro-controller sends data and sensor acknowledges the byte that it received. Data sent are 8bit.

In the case of read status register, the sensor sends data and the micro-controller sends an acknowledgment back for every received byte. The transmission is 2 bytes long, 1byte data and 1byte CRC.

Power Consumption

At sleep mode, the sensor is powered with 0.2 to 1.5 uA. At measuring mode, it is powered with 0.5 to 1mA.
Power consumption relies between 2 and 5uW in sleep mode. In measuring mode, power consumption goes up to 3mW.
These measurements took place in regular conditions, such as 25°C, and with 3.3V power supply.

Source code

For this project we connected the sensor to the micro-controller and programmed the system to get and calculate values for temperature and relative humidity.
The source code was based on the *SHT1x sample code* found in sensor's web page. The code that implements these functions is described below:

```
//Clock Pin P2_2
#define SCK_OUT  ( P2DIR |= BIT2 )
#define SCK_HIGH ( P2OUT |= BIT2 )
#define SCK_LOW  ( P2OUT &= ~(BIT2) )

//Data Pin P7_4
#define DATA_DIR_OUT    ( P7DIR |= BIT4 )
#define DATA_DIR_IN     ( P7DIR &= ~(BIT4) )
#define DATA_OUT_HIGH   ( P7OUT |= BIT4 )
#define DATA_OUT_LOW    ( P7OUT &= ~(BIT4) )
#define DATA_IN         ( P7IN  & BIT4 )
```

At this point, we set up the CLOCK and DATA pins. The pin 2.2 is set as clock input for the sensor. Pin 7.4 is set as data input/output.

```
//command codes
#define MEASURE_TEMP 0x03 //000 0001 1
#define MEASURE_HUMI 0x05 //000 0010 1
```

Next, we define the command codes that will be sent to the sensor to perform the appropriate measurement.

```
float temperature,humidity;

temperature=0.0;
humidity=0.0;

_delay_cycles(176000);    //wait 11ms for the sensor to get to Sleep
SCK_OUT;
```

Then we define the variables that will store the values of relative humidity and temperature. A delay is inserted here to give time to the sensor to get into sleep mode. Finally, pin 2.2 is set as output.

```
temperature = measure_temp();
_delay_cycles(16000000);  //wait 1s to prevent sensor overheating
humidity = measure_humi();
_delay_cycles(16000000);  //wait 1s to prevent sensor overheating
```

In the end, we call the functions that implement the measuring procedure. Between every one of them we insert a delay to keep the sensor self heating low.

These functions are described below:

```
float measure_temp(){

      int ackn, ret_crc, ret_value;
      float temperature;
      ackn=0;
      ret_crc=0;
      ret_value=0;
      temperature=0.0;

      s_transstart();
      ackn = s_write_byte(MEASURE_TEMP);
      _delay_cycles(5120000);   //wait 320ms for the sensor to complete the measurement

      ret_value = s_read_byte(1);
      ret_value = ret_value * 256;
      ret_value += s_read_byte(1);
      ret_crc = s_read_byte(0);
      temperature = ret_value * 0.01 - 39.7;

      return temperature;
}
```

```
float measure_humi(){

	int ackn, ret_crc, ret_value;
	float humidity;
	ackn=0;
	ret_crc=0;
	ret_value=0;
	humidity=0.0;

	s_transstart();
	ackn = s_write_byte(MEASURE_HUMI);
	_delay_cycles(1280000);   //wait 80ms for the sensor to complete the measurement

	ret_value = s_read_byte(1);
	ret_value = ret_value * 256;
	ret_value += s_read_byte(1);
	ret_crc = s_read_byte(0);
	humidity = -2.0468 + 0.0367*ret_value - 0.0000015955*ret_value*ret_value;

	return humidity;
}
```

The sequence of the commands starts with the transmission start command (`s_transstart();`).
After that, the command code for measuring relative humidity or temperature is sent to the sensor(`s_write_byte(MEASURE_HUMI);`).
Then a delay is inserted to give the sensor the necessary time to complete the measurement and the conversion (`_delay_cycles(1280000);`).
After that, the micro-controller reads three bytes of data from the sensor (`s_read_byte(1);`). Two bytes of the measured value and one byte of CRC. When we read the value, we calculate the level of temperature of relative humidity according to the right formula.

The functions we used are shown below:

```
void s_transstart(void){
  DATA_DIR_OUT;     //OUTPUT DATA
  DATA_OUT_HIGH;
  SCK_LOW;          //Initial state
  _delay_cycles(16);
  SCK_HIGH;
  _delay_cycles(16);
  DATA_OUT_LOW;
  _delay_cycles(16);
  SCK_LOW;
  _delay_cycles(32);
  SCK_HIGH;
  _delay_cycles(16);
  DATA_OUT_HIGH;
  _delay_cycles(16);
  SCK_LOW;
  DATA_OUT_LOW;
}
```

```
int s_write_byte(unsigned char value){
        unsigned char i,error=0;

        DATA_DIR_OUT;      // set data pin as output direction

        for (i=0x80;i>0;i/=2){ //shift bit for masking
                if(i & value){ DATA_OUT_HIGH; }//masking value with i , write to SENSI-BUS
                else { DATA_OUT_LOW; }
                sht11_pulse();
        }
        DATA_OUT_HIGH; //release DATA-line

        SCK_HIGH; //clk #9 for ack
        DATA_DIR_IN;     // set data pin as input direction

        if(DATA_IN){ error=1; }   //check ack (DATA will be pulled down by SHT11)
        SCK_LOW;

        return error; //error=1 in case of no acknowledge
}


int s_read_byte(unsigned char ack){
        unsigned char i,val=0;

        DATA_DIR_OUT;      // set data pin as output direction
        DATA_OUT_HIGH; //release DATA-line
        DATA_DIR_IN;     // set data pin as input direction

        for (i=0x80;i>0;i/=2){ //shift bit for masking
                SCK_HIGH; //clk for SENSI-BUS
                if (DATA_IN){ val=(val | i); }   //read bit
                _delay_cycles(32);
                SCK_LOW;
        }
        DATA_DIR_OUT;      // set data pin as output direction
        if(ack == 1) { DATA_OUT_LOW; } //in case of "ack==1" pull down DATA-Line
        else { DATA_OUT_HIGH; }

        sht11_pulse();
        DATA_OUT_HIGH; //release DATA-line
        return val;
}


void sht11_pulse(void)
{
    SCK_HIGH;    //CLOCK pulse -> approx 2us;
    _delay_cycles(16);
    SCK_LOW;
    _delay_cycles(16);
}
```

We used the default settings stored in the sensor's status register to complete the measurement of temperature and relative humidity.

Chapter 3. Wireless Communication

Xbee Series 2 Radio module

To enable wireless communication on the developed mote, we used the wireless transceiver Xbee Series 2(ZigBee Mesh). This module allows the establishment of complex mesh networks based on IEEE802.15.4 MAC Layer and ZigBee protocols. It allows a very reliable and simple communication between micro-controllers, computers, all kinds of systems that have a serial port. Point-to-point and multi-point networks are supported.

Some features of this module are:

- 3.3V @ 40mA
- 250kbps Max data rate
- 2mW output (+3dBm)
- 400ft (120m) range
- Built-in antenna
- Fully FCC certified
- 6 10-bit ADC input pins
- 8 digital IO pins
- 128-bit encryption
- Local or over-air configuration
- AT or API command set

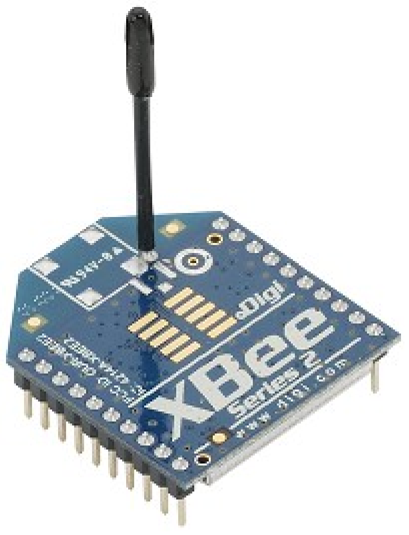


Fig 20: Xbee Series 2 module

The **Xbee Explorer USB** is used in order to set up the wireless module. Xbee Explorer USB is a device that implements a USB to serial interface. It connects the Xbee radio module and the computer. It supports all Series1 and Series2 standard and PRO versions.
In this way, the wireless module is connected to the computer, allowing the user to program the device, change the settings, and send data for it to transmit through the air.

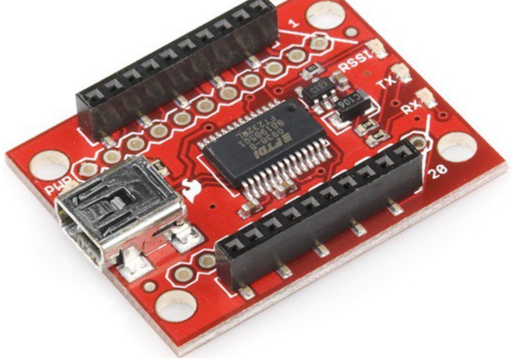


Fig 21: Xbee Explorer USB

Apart from these basic programming and data sending/receiving capabilities, Xbee Explorer has a **reset button**, a **voltage regulator** to supply the XBee with plenty of power, four **LEDs** that help debug the XBee: RX (receive data), TX (transmit data), RSSI (signal-strength indicator), and power indicator.

Configuration

For this project, we used **two Xbee modules**. One of them was connected to the **computer**, and the other was connected to the **micro-controller**. The transceivers communicate with each other transferring data between those devices. The purpose of this communication is to transfer temperature, relative humidity, and light level values from the micro-controller to a gateway node. In turn the gateway node will transfer the measurements to a Server machine. In our case we used a Xbee connected to a Windows machine acting as a gateway node.

In order to perform this function, we followed some specific steps, described below. Initially, we set up the two wireless modules with the Xbee Explorer. The software that we used for this, was **X-CTU** and **CoolTerm**.

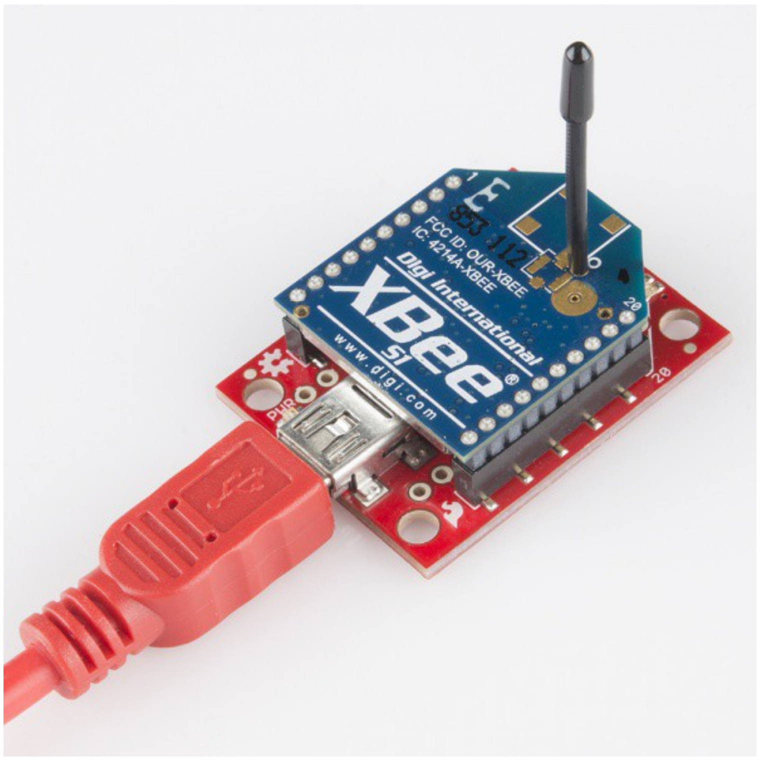


Fig 22: Xbee radio module connected to Xbee Explorer USB

At first we connected one of the modules to the computer, with the Xbee Explorer. We started the X-CTU application. The device was recognized as an Xbee Series2 module. The firmware was updated to set up the device as a **ZigBee coordinator AT**.

The coordinator is the one who defines the communication. It is **always ON**, and is the **root** device(parent). Default settings for the device are set, such as power level, packet encryption, Baud rate. The AT option defines the command set that controls the device. After this firmware update, CoolTerm was used to set the PAN ID (Personal Area Network ID) and the destination address, the MAC address of the other Xbee module.
Next, this device was disconnected from the computer and the Xbee Explorer, and the other wireless module was connected. This one was set as a **ZigBee End Device AT**. The End Device depends on another device which is the Coordinator, and acts as a parent of the End Device(child).
The End Device is capable of getting into Sleep state, in order to save power. Default settings for the device where also set automatically. Then we used CoolTerm to set the PAN ID, and the destination MAC address, ie the Coordinator's address.

The values of these settings were:
PAN ID: 2014
Coordinator MAC address: 0013A20040B14038
End Device MAC address: 0013A20040AD74E8
Baud Rate: 9600

Regarding the End Devices sleep mode, we tested cyclic sleep and pin wake-up options.

In **cyclic sleep**, the module sleeps for a specified time, and then wakes and sends a poll request to its parent every 100ms to discover if the parent has any pending data for the end device.
When it wakes up, the module will start a sleep timer (time until sleep). Any serial or wireless data received will restart the timer. The sleep timer value is settable. The module returns to sleep when the sleep timer expires.

Pin sleep allows the module to sleep and wake according to the state of the Sleep_RQ pin (pin 9). When Sleep_RQ is asserted (high), the module will finish any transmit or receive operations and enter a low power state. For example, if the module has not joined a network and Sleep_RQ is asserted (high), the module will sleep once the current join attempt completes (ie when scanning for a valid network completes). The module will wake from pin sleep when the Sleep_RQ pin is de-asserted (low).
When the XBee is awake and is joined to a network, it sends a poll request to its parent to see if the parent has any buffered data for it. The end device will continue to send poll requests every 100ms while it is awake.

When a router or coordinator receives a data packet intended for one of its end device children, it buffers the packet until the end device wakes and polls for the data, or until a packet buffering timeout occurs.

For cyclic sleep end devices, the coordinator must keep these messages for at least 1.2 times the maximum sleeping period of its children devices.
For pin wake up end devices, this time is not specific.

If an end device does not send a poll message to its parent for more than a specified time, the parent will assume that the end device has moved out of range and will remove it from its child table. This timeout called cyclic sleep period is settable between 1 and 84 seconds. Another option allows the end device not to poll its parent for a number of cyclic sleep periods. This leads to a maximum poll timeout that exceeds 2 months.
These settings allow routers and coordinators to be responsive to changing network conditions.

After we set up both the wireless modules, we connected the coordinator to the computer with the Xbee Explorer USB, and the end device with the micro-controller, using an adapter.

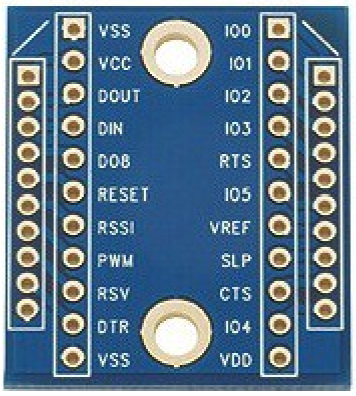


Fig 24: Xbee adapter board

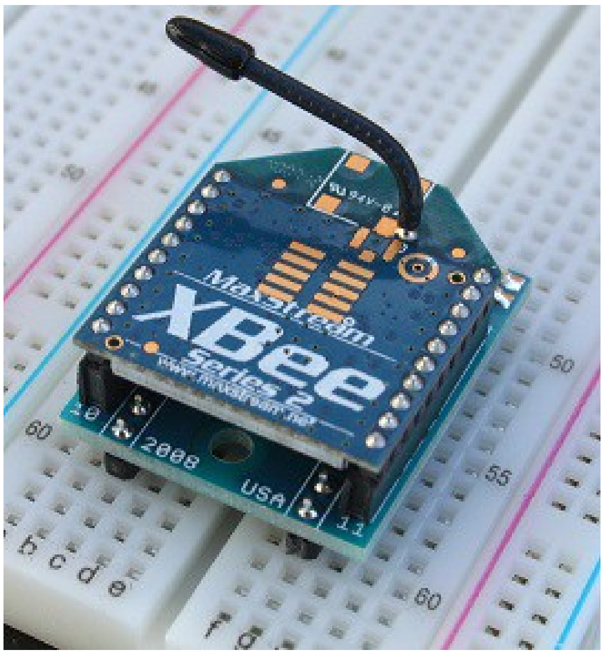


Fig 23: Xbee radio module connected to a breadboard using a Xbee adapter board

The adapter board helps connecting the Xbee module to the micro-controller by converting the Xbee's 2mm spacing to breadboard friendly spacing.

Pin 1 of the Xbee module was connected to a 3.3V **supply voltage**, pin 10 was connected to the **ground**, and pins 2 and 3(Data In and Data Out) were connected to the micro-controller's **UART** interface pins 3.4 and 3.3. The **wake up** pin DTR(pin 9) of the Xbee module was connected to the micro-controller's pin 1.6 and the **on/sleep** pin(pin 13) was connected to the pin 2.7.

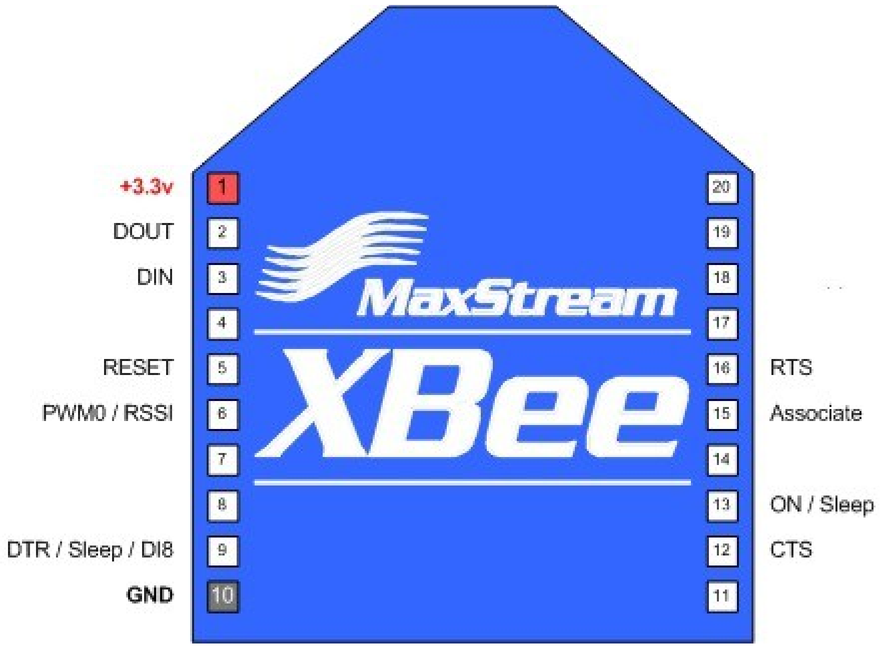


Fig 25: Xbee Serries2 radio module pinout

1st implementation: End Device with pin wake-up

First, we tried the implementation where the micro-controller controls the function of the Xbee radio module connected to it. The Xbee's sleep mode was configured with the option **Pin Hibernate**, using X-CTU. The micro-controller sends a Hi-to-Lo transition to Xbee's Sleep-RQ pin(pin 9). In a short period of time, usually some milliseconds, the radio module turns to **ON** state. In contrary to that, when the Sleep-RQ pin is asserted (high) by the micro-controller, the Xbee completes its current function and gets to **Sleep** mode.

This implementation sets the micro-controller to operate following some steps. These steps are executed repeatedly. We describe them next:

- Micro-controller operating in **low power mode**. Operating in low power mode means that the main cpu clock is turned off, and the micro-controller stays idle according to some specified setting. This feature will be described thoroughly later.

- After a period of time, a **timer** expires and causes the micro-controller to exit low power mode and switch to **active mode**. Entering/exiting low power mode and configuring timers is described later.
- When the micro-controller switches to active mode, it starts communicating with the sensors and acquires **measurement** values.
- Then it calculates and stores temperature, relative humidity and light density.
- After that, writes this data to a **string variable**, to be able to send it to the Xbee module.
- Sends a LOW signal to Xbee's **Sleep-RQ** pin, forcing it to switch to **active mode**.
- When the Xbee turns ON, the micro-controller **sends** each one of the string variable's characters.
- The Xbee **transmits** the characters through air, the Xbee connected to the **host** PC **receives** the data.
- After the completion of the transmission, the micro-controller sends a HIGH signal to the wireless module's **Sleep-RQ** pin, forcing it to switch back to **sleep mode**.
- Finally, the **micro-controller** itself switches back to **low power mode**, and this execution sequence starts all over again.

The basic advantage of this implementation is the fact that the Xbee is ON for the minimum time possible. Thus, the power consumption is minimized. Also, the micro-controller wakes up only on certain periods of time, unlike the 2nd implementation. As a result, the micro-controller's power consumption is minimized too.
The disadvantage of this implementation is the fact that it is necessary to communicate with a host that is always ON, in order to successfully send data. The two wireless modules cannot be synchronized, because the micro-controller selects the state(Sleep/ON) of the End Device.

2nd implementation: End Device with cyclic sleep

For the second implementation, we configured the Xbee to **Cyclic Sleep** sleeping mode. This means that the radio module goes to sleep for a specified amount of time, and then turns to ON mode by itself. After that, it stays ON waiting for packets from the serial input to transmit or the wireless network to receive. When there is no input from the serial or wireless channel for some time, the Xbee gets back to sleep state.
The *Cyclic Sleep Period* option determines the duration of the time that the Xbee will stay in sleep mode. The *Time Before Sleep* option determines how much time Xbee will wait ON for messages until it switches to Sleep state(when a message is received, serial or wireless, this time is restarted).
We configured these options using X-CTU. *Cyclic Sleep Period* was set to 3000ms and *Time Before Sleep* was set to 1000ms.

For the Xbee set in coordinator mode, *Cyclic Sleep Period* option determines the transmission timeout when sending to a sleeping end device. Also determines the duration the parent will buffer a message for a sleeping child. We set it to 3000ms as well.
Another option that we set on the Coordinators configuration was *Number of Cyclic Sleep Periods.* This option determines the number of cyclic sleep periods used to calculate end device poll timeout. If an end device does not send a poll request to its parent coordinator or router within the poll timeout, the end device is removed from the child table. The poll timeout is calculated by 3**Cyclic Sleep Period*Number of Cyclic Sleep Periods.* In the context of the project, we set the *Number of Cyclic Sleep Periods* to 100, in order to rule out the possibility that the End Device is removed from the Coordinator's child table.
Implementing this operation, we programmed the micro-controller to take measurements from the sensors and get to a Low Power mode for some time. Every once in a while, the micro-controller wakes up again, takes measurements, and gets back to LPM state.
The Xbee itself wakes up periodically.

The execution is split into two parts:
<u>Measuring part</u>

- Micro-controller is in low power mode.
- After a period of time, a timer causes the micro-controller to wake up.
- Then, the micro-controller starts communicating with the sensors and acquires **measurement** values.
- Then it calculates and stores temperature, relative humidity and light density.
- After that, writes this data to a **string variable**, to be able to send it to the Xbee module.
- Finally, the micro-controller switches back to sleep mode.

<u>Transmitting part</u>

- The Xbee module operates in Sleep mode.
- After a period of time, it wakes up.
- When this happens, a HIGH signal is sent to a micro-controller's pin, causing an interrupt.
- When this interrupt is detected, the micro-controller pauses its current function/state and sends each one of the string variable's characters.
- The Xbee **transmits** the characters through air, the Xbee connected to the **host** PC **receives** the data.
- After the completion of the transmission, the Xbee stays ON waiting for input.
- The micro-controller returns to its previous state.
- After an idle time, the Xbee module switches back to Sleep mode.

The execution of these parts is independent. The transmitting part can happen at any point of the measuring part.

The disadvantage of this implementation is the fact that the Xbee stays ON waiting for possible input before it switches back to sleep mode. Thus, the power consumption is greater compared to the first implementation. Also, the micro-controller might wake up two times. One to perform the measuring part, and one for the transmitting part. As a result, the micro-controller's power consumption is increased.
The advantage of this implementation is the fact that it is not necessary to communicate with a node that is always ON, in order to successfully send data. The two wireless modules can be synchronized to wake up at the same time, exchange messages and switch back to sleep mode.

Port/Pin Interrupt

For the second implementation, our goal was to write a firmware that will signal the micro-controller to switch to operation mode to send the data through the radio module, when the XBee wakes up.
In order to implement this, we used Xbee's Sleep/ON pin(pin 13). This pin is asserted HIGH when the radio module is ON, and LOW when the radio module is in Sleep state. We connected this pin to the pin 2.7 of the micro-controller. Then we set pin 2.7 as input pin. We programmed the micro-controller to detect an interrupt when a Low-to-High transition occurs at pin 2.7. When this interrupt occurs, we handle it by sending data(measured values from sensors) to the Xbee to transmit.

Source Code

To program the device in order to be able to detect this interrupt we followed the next steps:

First of all, we select the interrupt edge for the specific pin(P2.7), to Low-to-High.

```
P2IES &= ~BIT7;
```

P2IES is the 8bit register that determines the interrupt edge for each of the eight Port2 pins.
0 → Low-to-High transition
1 → High-to-Low transition

Next we clear the interrupt flag for pin P2.7

```
P2IFG &= ~BIT7;
```

P2IFG is the 8bit register that keeps the interrupt flags for the eight pins of Port2.
0 → No interrupt pending
1 → Interrupt pending

Also we enable the port2 interrupts for pin P2.7

```
P2IE |= BIT7;
```

P2IE is the 8bit register that enables/disables interrupts for the pins of Port2.
0 → Interrupt disabled
1 → Interrupt enabled

After this configuration, the micro-controller is programmed to detect Low-to-High transitions that occur at pin P2.7

When this interrupt occurs, the micro-controller sends the measured values to the Xbee radio module via serial UART interface. This process is described below.

UART Interface

The MSP430F5529 micro-controller, has an integrated **eUSCI**(enhanced Universal Serial Communication Interface), interface that allows multiple serial communication modes from a single hardware module.
Specifically, the micro-controller contains two Universal Serial Communication Interface modules, called **USCI0** and **USCI1**. Each one of them provides two interfaces, **USCI_A** and **USCI_B**.
USCI_A interface supports **UART**, **IrDA** and **SPI** protocols, and USCI_B supports **SPI** and **I2C**.

For this project, we used the asynchronous UART communication to transfer data between the micro-controller and the Xbee wireless module. Using UART mode, the USCI modules connected the device with an external system through two pins, RXD and TXD. The first one is data input and the second data output. The module transmits and receives UART characters to/from the external system with a specific rate.

To be able to communicate using the UART protocol, we have to configure the interface. The configuration is made through registers.

First of all, to configure the USCI module, we have to put the module's state in reset.

```
UCA0CTL1 |= UCSWRST;
```

This command sets the module in reset state. UCACTL1 is the first UART control register.

After this we can configure the module.

```
UCA0CTL1 |= UCSSEL_2;
```

SMCLK is selected as a clock source.

```
UCA0BR0 = 6;
UCA0BR1 = 0;
```

These two registers determine the Baud Rate.
The 16-bit value of UCA0BR0 + UCA0BR1 × 256 is used to calculate the Baud Rate of the interface.

```
UCA0MCTL = UCBRS_0 + UCBRF_13 + UCOS16;
```

UCBRS_0 and UCBRF_13 determine the first and second modulation stage.
UCOS16 enables oversampling.

After the completion of the configuration we force the module to exit reset state and be initialized.

```
UCA0CTL1 &= ~UCSWRST;
```

The last command de-asserts the reset bit which was set by the first command.

After this, the interface is ready. A string contains the data that need to be sent to the radio module. To send data over the USCI module to the Xbee radio module, all we have to do is to put the data byte-by-byte in the UART's transmit buffer.

```
for(i=0;i<240;i++){
	while (!(UCA0IFG&UCTXIFG));		// USCI_A0 TX buffer ready?
	UCA0TXBUF = String[i];
}
```

A *for* loop serves this functionality, writing every byte into the transmit buffer. Each byte/character is written to the buffer after a check, ensuring that the buffer is ready(ie the previous character was transmitted).

Chapter 4. Implementation Features

Except of the basic mote's functionality which is measuring environmental information and transmitting them to another network node, we implemented some special features. These features give the node the following attributes:

- The ability to **reset**, and restart its execution, if a software error occurs. The causes of this error could be external disturbances, hardware malfunctions or even software bugs. This feature is provided by the **watchdog** timer.
- **Low power mode**. This feature allows the micro-controller to turn of system clocks and integrated modules to **save power**. There are several low power modes available, to select according to the demands. **Exiting** a low power mode and switching back to active mode is possible through **interrupts** caused by many reasons. In our implementation, exiting a low power mode is caused by a **timer** interrupt.

In this chapter, we describe the functions and modules that provide these features, and we analyze the study and procedure followed in order to add them to our development.

Watchdog

The watchdog timer is a 32-bit timer that can be used as a watchdog or as an interval timer.
The primary function of the watchdog timer (WDT_A) module is to perform a controlled system restart after a software problem occurs. If the selected time interval expires, a system reset is generated. If the watchdog function is not needed in an application, the module can be configured as an interval timer and can generate interrupts at selected time intervals.

Features of the watchdog timer module include:

- Eight software-selectable time intervals
- Watchdog mode
- Interval mode
- Password-protected access to Watchdog Timer Control (WDTCTL) register
- Selectable clock source
- Can be stopped to conserve power
- Clock fail-safe feature

After a PUC, the WDT_A module is automatically configured in the watchdog mode with an initial ~32-ms reset interval using the SMCLK. The user must setup or halt the WDT_A prior to the expiration of the initial reset interval.

The watchdog timer module can be configured as either a watchdog or interval timer with the **WDTCTL** register. WDTCTL is a 16-bit password-protected read/write register. Any read or write access must use word instructions and write accesses must include the write password 05Ah in the upper byte. Any write to WDTCTL with any value other than 05Ah in the upper byte is a password violation and triggers a PUC system reset, regardless of timer mode. Any read of WDTCTL reads 069h in the upper byte. Byte reads on WDTCTL high or low part result in the value of the low byte. Writing byte wide to upper or lower parts of WDTCTL results in a PUC.

The **WDTCNT** is a 32-bit up counter that is not directly accessible by software. The WDTCNT is controlled and its time intervals are selected through the Watchdog Timer Control (WDTCTL) register. The WDTCNT can be sourced from SMCLK, ACLK, VLOCLK, or X_CLK on some devices. The clock source is selected with the WDTSSEL bits. The timer interval is selected with the WDTIS bits.

Watchdog Mode

After a PUC condition, the WDT module is configured in the watchdog mode with an initial ~32ms reset interval using the SMCLK. The user must setup, halt, or clear the watchdog timer prior to the expiration of the initial reset interval or another PUC is generated. When the watchdog timer is configured to operate in watchdog mode, either writing to WDTCTL with an incorrect password, or expiration of the selected time interval triggers a PUC. A PUC resets the watchdog timer to its default condition.

Watchdog Timer Interrupts

When using the watchdog timer in the watchdog mode, the WDTIFG flag sources a reset vector interrupt. The WDTIFG will self clear upon a watchdog timeout event. The SYSRSTIV can be read to determine if the reset was caused by a watchdog timeout event. When using the watchdog timer in interval timer mode, the WDTIFG flag is set after the selected time interval and requests a watchdog timer interval timer interrupt if the WDTIE and the GIE bits are set. The interval timer interrupt vector is different from the reset vector used in watchdog mode. In interval timer mode, the WDTIFG flag is reset automatically when the interrupt is serviced, or can be reset with software.

Clock Fail-Safe Feature

The WDT_A provides a fail-safe clocking feature, ensuring the clock to the WDT_A cannot be disabled while in watchdog mode. This means that the low-power modes may be affected by the choice for the WDT_A clock. If SMCLK or ACLK fails as the WDT_A clock source, VLOCLK is automatically selected as the WDT_A clock source. When the WDT_A module is used in interval timer mode, there is no fail-safe feature within WDT_A for the clock source.

Watchdog time intervals and clock sources

There are eight different time intervals for the watchdog. They are software selectable.
The watchdog counter counts from zero to a specified value. This value is determined by the time interval select bits.
A clock source is selected to do the counting.
The time that will take the watchdog to expire depends on the time interval and the clock speed(frequency).

Watchdog Control Registers and Source Code

The 16-bit watchdog control register WDTCTL contains a field (bits 6-5) for clock selection. Clock source is selected according to the following way:

WDTCTL bits 6-5		Selected Clock Source
00	→	SMCLK
01	→	ACLK
10	→	VLOCLK

value 11 is reserved.

The WDTCTL's 2-0 bits configure the watchdog time intervals according to the following way:

WDTCTL bits 2-0		Selected Time Interval
000	→	2^31
001	→	2^27
010	→	2^23
011	→	2^19
100	→	2^15
101	→	2^13
110	→	2^9
111	→	2^6

When the watchdog clock source is ACLK (32KHz), it is calculated that if 000 is selected as time interval, the watchdog timer will expire after 18h:12m:16s. When 111 is selected as time interval, the watchdog timer will expire after 1.95ms.

In the context of the project, we selected ACLK as clock source, and 011 as time interval. The watchdog timer is supposed to expire after 16seconds (2^19 / (32*(2^10) Hz) = 16s).
Although, when testing, we found out that with these options the watchdog expires after about 1min.

We have direct access to the watchdog control register. We configure it with the following command:

```
WDTCTL = WDTPW+WDTCNTCL+WDTSSEL0+WDTIS_3;
```

WDTPW is the password 5Ah written to bits 15-8
WDTCNTCL clears the watchdog counter to 0. The watchdog is restarted.
WDTSSEL0 selects the watchdog clock source to ACLK.
WDTIS_3 selects the watchdog time interval to 2^19.

Every time the micro-controller takes measurements, we execute this command to restart the watchdog timer. If for any reason the micro-controller is not taking measurements, this timer will not be restarted, and will expire causing the system to reset (PUC, Power-up Clear reset).

Low Power Modes

Operating Modes

The MSP430 family is designed for ultra-low-power applications and uses different operating modes. It has one active mode and six software selectable low-power modes of operation. The operating modes take into account three different needs:
• Ultra-low power
• Speed and data throughput
• Minimization of individual peripheral current consumption

An interrupt event can wake up the device from any of the low-power modes, service the request, and restore back to the low power mode on return from the interrupt program.

<u>The following seven operating modes can be configured by software:</u>

- Active mode (AM)
  – All clocks are active
- Low-power mode 0 (LPM0)
  – CPU is disabled
  – ACLK and SMCLK remain active, MCLK is disabled
  – FLL loop control remains active

- Low-power mode 1 (LPM1)
  - CPU is disabled
  - FLL loop control is disabled
  - ACLK and SMCLK remain active, MCLK is disabled
- Low-power mode 2 (LPM2)
  - CPU is disabled
  - MCLK and FLL loop control and DCOCLK are disabled
  - DCO's dc-generator remains enabled
  - ACLK remains active
- Low-power mode 3 (LPM3)
  - CPU is disabled
  - MCLK, FLL loop control, and DCOCLK are disabled
  - DCO's dc generator is disabled
  - ACLK remains active
- Low-power mode 4 (LPM4)
  - CPU is disabled
  - ACLK is disabled
  - MCLK, FLL loop control, and DCOCLK are disabled
  - DCO's dc generator is disabled
  - Crystal oscillator is stopped
  - Complete data retention

Status Register

Status register(SR) is a 16bit register. The register's 7-3 bits control the operating mode of the micro-controller.

Specifically:

bit 7:	SCG1	System clock generator 1
bit 6:	SCG0	System clock generator 0
bit 5:	OSCOFF	Oscillator off
bit 4:	CPUOFF	CPU off
bit 3:	GIE	General interrupt enable

To force the micro-controller to enter low power modes, we configure these status register bits.
To get to LPM0 we set the SR's bit 4(CPUOFF).
To get to LPM1 we set the SR's bit 4 and 6 (CPUOFF + SCG0).
To get to LPM2 we set the SR's bit 4 and 7 (CPUOFF + SCG1).
To get to LPM3 we set the SR's bit 4, 6 and 7 (CPUOFF + SCG0 + SCG1).
To get to LPM4 we set the SR's bit 4, 5, 6 and 7 (CPUOFF +OSCOFF + SCG0 + SCG1).

To enable interrupts during a low power mode, we also set the SR's bit 3 (GIE).

The advantage of including the CPUOFF, OSCOFF, SCG0, and SCG1 mode-control bits in the SR is that the present operating mode is saved onto the stack during an interrupt service routine.
Program flow returns to the previous operating mode if the saved SR value is not altered during the interrupt service routine. Program flow can be returned to a different operating mode by manipulating the saved SR value on the stack inside of the interrupt service routine. When setting any of the mode control bits, the selected operating mode takes effect immediately.
Peripherals operating with any disabled clock are disabled until the clock becomes active. Peripherals may also be disabled with their individual control register settings.
All I/O port pin configurations and RAM/registers are left unchanged.

Wake up from LPM0 through LPM4 is possible through all enabled interrupts.

Exiting Low-Power Modes LPM0 Through LPM4

An enabled interrupt event wakes the device from low-power operating modes LPM0 through LPM4. The program flow for exiting LPM0 through LPM4 is:

- Enter interrupt service routine
    - ➢ The PC and SR are stored on the stack.
    - ➢ The CPUOFF, SCG1, and OSCOFF bits are automatically reset.
- Options for returning from the interrupt service routine
    - ➢ The original SR is popped from the stack, restoring the previous operating mode.
    - ➢ The SR bits stored on the stack can be modified within the interrupt service routine returning to a different operating mode when the RETI instruction is executed.

In the context of this project, we programmed the micro-controller to enter LPM3 when it is idle.
The reason we chose this low power mode, is that the ACLK is active during LPM3.
When entering LPM3 we initiate a timer. When the timer expires(overflows), the micro-controller switches to active mode and takes measurements from the sensors. In order to be able to do that, the timer must be active during LPM3.
The timer's clock source is set to be ACLK. LPM4 disables this clock.
Thus we selected LPM3, which has the lowest power consumption after LPM4.
Of course, we also set the GIE bit, in order to enable the interrupt that will occur during LPM3 and force exit to active mode, continuing execution right below the enter LPM command.

The functions that we used to configure SR in order to force the micro-controller to enter and exit LPM3 are:

```
__bis_SR_register();
```

and

```
__bic_SR_register_on_exit();
```

Entering LPM3: `__bis_SR_register(SCG1+SCG0+CPUOFF+GIE);`

Exiting LPM3: `__bis_SR_register_on_exit(SCG1+SCG0+CPUOFF+GIE);`

In the second implementation, the Xbee radio module can also wake up the micro-controller with a pin interrupt as described. The difference is that when this interrupt occurs, the micro-controller sends a message to the radio module and then the SR(state register) is restored, returning the system to the previous state. If it was operating in LPM3 it will switch back to it. If it was operating in active mode, it will switch back to it and continue with the execution from the point it stopped when the interrupt occurred.
Only when the timer interrupt occurs we execute the __bic_SR_register_on_exit command and the micro-controller is forced to get to active mode.

Timers

The micro-controller must exit low power mode after a short period of time. To specify and select how long will the system stay in low power mode and when it will get back to active mode to obtain measure values from the sensors, we use a timer.

The micro-controller has four different timers. Three timers of the TIMER_A family(TIMER0_A5, TIMER1_A3, TIMER2_A3) and one timer of the TIMER_B family(TIMER0_B7).
In this case, we use TIMER0_A5.

Timer_A is a 16-bit timer/counter with up to five capture/compare registers.
TIMER0_A5 has five, TIMER1_A3 and TIMER2_A3 have three.
Timer_A module is configured with user software, can support multiple capture/compares, PWM outputs, and interval timing. It also has extensive interrupt capabilities. Interrupts may be generated from the counter on overflow conditions and from each of the capture/compare registers.

Timer_A features include:

• Asynchronous 16-bit timer/counter with four operating modes
• Selectable and configurable clock source
• Up to five configurable capture/compare registers
• Configurable outputs with pulse width modulation (PWM) capability
• Asynchronous input and output latching
• Interrupt vector register for fast decoding of all Timer_A interrupts

Timer_A registers

TIMER0_A5 has fourteen 16bit registers, in order to operate. These are:

• TA0CTL:	Control Register
• TA0CCTL0:	Capture/Compare Control 0
• TA0CCTL1:	Capture/Compare Control 1
• TA0CCTL2:	Capture/Compare Control 2
• TA0CCTL3:	Capture/Compare Control 3
• TA0CCTL4:	Capture/Compare Control 4
• TA0R:	Counter
• TA0CCR0:	Capture/Compare 0
• TA0CCR1:	Capture/Compare 1
• TA0CCR2:	Capture/Compare 2
• TA0CCR3:	Capture/Compare 3
• TA0CCR4:	Capture/Compare 4
• TA0IV:	Interrupt Vector
• TA0EX0:	Expansion

Capture/Compare registers 1-4 are used to record a time event, generate output signal with modulated width, or cause an interrupt at specific time intervals in continuous mode. **TA0CCR0** is used as a time interval in up mode.
In our case, they are not used. The timer's time interval is 0xFFFF, the maximum value of the 16bit counter.

TA0R: 16-Bit Timer Counter
The 16-bit timer/counter register, TA0R, increments or decrements (depending on mode of operation) with each rising edge of the clock signal. TA0R can be read or written with software. Additionally, the timer can generate an interrupt when it overflows.

TA0CTL: Control Register
The 16-bit timer control register is responsible for the configuration and operation of the timer. It contains six programmable fields.

TASSEL (bits 9-8) selects the timer's clock source.
The timer clock can be sourced from ACLK, SMCLK, or externally via TA0CLK or INCLK.

ID (bits 7-6) determines the clock divider.
The selected clock source may be passed directly to the timer or divided by 2, 4, or 8, using the ID bits.

MC (bits 5-4) configures timer's mode.
There are four modes. Stop mode, the timer is halted. Up mode, the timer counts up to TA0CCR0. Continuous mode, the timer counts up to 0xFFFF. Up/down mode, the timer counts up to TA0CCR0 and then down to 0.

TACLR (bit 2) clears the timer.
Setting this bit resets TA0R, the timer clock divider logic, and the
count direction. The TACLR bit is automatically reset and is always read as zero.

TAIE (bit 1) enables timer interrupt.
This bit enables the TAIFG interrupt request.

TAIFG (bit 0) interrupt flag.
0 = No interrupt pending
1 = Interrupt pending

Bits 15-10 and 3 are reserved.

TA0IV: Interrupt Vector

TA0IV is a register used to identify and separate the different interrupts that can be caused by the timer. According to the interrupt source, TA0IV takes a value that we use to identify and handle the interrupt the desired way .

00h	No interrupt pending
02h	Interrupt source: Capture/Compare 1 (Highest Priority)
04h	Interrupt source: Capture/Compare 2
06h	Interrupt source: Capture/Compare 3
08h	Interrupt source: Capture/Compare 4
0Ah	reserved
0Ch	reserved
0Eh	Interrupt source: Timer overflow (Lowest Priority)

TA0EX0: Expansion

The selected clock source can be further divided by 2, 3, 4, 5, 6, 7, or 8 using the TAIDEX bits.

After programming ID or TAIDEX bits, we set the TACLR bit. This clears the contents of TA0R and resets the clock divider logic to a defined state.
The clock dividers are implemented as down counters. Therefore, when the TACLR bit is cleared, the timer clock immediately begins clocking at the first rising edge of the Timer_A clock source selected with the TASSEL bits and continues clocking at the divider settings set by the ID and TAIDEX bits.

The timer starts when **MC** field of TA0CTL is set to a clock, and this clock is active, and stops when **MC** is set to 00. When this happens TA0R is automatically set to 0.

Modifying Timer_A registers

It is recommended to stop the timer before modifying its operation (with exception of the interrupt enable, interrupt flag, and TACLR) to avoid errant operating conditions.
When the timer clock is asynchronous to the CPU clock, any read from TAxR should occur while the timer is not operating or the results may be unpredictable. Alternatively, the timer may be read multiple times while operating, and a majority vote taken in software to determine the correct reading. Any write to TAxR takes effect immediately.

Configuration

The Configuration that we chose for this project is the following:

Clock Source:	ACLK
Input clock divider:	CLK/4
Mode:	Continuous
Interrupts:	Enable interrupts
Expansion divider:	CLK/1

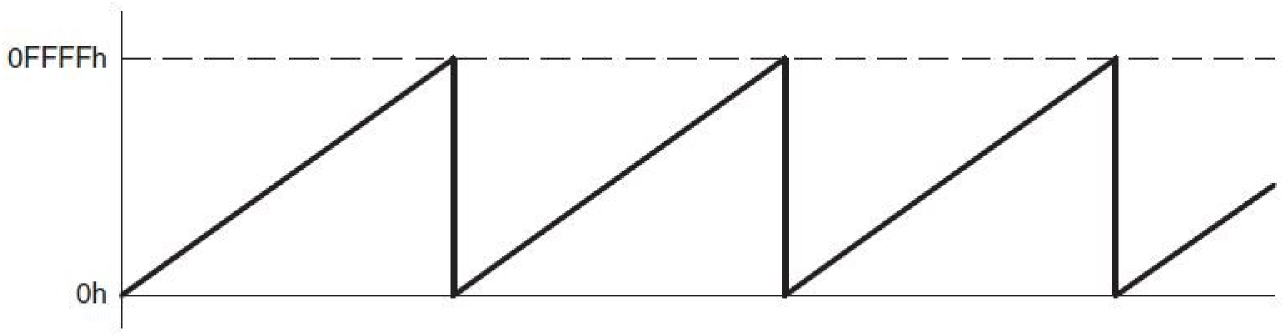


Fig 26: Timer_A continuous mode

With these options, the micro-controller will exit LPM3 after six seconds.

Source code

```
TA0EX0 = TAIDEX_0;
TA0CTL = TASSEL_1 + MC_0 + ID_2 + TACLR + TAIE;
```

Configure initial timer state.

Right before the micro-controller enters LPM3 we start the timer.

```
TA0CTL |= MC_2;
```

When an interrupt occurs and the micro-controller must exit LPM3 we detect the interrupt source. We use the function `__even_in_range` in order to do that.

```
__even_in_range(TA0IV,14)
```

If the interrupt occurs in vector 14 (0Eh) the interrupt source is an overflow of the timer.

When this happens, we stop the timer and force the micro-controller to exit LPM3.

```
TA0CTL |= MC_0;
__bic_SR_register_on_exit(LPM3_bits);
```

`LPM3_bits` is defined as `SCG1+SCG0+CPUOFF+GIE`

Conclusion - Summary

Final Results

Within this thesis we assembled and programed a wireless sensor mote. This mote is able to function individually and obtain environmental information, such as temperature, relative humidity and ambient light level. It is able to send this data to a server using the connected Xbee radio module. The mote's special features allow it to be operational as long as it is connected to a power supply. It can be part of a network of wireless sensors, providing valid information about the state of its environment. Information that might be crucial to a central server.

Future Work

A further step based on this implementation could be adding more sensors to the mote, such as human presence sensor. Also, the mote could be powered by a battery, in order to be completely independent.
Different low power modes and timers, or wireless modules could be used, to serve particular demands.
Finally, the sensors and wireless module could be united to form a board which can be connected on top of the launchpad.

Appendix

Source Code

The project's entire source code is quoted below:

Definitions and Functions

The definitions and operational functions are common for both implementations.

```
#include <msp430.h>
#include <stdio.h>


//Internal Temperature Sensor calibration values
#define CALADC12_15V_30C  *((unsigned int *)0x1A1A)   // Temperature Sensor
Calibration-30 C
#define CALADC12_15V_85C  *((unsigned int *)0x1A1C)   // Temperature Sensor
Calibration-85 C

//SHT11 Clock Pin P2_2
#define SCK_OUT  ( P2DIR |= BIT2 )
#define SCK_HIGH ( P2OUT |= BIT2 )
#define SCK_LOW  ( P2OUT &= ~(BIT2) )

//SHT11 Data Pin P7_4
#define DATA_DIR_OUT    ( P7DIR |= BIT4 )
#define DATA_DIR_IN     ( P7DIR &= ~(BIT4) )
#define DATA_OUT_HIGH   ( P7OUT |= BIT4 )
#define DATA_OUT_LOW    ( P7OUT &= ~(BIT4) )
#define DATA_IN         ( P7IN  & BIT4 )



//SHT11 command codes
#define STATUS_REG_W 0x06 //000 0011 0
#define STATUS_REG_R 0x07 //000 0011 1
#define MEASURE_TEMP 0x03 //000 0001 1
#define MEASURE_HUMI 0x05 //000 0010 1
#define RESET 0x1e //000 1111 0


/////////    SHT11 Function declarations      ///////////////////////
void sht11Reset();
void sht11_pulse();
int s_write_byte(unsigned char value);
int s_read_byte(unsigned char ack);
void s_transstart();
int s_softreset();
float measure_temp();
float measure_humi();
```

```
//////////   Internal temp sensor and Photocell function declarations   ///////////////
void internalTemp();
void lightLevel();

//////////////////  Variables    /////////////////
unsigned int temp;
volatile float InternalTemperatureDegC;
volatile float InternalTemperatureDegF;
unsigned long brightness=0.0;
volatile float SHT11TemperatureDegC=0.0;
volatile float SHT11RelativeHumidity=0.0;

char String[240];



/////////////////////////////////       Functions    ////////////////////////////////////

// Measure and return temperature from internal temperature sensor
void internalTemp(){

	ADC12CTL0 &= ~ADC12ENC;
	ADC12CTL0 = ADC12ON;         // Sampling time, ADC12 on
	ADC12CTL0 |= ADC12SHT0_8 + ADC12REFON;        // Internal ref = 1.5V

	ADC12MCTL0 = ADC12SREF_1 + ADC12INCH_10;  // ADC i/p ch A10 = temp sense i/p

	__delay_cycles(100);                        // delay to allow Ref to settle
	ADC12CTL0 |= ADC12ENC;

	ADC12CTL0 &= ~ADC12SC;
	ADC12CTL0 |= ADC12SC;                    // Sampling and conversion start
	_delay_cycles(60);

	temp = ADC12MEM0;

	// Temperature in Celsius
	InternalTemperatureDegC = (float)(((long)temp - CALADC12_15V_30C) * (85 - 30)) /
	            (CALADC12_15V_85C - CALADC12_15V_30C) + 30.0f;

	// Temperature in Fahrenheit
	InternalTemperatureDegF = InternalTemperatureDegC * 9.0f / 5.0f + 32.0f;
}


// Measure and return brightness from Photocell
void lightLevel(){
	ADC12CTL0 &= ~ADC12ENC;
	ADC12CTL0 = 0x0000;
	ADC12CTL0 = ADC12ON;             // Sampling time, ADC12 on
	ADC12CTL0 |= ADC12SHT02;  // Sampling time
	ADC12MCTL0=0x00;
	ADC12CTL0 |= ADC12ENC;

	ADC12CTL0 |= ADC12SC;            // Start sampling/conversion
	_delay_cycles(15);

	brightness=((long)ADC12MEM0/4095.0)*100;
}
```

```
// Measure and return temperature from SHT11
float measure_temp(){

	int ackn, ret_crc, ret_value;
	float temperature;
	ackn=0;
	ret_crc=0;
	ret_value=0;
	temperature=0.0;

	s_transstart();
	ackn = s_write_byte(MEASURE_TEMP);
	if(ackn==1){
		sht11Reset();
		return 0;
	}
	_delay_cycles(320000);	//wait 320ms for the sensor to complete the
measurement

	ret_value = s_read_byte(1);
	ret_value = ret_value * 256;
	ret_value += s_read_byte(1);
	ret_crc = s_read_byte(0);
	temperature = ret_value * 0.01 - 39.7;

	return temperature;
}




// Measure and return relative humidity from SHT11
float measure_humi(){

	int ackn, ret_crc, ret_value;
	float humidity;
	ackn=0;
	ret_crc=0;
	ret_value=0;
	humidity=0.0;

	s_transstart();
	ackn = s_write_byte(MEASURE_HUMI);
	if(ackn==1){
		sht11Reset();
		return 0;
	}
	_delay_cycles(80000);	//wait 80ms for the sensor to complete the measurement

	ret_value = s_read_byte(1);
	ret_value = ret_value * 256;
	ret_value += s_read_byte(1);
	ret_crc = s_read_byte(0);
	humidity = -2.0468 + 0.0367*ret_value - 0.0000015955*ret_value*ret_value;

	return humidity;
}
```

```
// clock output
void sht11_pulse(void){
    SCK_HIGH;    //CLOCK pulse -> approx 32us;
    _delay_cycles(16);
    SCK_LOW;
    _delay_cycles(16);
}

// writes a byte on the Sensibus and checks the acknowledge
int s_write_byte(unsigned char value){
	unsigned char i,error=0;

	DATA_DIR_OUT;     // set data pin as output direction

	for (i=0x80;i>0;i/=2) //shift bit for masking
	{
		if (i & value) { DATA_OUT_HIGH; }//masking value with i, write to SENSI-
BUS
		else { DATA_OUT_LOW; }
		sht11_pulse();
	}
	DATA_OUT_HIGH; //release DATA-line

	SCK_HIGH; //clk #9 for ack
	DATA_DIR_IN;     // set data pin as input direction

	if(DATA_IN){ error=1; }//check ack (DATA will be pulled down by SHT11)
	SCK_LOW;

	return error; //error=1 in case of no acknowledge
}


// reads a byte form the Sensibus and gives an acknowledge in case of "ack=1"
int s_read_byte(unsigned char ack){
	unsigned char i,val=0;

	DATA_DIR_OUT;       // set data pin as output direction
	DATA_OUT_HIGH;      // release DATA-line
	DATA_DIR_IN;     // set data pin as input direction

	for (i=0x80;i>0;i/=2) //shift bit for masking
	{
		SCK_HIGH; //clk for SENSI-BUS
		if (DATA_IN){
			val=(val | i); //read bit
		}
	  _delay_cycles(32);
		SCK_LOW;
	}
	DATA_DIR_OUT;     // set data pin as output direction
	if(ack == 1) { DATA_OUT_LOW; } //in case of "ack==1" pull down DATA-Line
	else { DATA_OUT_HIGH; }

	sht11_pulse();
	DATA_OUT_HIGH; //release DATA-line
	return val;
}
```

```c
// generates a transmission start
void s_transstart(void){
  DATA_DIR_OUT;     //OUTPUT DATA
  DATA_OUT_HIGH;
  SCK_LOW;          //Initial state
  _delay_cycles(16);
  SCK_HIGH;
  _delay_cycles(16);
  DATA_OUT_LOW;
  _delay_cycles(16);
  SCK_LOW;
  _delay_cycles(32);
  SCK_HIGH;
  _delay_cycles(16);
  DATA_OUT_HIGH;
  _delay_cycles(16);
  SCK_LOW;
  DATA_OUT_LOW;
}


// communication reset: DATA-line=1 and at least 9 SCK cycles followed by transstart
void sht11Reset(void){
  unsigned char i;
  DATA_DIR_OUT;//OUTPUT DATA
  DATA_OUT_HIGH;
  SCK_LOW;          //Initial state
  for(i=0;i<9;i++){ //9 SCK cycles
        sht11_pulse();
  }
  s_transstart(); //transmission start
}
// resets the sensor by a softreset
int s_softreset(void){
      int error=0;
      sht11Reset(); //reset communication
      error+=s_write_byte(RESET); //send RESET-command to sensor
      return error; //error=1 in case of no response form the sensor
}
```

The **`main()`** function and the *interrupt handlers* are different for the implementations.

1st Implementation: Xbee pin wake-up

```
int main(void)
{

///////////////////////// Watchdog configuration    //////////////////////////////////
      WDTCTL = WDTPW+WDTCNTCL+WDTSSEL0+WDTIS_3;


////////////////////////  Timer Interrupt (exit LPM)//////////////////////////////////
  TA0EX0 = TAIDEX_0;
  TA0CTL = TASSEL_1 + MC_0 + ID_2 + TACLR + TAIE;  // ACLK, halt timer, divider, clear
TAR, enable interrupt
                                        // approximately 6 seconds to a full cycle

///////////////////////// ADC configuration   //////////////////////////////////////
  REFCTL0 &= ~REFMSTR;                      // Reset REFMSTR to hand over control to
ADC12_A ref control registers
  ADC12CTL1 = ADC12SHP;                     // enable sample timer
  ADC12IE = 0x01;                           // Enable interrupt
  P6SEL |= 0x01;                            // P6.0 ADC option select


////////////////////////  UART and LED configuration//////////////////////////////////
  P3SEL = BIT3+BIT4;                        // P3.3,4 = USCI_A0 TXD/RXD

  P1DIR |= BIT6;                                          //Wake up pin P1.6
  P1DIR |= BIT0;                                          //Launchpad's Red LED as
output --> Xbee Transmitting
  P4DIR |= BIT7;                                          //Launchpad's Green LED as
output: Microcontroller Active mode/LPM --> LED ON/OFF
  P4OUT |= BIT7;
  P2DIR &= ~(BIT7);                                       //P2.7 as input    --> HIGH
when Xbee is ON

  UCA0CTL1 |= UCSWRST;                      // **Put state machine in reset**
  UCA0CTL1 |= UCSSEL_2;                     // SMCLK
  UCA0BR0 = 6;                              // 1MHz 9600
  UCA0BR1 = 0;                              // 1MHz 9600
  UCA0MCTL = UCBRS_0 + UCBRF_13 + UCOS16;   // Modln UCBRSx=0, UCBRFx=0, over sampling

  UCA0CTL1 &= ~UCSWRST;                     // **Initialize USCI state machine**

//////////////////////     Local variables     ////////////////////////////////////////
  int i;
  int InternalTemperatureDegCinteger,brightnessinteger;
  int SHT11TemperatureDegCinteger,SHT11RelativeHumidityinteger;


//////////////////////     SHT11 initializations     /////////////////////////////
  _delay_cycles(12000);    //wait 12ms for the SHT11 sensor to get to Sleep
  SCK_OUT;                              //set the P2.2 as output
```

```
//////////// Initialize Xbee, give it time to connect to the network ////////////////
  P1OUT |= BIT6;
  P1OUT &= ~(BIT6);
  while((P2IN & BIT7)!=BIT7);    //wait until the Xbee module turns ON
  P1OUT |= BIT6;



/////////////////////    Measuring and Transmitting Loop  ///////////////////////////
  while(1){
         WDTCTL = WDTPW+WDTCNTCL+WDTSSEL0+WDTIS_3;         //Reset Watchdog


//////////Measuring values: Temperature, relative humidity and light level ///////////
         internalTemp();

         lightLevel();

         SHT11TemperatureDegC = measure_temp();
         _delay_cycles(1000000); //wait 1s to prevent sensor overheating
         SHT11RelativeHumidity = measure_humi();
         _delay_cycles(1000000); //wait 1s to prevent sensor overheating


//////////////////////// Preparing the Data to be transmitted////////////////////////
         InternalTemperatureDegCinteger=(int)InternalTemperatureDegC;
         brightnessinteger=(int)brightness;
         SHT11TemperatureDegCinteger=(int)SHT11TemperatureDegC;
         SHT11RelativeHumidityinteger=(int)SHT11RelativeHumidity;

         for(i=0;i<240;i++){
               String[i]=' ';
         }

         sprintf(String, "\nTemperature as measured by the internal tempsensor is: %d
Celsius\n"
                                   "Temperature as measured by the SHT11 sensor is: %d
Celsius\n"
                                   "Relative humidity as measured by the SHT11 sensor
is: %d%%\n"
                                   "Brightness as measured by the Photocell is: %d%
%\n",

InternalTemperatureDegCinteger,SHT11TemperatureDegCinteger,
                                   SHT11RelativeHumidityinteger,brightnessinteger);




//////////////////////// Data Transmitting procedure////////////////////////////////
         P1OUT &= ~(BIT6);
         while((P2IN & BIT7)!=BIT7);    //wait until the Xbee module turns ON


         for(i=0;i<240;i++){
               while (!(UCA0IFG&UCTXIFG));              // USCI_A0 TX buffer ready?
               UCA0TXBUF = String[i];
               P1OUT ^= BIT0;
```

```
        }
        P1OUT &= ~(BIT0);
        _delay_cycles(50000);
        P1OUT |= BIT6;
        while((P2IN & BIT7)==BIT7);    //wait until the Xbee module gets to Sleep


/////////////// Entering Low Power mode, starting interrupt timer/////////////////////
        P4OUT &= ~(BIT7);
        TA0CTL |= MC_2;                                          //start timer
        __bis_SR_register(LPM3_bits + GIE);         //Enter LPM3
  }
}




// Timer0_A5 Interrupt Vector (TAIV) handler
#pragma vector=TIMER0_A1_VECTOR
__interrupt void TIMER0_A1_ISR(void)
{
  switch(__even_in_range(TA0IV,14))
  {
    case  0: break;                          // No interrupt
    case  2: break;                          // CCR1 not used
    case  4: break;                          // CCR2 not used
    case  6: break;                          // reserved
    case  8: break;                          // reserved
    case 10: break;                          // reserved
    case 12: break;                          // reserved
    case 14:                                        // overflow
             TA0CTL |= MC_0;                        // halt timer
             __bic_SR_register_on_exit(LPM3_bits);   // exit LPM3
             P4OUT |= BIT7;
             break;
    default: break;
  }
}
```

2nd Implementation: Xbee cyclic sleep

```
int main(void)
{

///////////////////////// Watchdog configuration    ///////////////////////////////////
      WDTCTL = WDTPW+WDTCNTCL+WDTSSEL0+WDTIS_3;

///////////////////////// Port Interrupt (Xbee ON/Sleep)   //////////////////////////
  P2IES &= ~BIT7;                               // P2.7 Lo/Hi edge
  P2IFG &= ~BIT7;                             // P2.7 IFG cleared
  P2IE |= BIT7;                               // P2.7 interrupt enabled

////////////////////////  Timer Interrupt (exit LPM)//////////////////////////////////
  TA0EX0 = TAIDEX_0;
  TA0CTL = TASSEL_1 + MC_0 + ID_2 + TACLR + TAIE;    // ACLK, halt timer, divider, clear
TAR, enable interrupt
                                          // approximately 6 seconds to a full cycle

///////////////////////// ADC configuration   ////////////////////////////////////////
  REFCTL0 &= ~REFMSTR;                        // Reset REFMSTR to hand over control to
ADC12_A ref control registers
  ADC12CTL1 = ADC12SHP;                       // enable sample timer
  ADC12IE = 0x01;                             // Enable interrupt
  P6SEL |= 0x01;                              // P6.0 ADC option select

////////////////// UART and LED configuration////////////////////////////////////////
  P3SEL = BIT3+BIT4;                          // P3.3,4 = USCI_A0 TXD/RXD

  P1DIR |= BIT0;                                            //Launchpad's Red LED as
output --> Xbee Transmitting
  P4DIR |= BIT7;                                            //Launchpad's Green LED as
output: Microcontroller Active mode/LPM --> LED ON/OFF
  P4OUT |= BIT7;
  P2DIR &= ~(BIT7);                                         //P2.7 as input     --> HIGH
when Xbee is ON

  UCA0CTL1 |= UCSWRST;                        // **Put state machine in reset**
  UCA0CTL1 |= UCSSEL_2;                       // SMCLK
  UCA0BR0 = 6;                                // 1MHz 9600
  UCA0BR1 = 0;                                // 1MHz 9600
  UCA0MCTL = UCBRS_0 + UCBRF_13 + UCOS16;     // Modln UCBRSx=0, UCBRFx=0, over sampling

  UCA0CTL1 &= ~UCSWRST;                       // **Initialize USCI state machine**

//////////////////////     Local variables     ///////////////////////////////////////
  int i;
  int InternalTemperatureDegCinteger,brightnessinteger;
  int SHT11TemperatureDegCinteger,SHT11RelativeHumidityinteger;

////////////////////      SHT11 initializations     //////////////////////////////////
  _delay_cycles(12000);    //wait 12ms for the SHT11 sensor to get to Sleep
  SCK_OUT;                                 //set the P2.2 as output
```

```
//////////////////////    Measuring Loop     ////////////////////////////////////////
  while(1){
        WDTCTL = WDTPW+WDTCNTCL+WDTSSEL0+WDTIS_3;         //Reset Watchdog

/////////   Measuring values: Temperature, relative humidity and light level  /////////
        internalTemp();

        lightLevel();

        SHT11TemperatureDegC = measure_temp();
        _delay_cycles(1000000); //wait 1s to prevent sensor overheating
        SHT11RelativeHumidity = measure_humi();
        _delay_cycles(1000000); //wait 1s to prevent sensor overheating


/////////////////   Preparing the Data to be transmitted  ////////////////////////
        InternalTemperatureDegCinteger=(int)InternalTemperatureDegC;
        brightnessinteger=(int)brightness;
        SHT11TemperatureDegCinteger=(int)SHT11TemperatureDegC;
        SHT11RelativeHumidityinteger=(int)SHT11RelativeHumidity;

        for(i=0;i<240;i++){
              String[i]=' ';
        }


        sprintf(String, "\nTemperature as measured by the internal tempsensor is: %d
Celsius\n"
                                  "Temperature as measured by the SHT11 sensor is: %d
Celsius\n"
                                  "Relative humidity as measured by the SHT11 sensor
is: %d%%\n"
                                  "Brightness as measured by the Photocell is: %d%
%\n",

InternalTemperatureDegCinteger,SHT11TemperatureDegCinteger,
                                  SHT11RelativeHumidityinteger,brightnessinteger);

/////////////      Entering Low Power mode, starting interrupt timer   /////////////
        P4OUT &= ~(BIT7);
        TA0CTL |= MC_2;                                          //start timer
        __bis_SR_register(LPM3_bits + GIE);         //Enter LPM3
  }
}


// Timer0_A5 Interrupt Vector (TAIV) handler
#pragma vector=TIMER0_A1_VECTOR
__interrupt void TIMER0_A1_ISR(void)
{
  switch(__even_in_range(TA0IV,14))
  {
    case  0: break;                          // No interrupt
    case  2: break;                          // CCR1 not used
    case  4: break;                          // CCR2 not used
    case  6: break;                          // CCR3 not used
    case  8: break;                          // CCR4 not used
    case 10: break;                          // reserved
    case 12: break;                          // reserved
    case 14:                                              // overflow
             TA0CTL |= MC_0;                                         //halt timer
             __bic_SR_register_on_exit(LPM3_bits);  //exit LPM3
             P4OUT |= BIT7;
             break;
    default: break;
  }
}

// Port 1 interrupt service routine
#pragma vector=PORT2_VECTOR
__interrupt void Port_2(void)
{
      int i;

//////////////////////// Data Transmitting procedure////////////////////////////////
      for(i=0;i<240;i++){
        while (!(UCA0IFG&UCTXIFG));    // USCI_A0 TX buffer ready?
        UCA0TXBUF = String[i];
        P1OUT ^= BIT0;                          //Toggle launchpad's Red LED while
transmitting
      }
      P1OUT &= ~(BIT0);
      while((P2IN & BIT7)==BIT7);            //wait until the Xbee module gets to
Sleep

      P2IFG &= ~BIT7;                              // P2.7 IFG cleared
}
```

References

[1] Texas Instruments: http://www.ti.com/

[2] TI E2E Community: http://e2e.ti.com/

[3] 43Oh forum: http://forum.43oh.com/

[4] Sparkfun: https://www.sparkfun.com/

[5] Energia: http://energia.nu/pin-maps/guide_msp430f5529launchpad/

[6] Custom Computer Services: http://www.ccsinfo.com/forum/

[7] Element14 Community: http://www.element14.com/community/welcome

[8] Stackoverflow: http://stackoverflow.com/

[9] http://www.education.rec.ri.cmu.edu/content/electronics/common/resistors/1.html

[10] http://en.wikipedia.org/wiki/Photoresistor

[11] http://www.technologystudent.com/elec1/ldr1.htm

[12] Sensirion SHT1x: http://www.sensirion.com/en/products/humidity-temperature/humidity-temperature-sensor-sht1x/

[13 Sensirion download center: http://www.sensirion.com/en/products/humidity-temperature/download-center/

[14] Digi XBee ZB: http://www.digi.com/products/wireless-wired-embedded-solutions/zigbee-rf-modules/zigbee-mesh-module/xbee-zb-module

[15] Digi XBee / XBee-PRO ZB (S2) Modules: http://www.digi.com/support/productdetail?pid=3430&type=utilities

[16] Digi XBee Examples and Guides: http://examples.digi.com/get-started/configuring-xbee-radios-with-x-ctu/

[17] Parallax XBee adapter board: http://www.parallax.com/product/32403