

Beginning Microcontrollers with the MSP430

Tutorial

by Gustavo Litovsky

Version 0.2, January 18, 2010

Copyright ©2008-2009 Gustavo Litovsky. All rights reserved.

This document is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

You May copy, transmit and distribute this document to anyone provided you attribute it to Gustavo Litovsky and the copyright notice remains.

This document is located at:

http://www.utdallas.edu/~gustavo.litovsky/Tutorialv0_2.pdf

Preface

I decided to write this tutorial after seeing many students struggling with the concepts of programming the MSP430 and being unable to realize their applications and projects. This was not because the MSP430 is hard to program. On the contrary, it adopts many advances in computing that has allowed us to get our application running quicker than ever. However, it is sometimes difficult for students to translate the knowledge they acquired when studying programming for more traditional platforms to embedded systems.

Programming for embedded systems (as is the case with the MSP430) is not more difficult than personal computers. In fact, it is much better in that it exposes us to the little details of how the system operates (the clocks, I/O) at a level that anyone can learn, as well as unparalleled flexibility. This, however, also forces us to make critical decisions that affect how the application runs.

The MSP430 microcontroller is an extremely versatile platform which supports many applications. With its ultra low power consumption and peripherals it enables the designing engineer to meet the goals of many projects. It has, of course, its limitations. It is geared mostly towards low energy and less intensive applications that operate with batteries, so processing capabilities and memory, among other things, are limited.

This tutorial will begin from the basics, introducing you to the theory necessary to manipulate binary digits and digital logic as used in the microcontroller. Using this you will be able to see how to manipulate the registers which control the operation of all microcontrollers. It will then cover the specifics of modules in the MSP430, from both the hardware and software perspective. I decided to follow this format primarily because you, the reader, might want to use this tutorial as a reference and just jump to a module you need help with. But I also wanted to keep the tutorial to be accessible for beginners and so the first part of the tutorial covers many of the essentials.

If you wish to begin programming immediately and understand code, you could skip to Chapter 3. Previous knowledge of the C programming language is assumed, although if you've done some Java programming, you will likely be familiar with the concepts. It is important to note that this tutorial should be supplemented by the Datasheet, User's Guide, Example Code and Application Notes for the specific MSP430 derivative used and any other component in the system. These are extremely useful in teaching you how to integrate all the ideas you have and apply them. All these documents are freely available at www.TI.com Don't Ignore them! They will answer many of your questions.

A companion file is included. This code has two components. The first is straight C code primarily based on the slaa325 application note from TI which was ported to the EZ430 and establishes very basic communications. The second part is based upon the TI Sensor Demo Code for the EZ430-RF2500.

Contents

Preface	v
1 Beginning With Microcontrollers and MSP430	1
1.1 Why Microcontrollers?	1
1.2 What Hardware do I need?	1
1.2.1 EZ430-F2013 USB Stick Development Tool	2
1.2.2 EZ430-RF2500	2
1.2.3 Experimenter Boards	3
1.2.4 FET Debugger	3
1.2.5 Custom Hardware	3
1.3 What Software do I need?	4
2 Microcontroller Basics	5
2.1 Data Types and Numeric Representation	5
2.2 Hexadecimal for MSP430	6
2.2.1 Conversion of Numbers	6
2.3 Digital Operations	7
2.4 Manipulating Module Registers	8
2.4.1 XOR Operator	12
2.5 ASCII	12
3 Beginning Programming for MSP430	15
4 MSP430 Clocks	17
4.1 Introduction	17
4.2 Clock Generation	17
4.2.1 Internal Oscillators	17
4.2.2 External Crystal	18
4.3 Clock Sources	18
4.4 Clock Signals	18
4.5 Considerations for Using Clocks	19
4.6 Basic Clock Module In EZ430-RF2500	19

5	General Purpose Input Output - GPIO	23
6	Interrupts and Low Power Modes	27
6.1	Interrupts	27
6.2	Low Power Modes	31
6.2.1	Exiting Low Power Modes	34
7	Analog to Digital Converters - ADCs	37
7.1	Introduction	37
7.2	ADC Parameters	37
7.2.1	ADC Clock	38
7.2.2	ADC Modes	38
8	Digital Interfaces	41
8.1	Serial Peripheral Interface (SPI)	42
8.1.1	Configuring SPI	43
8.1.2	Using SPI for Communications	45
9	UART Module and Communications with a PC	47
9.1	Configuring the UART	47
9.1.1	Selecting the UART Function for Pins	48
9.1.2	Sending and Receiving Information with the UART	50
9.1.3	Sending Data bigger than a byte	52
9.1.4	Sending Strings	52
9.1.5	UART Errors	52

List of Figures

1.1	EZ430-F2013 Development Stick	2
1.2	EZ430-RF2500 Development Stick	3
2.1	Converting Numbers Using the Windows Calculator	7
2.2	ADC10CTL0 Register - Source: MSP430x2xx Family User's Guide (Rev. E)	9
4.1	Selecting the DCO Frequency	20
5.1	MSP430F2274 Schematic	23
5.2	Connecting Switches to MSP430	25
6.1	Low Power Mode Savings	32
7.1	3-bit ADC Example	38
8.1	SPI Configurations	42

List of Tables

2.1	Data Types	5
2.2	Hexadecimal Number System	6
2.3	Extended Hexadecimal vs. Binary and Decimal	7
2.4	Digital Operations - OR and AND	8
2.5	Digital Operations - NOT and XOR	8
2.6	Hexadecimal representation of position	11
2.7	XOR Truth Table	12
2.8	Extended Hexadecimal vs. Binary and Decimal	13
6.1	Interrupt Sources by Priority	32
8.1	SPI Signals	42
8.2	SPI Pins in the MSP430F2274	44

Chapter 1

Beginning With Microcontrollers and MSP430

1.1 Why Microcontrollers?

To those who are unfamiliar with these devices, microcontrollers might seem extremely simple and rare as compared to personal computers. However, microcontrollers and microprocessors are embedded in many devices, with hundreds of them forming part of today's automobile, controlling everything from the engine to the sound system. Cellular phones include more sophisticated microprocessors, but these are not as different from the MSP430 that we will cover here in that the basics apply to both. The power of microcontrollers lies in their small size and adaptability. As opposed to fixed digital circuitry, microcontrollers can be programmed to perform many applications and can be later changed when improvement are required. This saves both time and money when a field upgrade is required (which you will discover to be a grand objective of any company). However, there are limitations with respect to processing power and memory (the two biggest problems you face the use of embedded processors). It is the job of the engineer to come up with the requirements of the application and select the proper device for the application. With the advances in processing capability, many more applications can be realized today with microcontrollers than ever before, especially due to their low power profile. Indeed, the flexibility of these devices will ensure their incorporation in designs many years into the future.

It is important to note that a wide array of microcontrollers exist, some rivaling or surpassing the capabilities of full fledged computers in the 70s, 80s, and maybe even 90s. UV Erasure of microcontroller and ROM are today mostly a thing of the past. With the advent of Flash memory, the microcontroller can be programmed hundred of thousands of times without any problems. Also, they incorporate a wide array of modules such Analog to Digital Converters, USB, PWM, and Wireless transceivers, enabling integration into any kind of application.

1.2 What Hardware do I need?

It is often the case students are confused about what exactly is needed for using a particular microcontroller. Indeed, many companies assume everyone will understand the basics and therefore skip this vital information. It used to be that companies provided the silicon (actual chip) and let the application engineer sort out everything else. Today, most companies attempt to provide as much information as possible to the developing engineer since most engineers want to get the application working as soon as possible and avoid wasting time.

Normally, an embedded system is composed of the following:

- An Embedded Microcontroller
- A Programming/Debugging interface for the Microcontroller

- Microcontroller Support Circuitry
- Application Specific Circuitry

The Programming/Debugging interface is the most often ignored element of the system, but it is a very important factor. Without it, how is code supposed to be put in the Microcontroller? With just Programming capabilities, we can download a firmware image into the microcontroller. However, Debugging is often a necessity since no person can write perfectly good code and ensure it can run correctly. A common debugging interface is JTAG, which is often used in Microcontrollers. The MSP430 also uses this interface, but TI adds extra functionality whose information is available only under a Non Disclosure Agreement. Therefore, I would argue that selection of both the debugger(or programmer) and even the compiler will dictate much of the effectiveness of the time spent on the design. Do not settle for inferior tools! You will pay for them later in sweat and suffering.

Today's companies offer many platforms with which to develop and learn how to use their microcontroller. Such is the case with TI. Some of their most useful development platforms are:

1.2.1 EZ430-F2013 USB Stick Development Tool

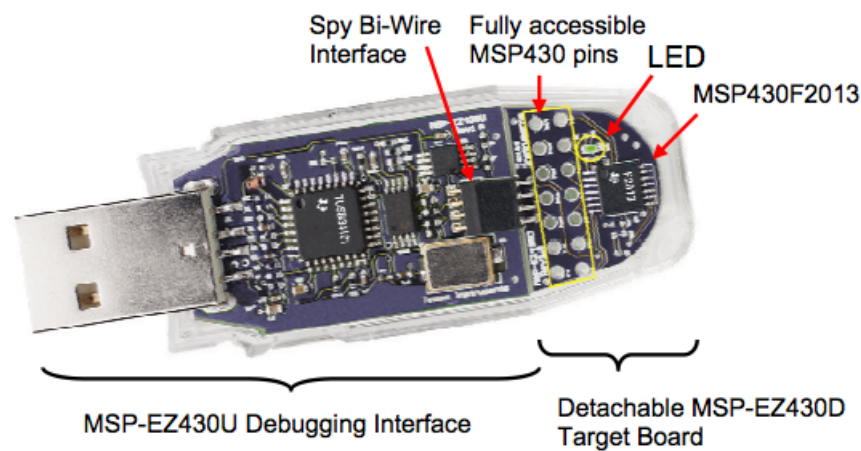


Figure 1.1: EZ430-F2013 Development Stick

This is one of the first MSP430 development tools to be in the USB stick form factor, which is gaining popularity because it is so easy to use. Since it is very low cost(\$20)and has an integrated debugger, it allows very quick development. It is composed of two boards (both located inside the plastic casing): The programming board and the target board. The target board is the board with the actual microcontroller (an MSP430F2013) and all the circuitry needed to use it. It can be detached from the programmer once the software has been downloaded to the MSP430.

The debugger board, with its USB connector, can allow programming on any computer (although there might be issues with a specific Operating System being unsupported). For more information, see the following links:

[EZ430-F2013 Home Page](#)
[MSP430F2013 Home Page](#)
[MSP430 Design Page - F2013 Contest](#)

This last website has the EZ430 contest that can provide you with real insight as to what can be done just with the EZ430.

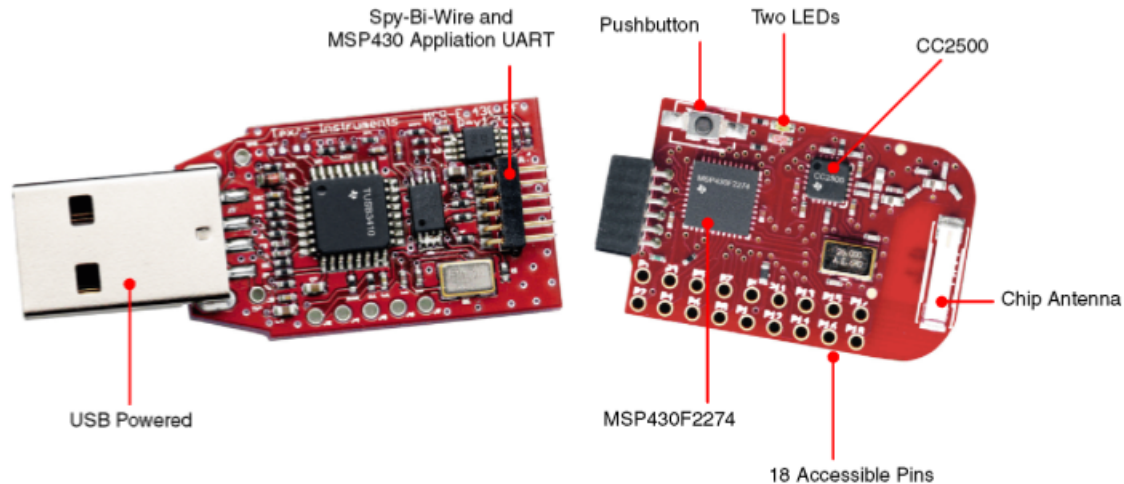


Figure 1.2: EZ430-RF2500 Development Stick

1.2.2 EZ430-RF2500

This development board is very similar to the regular EZ430 but includes a RF2500 transceiver and a MSP430F2274 microcontroller (not the F2013). This is the kit that is the subject of this tutorials because it delivers great value - microcontroller, transceiver and debugger - at a very low price

TI also supplies the sensor demo code which can be quickly programmed on both devices to enable quick temperature monitoring. One target board has to be programmed with the End Device code and the other with the Access Point code. The End device is the connected to the battery board while the target board that has the Access Point software is left connected to the programmer board and to the PC.

The sensor demo is simple. The End Device takes a reading of its Temperature Sensor (integrated into the MSP430 and accessible as an ADC channel) and Voltage (the one applied to the MSP430F2274). It sends this data to the Access Point, which also takes its own temperature and voltage readings. The AP then integrates both of these and forwards them to the PC using the UART. This data can be read using a hyperterminal equivalent program (Putty, a free program, is recommended).

[EZ430-RF2500 Home Page](#)

1.2.3 Experimenter Boards

TI also provides more comprehensive boards, called Experimenter Boards. These are much larger and don't come with a built in Debugger. For that you need a FET Debugger(which is discussed below). The two boards that are available are:

[MSP430FG4618/F2013 Experimenter Board Home Page](#)

[MSP430F5438 Experimenter Board Home Page](#)

[Putty - Terminal Emulation Software Home Page](#)

1.2.4 FET Debugger

To access the JTAG pins and debug the MSP430, TI provides a debugger, called a FET Debugger. If you have purchased a board with a 14-pin JTAG connector, such as the experimenter boards or many other boards by third parties, you will likely need this debugger.

There are two versions: one that connects a parallel port and another with USB connectivity. The USB one is the most common one. This debugger allows step by step execution of code, breakpoints, and other advanced things.

For more information:

MSP430 USB Debugging Interface Homepage

There are a few other debuggers out there, mostly sold by a company called Olimex and some other third parties.

1.2.5 Custom Hardware

Most often, once you've developed your code and tested it on a development platform, you would like to create your own PCB with the microcontroller. The microcontroller itself is only an IC. It requires several things to operate (which every development kit as mentioned above provides):

- Supply voltage consistent with Electrical Specifications (1.8V - 3.6V for most MSP430)
- Decoupling capacitors to reduce noise on the supply voltage (no power supply is perfect)
- External Crystals (in some cases where they are needed for the clock generation)
- A Programmer/Debugger or Bootstrap Loader

The list above provides the basic elements most microcontrollers need, which are besides the specific parts and connections related to the application itself (such as sensors, transceivers, passive components etc).

Users of the Microcontroller must ensure they provide the correct power supply. Although the MSP430 family requires little current and thus can be operated by batteries or other low current sources without any problems, it requires a specific voltage to be applied, and any deviation from the Maximum Recommended Specifications (available in the datasheet) can destroy the IC. Microcontrollers such as those made by Microchip and Atmel usually can tolerate 5V (and in some cases require it), but MSP430 generally accepts 1.8V to 3.6V, with possible damage to the IC when the voltage is over 3.9V or so. It is essential that if you use your custom designed circuit for the MSP430, you comply with the requirements set forth by the datasheet.

External Crystals should be used in applications that require more accuracy than is available from the microcontroller. They add more size and cost, but enable more accurate clocks. Chapter 4 provides some more information about the clocks and crystals.

Most students are familiar with computers and how to program them, using a compiler to generate the binary code and execute it. Microcontrollers, however, are different. They are programmed with code produced by a special compiler in a computer. Once the code is compiled and linked, it can be downloaded and debugged on the actual microcontroller (provided the interface to do so is available). The MSP430 devices use an interface called JTAG to perform these functions. JTAG allows a user to download and debug the code on the microcontroller, after being written with one of the several compilers that are available. This interface is accessed by using a FET Programmer that connects the computer to the microcontroller. For MSP430 devices that do not have many pins available, the programming interface available is called Spy-bi-Wire. This is proprietary to TI and roughly makes JTAG use 2 lines instead of the usual 4 or 5.

1.3 What Software do I need?

By software, we mainly refer to the compiler and IDE (Integrated Development Environment). Several of the Most popular choices are:

- IAR - A very commonly used compiler with a free version but rather expensive (some editions up to \$2000)
- Code Composer Essentials (CCE) - Developed by TI and also available with a free version (with more code).
- MSPGCC - Free Compiler with good capabilities based on GCC

There are several other compilers out there.

IAR And CCE are not free, but have a free (and code limited) version. MSPGCC is completely free and open source but has limitations such as problems with the hardware multiplier, but it has been proven to work by its use in the TinyOS operating system for embedded systems. It does not include a native IDE (graphical user interface) but can be used through Eclipse, which works nicely.

If you are just beginning to program, I recommend that you use either IAR or CCE and avoid the others until you get more familiarized as they require some porting between the compilers.

IAR Embedded Workbench Kickstart for MSP430
CCE for MSP430

Chapter 2

Microcontroller Basics

Microcontrollers are binary computers and so they operate on the basis of binary numbers. Binary numbers consist of only 1 and 0. Binary, however, is unnatural for humans to use. Assembly language is one step above binary. It is therefore most basic language for controlling computers since it represents binary directly and is easier to understand. Knowledge of assembly is not completely necessary for programming the MSP430, however it is useful in optimizing routines to get the maximum performance (in terms of speed or memory). The C programming language is the primary language used and will be followed throughout the tutorial. In general, a compiler will translate the C code into the binary code and so we do not need to worry ourselves with it.

A microcontroller is not just a CPU. It usually incorporates a range of peripherals, besides the memory and storage needed to operate. Simply put, it is a complete computer with some specialized functions. Of course, it cannot compare to a modern PC in aspects such as speed and processing capability, but it is useful in a variety of applications where a PC is simply too much. We begin by discussing the most important part of any computer: numbers and computation.

2.1 Data Types and Numeric Representation

Using binary to represent numbers is sometimes confusing. I will attempt to clear this up. This section is critical to understanding how we operate on registers and therefore the microcontroller. The first thing to understand is that the C code can accept many data types and that the compiler assigns them a specific number of bits. The table below summarizes these data types and their capacity to represent values. If this is not strictly followed, code problems, overflows and errors will likely occur. Table 2.1 shows the data types and their sizes as used in the IAR compiler:

Table 2.1: Data Types

Data Type	Bits	Decimal Range	Hex Range
Unsigned Char	8 bits	0 to 255	0x00 - 0xFF
Signed Char	8 bits	-128 to 127	0x00 - 0xFF
Unsigned Int	16 bits	0 - 65535	0x0000 - 0xFFFF
Signed Int	16 bits	-32768 to 32767	0x0000 - 0xFFFF
Unsigned Long	32 bits	-2^{31} to $2^{31}-1$	0x00000000 - 0xFFFFFFFF
Signed Long	32 bits	0 to $2^{32}-1$	0x00000000 - 0xFFFFFFFF

Therefore, when a variable is declared to be unsigned char, for example, only values from 0 to 255 can be assigned to it. If we tried to assign a higher number, overflow would occur. These data type representation are also compiler dependent. You should check your compiler's documentation. Unsigned and Signed simply determine whether numbers represented can be negative. In the case of unsigned, all numbers are assumed to be positive (0 to 255), while if the variable was declared to be a Signed char its values go from -128 to 127.

It is possible to specify whether by default a variable without the Signed or Unsigned keyword is Signed or Unsigned.

2.2 Hexadecimal for MSP430

Hexadecimal notation is essential because it is so widely used in the microcontroller as a direct form of representing binary numbers. Hexadecimal is a number system which uses 16 symbols to represent numbers. These symbols are 0 to 9 followed by A to F. Table 2.2 shows these symbols and their equivalent in the decimal and binary number system.

Table 2.2: Hexadecimal Number System

Hexadecimal	Decimal	Binary
0	0	0
1	1	1
2	2	10
3	3	11
4	4	100
5	5	101
6	6	110
7	7	111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

The decimal system only has symbols for 10 numbers(0 to 9), after which we must use two symbols to represent numbers (00 to 99). After 100 we must use 3 symbols(000 to 999), and so on. Hexadecimal numbers are usually denoted with *0x* before the actual number. This is standard notation although sometimes a *h* can indicate the same. Since Hexadecimal has 16 symbols(0x0 to 0xF), any number above 0xF must be represented by at least 2 symbols, and any number above 0xFF must be represented by at least three, etc.

It's important to realize that:

$$0x000F = 0xF$$

Leading zeroes don't change the actual value, but they do indicate the total range available. In the case above, a register which contains 0x000F is 16 bits (also known as 2 bytes, or 4 nibbles). However, the upper 3 nibbles (upper 12 bits) are all 0 either by default or because they've been changed.

Table 2.3 shows an extended list of hexadecimal numbers. Binary and Decimal are also shown for comparison. As you can see, Hexadecimal can simplify writing and prevent errors since any binary number can easily be represented with hexadecimal using less symbols. 11111111 is nothing more than 8 bits which can be represented more easily by 0xFF. In fact, going from 0x00 and 0xFF we can represent $2^8 = 256$ numbers. This is because each position in hex represent 4 bits (a nibble) and two of them represents 8 bits (a byte). This is a compact form and very convenient in programming. An MSP430 compiler such as IAR could accept binary, but this increases the complexity and the possibility of errors. Some software calculators, such as those in Windows or Linux, can easily convert between numbers in different number systems and are a big help while programming. Notice that sometimes they will not accept leading zeroes and these zeroes must be accounted for when programming.

2.2.1 Conversion of Numbers

To convert a number from one base to the other, first ensure the calculator is in **Scientific Mode**. To convert a number simply press the radio button corresponding to the base you want to to convert from, enter the number and press the

Table 2.3: Extended Hexadecimal vs. Binary and Decimal

Hexadecimal	Decimal	Binary
0	0	0
1	1	1
2	2	10
3	3	11
4	4	100
5	5	101
6	6	110
7	7	111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111
0x10	16	10000
0x11	17	10001
0x12	18	10010
0x13	19	10011
.	.	
.	.	
.	.	
0x1F	31	111111
0x20	32	100000

radio button corresponding to the base you want the result in. The number is automatically converted. The radio buttons are shown in Figure 2.1.

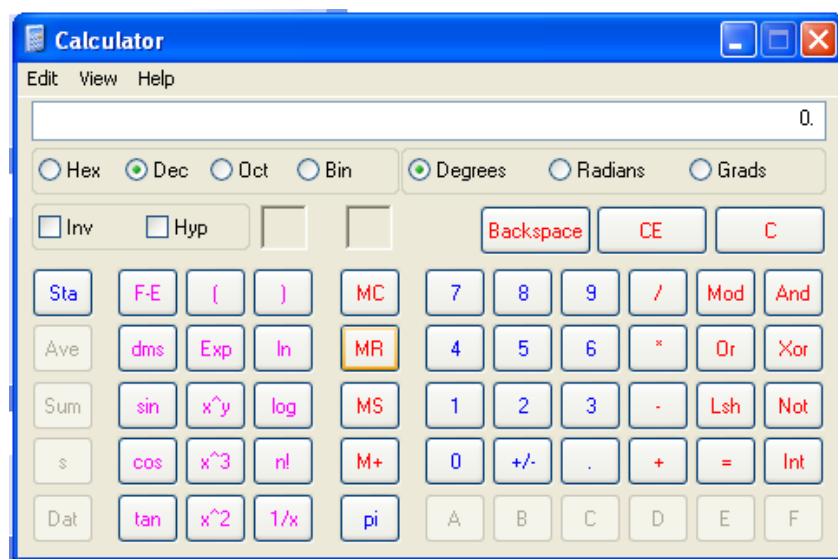


Figure 2.1: Converting Numbers Using the Windows Calculator

2.3 Digital Operations

To be able to manipulate the registers of the MSP430, some knowledge of digital logic is necessary. The most basic operations we will need to be familiar with are AND, OR, XOR, and NOT and their truth tables are shown in Tables 2.4 and 2.5.

(a) OR			(b) AND		
A	B	A B	A	B	A & B
0	0	0	0	0	0
0	1	1	0	1	0
1	0	1	1	0	0
1	1	1	1	1	1

Table 2.4: Digital Operations - OR and AND

(a) NOT	
A	\bar{A}
0	1
1	0

(b) XOR		
A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

Table 2.5: Digital Operations - NOT and XOR

2.4 Manipulating Module Registers

We previously discussed variables, which are collections of bits that can be used to represent numbers (although usually you don't deal with the individual bits and use decimal). Most programmers do not have a problem understanding their use. If we have to store the results of an operation, we can easily assign it to a variable and use it later, regardless of what number system we use. Registers are very similar to variables in that they are also a collection of bits that can be read and modified by you. As opposed to variables, however, registers are variables internal to the microcontroller and its modules. They do not reside in memory and therefore aren't allocated by the compiler.

There are two types of registers : **CPU Registers** and **Module Registers**. We will not discuss the CPU's registers that are used to store data and manipulate it. Since we are not programming in assembly, we do not need to concern ourselves with them as the compiler takes care of all of this. Rather, we are talking about Module Registers that control many of the peripherals in the microcontroller. The reason that these registers are important will become more and more apparent as you will read the tutorial.

In the MSP430 some registers are 8 bits (one byte) and some are 16 bits (2 bytes). We usually represent them visually by empty cells, with each cell capable of having a value of 1 or 0.

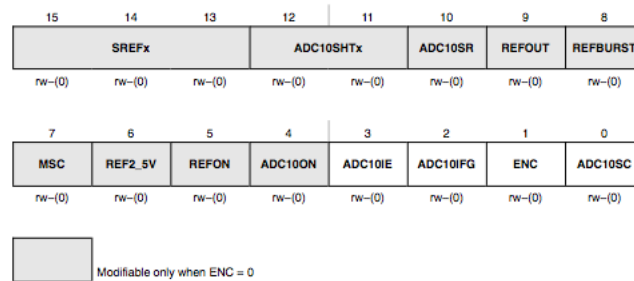
7	6	5	4	3	2	1	0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

The numbers above each cell simply represent the position, with position seven representing the **Most Significant Bit** (MSB) and position 0 representing the **Least Significant Bit** (LSB).

The register is simply a collection of bits. Each of the little boxes can have 1 or 0 in it used to control something or represent data. How do we control the above registers? We do so by using the Digital Operations we discussed in section ???. Before we discuss this, we must cover the defines used by the MSP430 code which we will be referencing. We have all these registers in the MSP430 and the information about the can be easily found in the User's Guide of each MSP430 module. At the end of each module's description will be a reference of the registers that control that module, their name, and the purpose of each bit. Figure 2.2 shows an example of a register from the User's Guide

ADC10CTL0, ADC10 Control Register 0



SREFx	Bits	Select reference	
		15-13	000 $V_{R+} = V_{CC}$ and $V_{R-} = V_{SS}$
			001 $V_{R+} = V_{REF+}$ and $V_{R-} = V_{SS}$
			010 $V_{R+} = V_{REF+}$ and $V_{R-} = V_{SS}$
			011 $V_{R+} = \text{Buffered } V_{REF+}$ and $V_{R-} = V_{SS}$
			100 $V_{R+} = V_{CC}$ and $V_{R-} = V_{REF-} / V_{REF-}$
			101 $V_{R+} = V_{REF+}$ and $V_{R-} = V_{REF-} / V_{REF-}$
			110 $V_{R+} = V_{REF+}$ and $V_{R-} = V_{REF-} / V_{REF-}$
ADC10 SHTx	Bits	ADC10 sample-and-hold time	
		12-11	00 4 x ADC10CLKs
			01 8 x ADC10CLKs
			10 16 x ADC10CLKs
			11 64 x ADC10CLKs
ADC10SR	Bit 10	ADC10 sampling rate. This bit selects the reference buffer drive capability for the maximum sampling rate. Setting ADC10SR reduces the current consumption of the reference buffer.	
		0	Reference buffer supports up to ~200 ksps
		1	Reference buffer supports up to ~50 ksps
REFOUT	Bit 9	Reference output	
		0	Reference output off
REFBURST	Bit 8	Reference burst.	
		0	Reference buffer on continuously
		1	Reference buffer on only during sample-and-conversion

Figure 2.2: ADC10CTL0 Register - Source: MSP430x2xx Family User's Guide (Rev. E)

The register in this case is 16-bit. Visually, bits are grouped together with a meaning. Therefore, Bits 15 to 13 are SREFx. Notice that below each cell there is the designation rw. This means that you can both read and write to these bits using the manipulation techniques we'll discuss in the following section. It is possible for a register to be read only. This is usually reserved for information about a particular module, indicating its status or something similar. Looking at the register description, we see a wealth of information. First of all, this register configures several aspects of the ADC such as the input channel (the physical pin), the sample and hold time, and the references.

A very important point to make is that the register is called ADC10CTL0. You might be asking yourself how can a register be named. In fact, the name is a convenient reference point. It's much easier to refer to it by ADC10CTL0 than by "The register starting at 04Ah". Remember, in programming we normally refer to variables by their names and we don't directly deal with their actual address since the compiler does that for us. We haven't declared these registers so how do we know their names? The compiler provides a header file which includes the defines for all registers in all

peripherals and shortcuts to utilizing them. This header file is included in every program we use and makes our life much easier. It is unlikely we'll ever refer to the register by its position. Rather, we simply refer to it as ADC10CTL0. Another important point to make is all those names in the bit cells (such as SREFx, ADC10SHTx, etc are shortcuts. We could (and we'll see how it's done next) add the bits individually. However, there is a much easier way. Notice that the list of references contains 16 binary numbers, from 000 to 111, we can represent these numbers with decimal numbers from 0 to 16. The header file specifically contains these definitions for SREF0 to SREF16.

We will leave aside ADC10CTL0 for now and will use another very useful register called P1OUT. The Input/Output (I/O) module contains registers to control the output of the microcontroller pins (to make them logically HIGH or LOW). bit 0 of P1OUT controls P1.0, bit 1 controls P1.1 and so forth. Suppose we want to make P1OUT equal to 0xFF, which is the same as setting all 8 bits in it to 1 and will cause all pins in Port 1 to be HIGH, then the register would look as follows:

7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1

The equivalent C code would be:

```
P1OUT = 0xFF;           // Set all the bits of P1OUT to 1
```

On the other hand we can also take the new register full of 1's and make it all zeroes again (this will make all pins in port 1 to be LOW):

```
P1OUT = 0x00;           // Set all the bits of P1OUT to 0
```

As noted before, 8 bits can represent all numbers from 0 to 255 (256 numbers total). 0xFF is equal to 255 (you can check in the calculator). Setting the register as above is just fine when we're changing all the bits, but it won't work if previously one of the bits was set to 1 or 0 and we want to leave it that way, (the whole register will be in effect erased) Perhaps another part of the program controls one of the bits in the register and if we use the assignment (=) symbol we are replacing everything. For example let's say we want make P1OUT = 0xF0, but before that it was 0x0C. It will simply set the register to 0xF0, not to 0xFC. This is something that we want to avoid in most cases; Normally you want to be very specific as to which pin you control. Masking comes into play here. You simply control one or more bits at a time without affecting the others.

Lets suppose P1OUT is all 0 except position 0 (bit 0), which contains a 1, as follows:

7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1

If we want to set bit 1 of the register to 1, this would be represented as follows in binary:

```
00000001
OR
00000010
=
00000011
```

Remember addition is bit by bit, each position with the same position so position 0 with position 0 is added, position 1 with 1, and so forth. Our previous 1 in position 0 still remains. We could, by virtue of what we learned previously, replace the binary with hex notation, which would make it easier to read and write:

```
0x01
+
```


$$\begin{array}{c} 0x02 \\ = \\ 0x03 \end{array}$$

Much easier to write obviously. We are still left with how to write this down in C so that the compiler would understand this is what we want to do. Remember, we are referring to P1OUT, which the compiler knows represents a register of 8 bits. Now in the MSP430 C programming, the addition used is the OR operand (represented as the pipe symbol, a vertical line):

```
P1OUT |= 0x02
```

This results in adding the 1 to position 1, while leaving the register as it was elsewhere. It's important to note that if the position to which 1 is added is already 1, there is no overflow (no carry). Therefore:

```
0x02 | 0x02 = 0x02
```

The OR is not exactly a summation. It is a bit by bit assignment. Actually, the OR operates on the register and simply sets that position to 1, no addition is done.

It is fairly obvious that if we wanted to set something to 0 we can't just use the OR operator since adding a zero will leave the bit unchanged. In this case the AND operand (In C it is represented by the ampersand symbol) is used to set something to 0 since

$$0xFF \text{ AND } 0x02 = 0x02$$

```
0xFF & 0x02 = 0x02
```

0xFF represents 11111111, so we're not affecting anything. Using the combination of the OR and the AND we can control any bit/bits in the register.

Diagram in table 2.6 shows the value of each position. for example, position 1 is represented by 0x02.

Table 2.6: Hexadecimal representation of position

Bit	7	6	5	4	3	2	1	0
1	0x80	0x40	0x20	0x10	0x08	0x04	0x02	0x01
0	0x7F	0xBF	0xDF	0xEF	0xF7	0xFB	0xFD	0xFE

Using the table, if we wanted to add a 1 in position 7, we just need to OR the register with 0x80. If we wanted to put a 0 in position 7, we would AND the register with the inverse of 0x80 which is 0x7F (a 0 in position 7 and 1 in the rest). These can also be added together to control multiple bits. For example, we can add a bit in position 7 and 6 by doing an OR with 0x80 + 0x40:

```
P1OUT |= 0x80 + 0x40
```

To make these operations easier, the MSP430 header includes the exact definitions for the bits up to 16 bits:

```
#define BIT0      (0x0001)
#define BIT1      (0x0002)
#define BIT2      (0x0004)
#define BIT3      (0x0008)
#define BIT4      (0x0010)
#define BIT5      (0x0020)
#define BIT6      (0x0040)
#define BIT7      (0x0080)
#define BIT8      (0x0100)
```

```
#define BIT9          (0x0200)
#define BITA          (0x0400)
#define BITB          (0x0800)
#define BITC          (0x1000)
#define BITD          (0x2000)
#define BITE          (0x4000)
#define BITF          (0x8000)
```

We can use these definitions so we don't have to memorize the hexadecimal notations. We can use them as follows:

```
P1OUT |= BIT0
```

The preprocessor (a program that runs before the compiler) goes through and replaces every instance of BIT0 with 0x0001. You don't see this happen (unless you look at some intermediate files). After the preprocessor runs, the above listing becomes:

```
P1OUT |= 0x0001
```

The above definition for each uses 4 nibbles (4 positions or 2 bytes) because we are representing 16 bits. So we can perform the following:

```
P1OUT |= BIT4    \\ Make P1.4 High(1)
```

This will turn on pin P1.4. To combine we can also do:

```
P1OUT |= BIT4 + BIT3    \\ Make P1.4 and P1.3 High(1)
```

We can put a zero in the position without explicitly inverting. The **tilde** is the equivalent of the logical NOT operation in C:

```
P1OUT &= ~BIT4    \\ Make P1.4 Low (0)
```

All this together makes register manipulation much easier. Binary can be input to the code by first putting 0b before the bits representing the number, but this is rarely used.

2.4.1 XOR Operator

Finally, there exists also the XOR operator. To better explain we can show it's truth table which we reprint to make it easier to read.

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

Table 2.7: XOR Truth Table

This table shows that if we always XOR with 1 we flip a bit from 0 to 1 or from 1 to 0. In both cases, either when A is 0 or A is 1, XORing with 1 results in the bit flipping. This is convenient with elements such as LEDs since it provides a toggle without needing to check what was the value previously.

```
P1OUT ^= BIT4    \\ Toggle P1.4
```

XORing with 0 leaves the bit as it is and is therefore not a directly useful operation.

The operations discussed above form the basic elements for bit manipulation necessary to control all the registers of the microcontroller, its peripherals and even of other devices connected using SPI, I²C or any other bus.

2.5 ASCII

Some knowledge of ASCII is useful since the UART information (communications with PC) can be interpreted as ASCII characters. ASCII is a character encoding scheme. This means that symbols such as letters and numbers are represented in a binary, hex, or decimal (which are related between themselves). Please refer to freely available ASCII tables which can show you, for example, that the number 0 is represented by 0x30. Therefore, if a 0x30 is sent to the PC using the UART module and the PC is in ASCII mode, it will show the number 0 on screen.

Table 2.8 shows The ASCII characters and it's equivalent in Hex, Binary and Decimal for a part of the ASCII table

Table 2.8: Extended Hexadecimal vs. Binary and Decimal

Glyph	Hexadecimal	Decimal	Binary
A	41	65	100 0001
B	42	66	100 0010
C	43	67	100 0011

For a more extended list, lookup ASCII in Wikipedia or another other source. ASCII differentiates between each glyph (letter, characters, etc), and even between their lowercase and uppercase equivalents. Also, ASCII defines other elements such as space or newline.

The use of the UART to send ASCII characters and strings will be covered in the chapter on UART. A simple example is when we want to send the string "Hello World" we would send each of the characters, with the space between the words also sent. At the end of the line most likely we would send the newline character.

Now that we know how to modify registers, we must understand why and when to modify them. Much of this depends on the applications, but some general guidelines can be established that will enable you to do as needed for your application.

Chapter 3

Beginning Programming for MSP430

We will now begin discussing the basics of how to start programming the MSP430. The code will be based for the EZ430-RF2500 platform and be primarily geared towards the IAR compiler. It will likely run on CCE or other compilers, but some porting might be needed. As we go through the next chapters, we'll learn to integrate more and more of the modules into our design for a variety of applications. We can't possibly cover all uses of these modules but we will cover enough so that you will be able to leverage these ideas and further learn about the modules.

A very basic piece of code that can be compiled and ran in an MSP430 microcontroller is:

Listing 3.1: MSP430 Hello World

```
#include "msp430x22x4.h"

volatile unsigned int i;           // volatile to prevent optimization

void main(void)
{
    WDTCTL = WDTPW + WDTHOLD;      // Stop watchdog timer
    P1DIR |= 0x01;                 // Set P1.0 to output direction

    for (;;)
    {
        P1OUT ^= 0x01;             // Toggle P1.0 using exclusive-OR
        i = 50000;                 // Delay
        do (i--);
        while (i != 0);
    }
}
```

This code has some elements that were previously discussed. The first line is one that many people ask about. It is relatively simple. The compiler provides a header file which includes definitions that simplify programming. We don't have to address P1DIR by its hexadecimal number (the address which points to it). Rather, we simply use P1DIR.

In this case we included msp430x22x4.h, but other header files will be needed for other MSP430 derivatives. The simplest way to see which one to use is to lookup the header file used in the example files of that MSP430. It is also possible to search the compiler directories for these files and guess. Notice the letter x in the header name can be any number or letter.

Following this, there is a variable declaration. i is declared to be unsigned int. The volatile modifier can be complex, but in general it tells the compiler not to eliminate it in its optimizations routines. If a variable is declared

and never modified in the main, but it is modified in some interrupt routine, the compiler will likely eliminate it because it doesn't know about this. Disregard this for now.

Now comes the main routine of the code. `main()` is the one thing any program will have. When code execution begins (when debugging for example), the first line to be executed is the first line in `main()`. Something that might not be obvious is what is actually set when the program begins. That is, are any clocks running before we execute the code for setting the speed?

The answer is that it must be running. The CPU cannot operate without a clock. However, what is the speed and who sets it? When the MSP430 starts, it has defaults for many of the peripherals, including clocks, I/O, and others.

One of these is the Watchdog timer, which must be disabled or else the microcontroller will restart endlessly. The first line of `main()` stops the Watchdog timer. This is a line that will be standard in all MSP430 programs, unless you will adopt something else (such as using the Watchdog timer for another purpose). A possible problem that is not apparent at first should be discussed. What happens when we require a lot of initialization time before `main()` starts and the watchdog stop line is executed? This can happen when we, for example, declare a very large array to be used in our program. The issue is that the Watchdog timer will expire and if we don't disable it quick enough, it will restart the system, causing an endless restart loop. This can happen if we have a lot of memory, greater than 2kB, and we are declaring an array with many elements.

To avoid this problem, we can declare a function for low level initialization, as follows:

```
void __low_level_init(void)
{
    WDTCTL = WDTPW+WDTHOLD;
}
```

This code will run before any variable initialization and will stop the Watchdog regardless of any initialization. However, this function is not available in all compilers. IAR and CCE do include it.

The Master Clock (MCLK), which is the source clock of the CPU, starts up at a frequency specified in the datasheet. This is done behind the scenes, but is very important. Although we won't address it yet, it is important to change the clock frequency and make sure it is adapted to our application and ensure that all clocks and crystals are running correctly.

The rest of code in Listing 3.1 is simply meant to flash an LED connected to P1.0. The first step in this is to change the direction of pin 0 in port 1 to an output. This only needs to be done once. The for loop, as it is declared, is simply an infinite loop. There is no condition in it so it continues forever. We could have easily used

```
while(1)
{
    P1OUT ^= 0x01;    // Toggle P1.0 using exclusive-OR
    i = 50000;        // Delay
    do (i--);
    while (i != 0);
}
```

The only way then to stop the loop is to use the break command. The rest of the code is simple. We XOR the value that is in position 0 of P1OUT with one, toggling it (from 1 to 0 and from 0 to 1). We then create a delay with a do... while loop. The timing of the delay is not always easy to establish (unless steps are taken to measure the number of cycles it takes). We simply have a loop that goes 50000 times and executes nothing really (except the check for the condition). Once this loop is done, we begin again with the XOR of P1OUT.

The code in Listing 3.1 is simply meant to demonstrate what could be very similar to C code in a desktop computer. Except the I/O manipulation and the Watchdog, it is quite similar.

Usually, we write the code in the following order:

1. Stop the Watchdog Timer
2. Setup the Oscillators and clocks and check for their correct operation using the appropriate handlers

3. Setup The I/O pins
4. Setup the rest of the modules to be used

The Watchdog timer will not be covered, but that topic is easily found in the User's Guide. The next chapter will add upon the elements we will need to create a fully functional application

Chapter 4

MSP430 Clocks

4.1 Introduction

Although most users of Personal Computers would not be aware of this, clocks are at the heart of any synchronous digital system. CPUs require clocks to set the time taken to run each instruction since asynchronous operation is not possible in computer processing (it presents many difficulties such as when the data to be processed comes from different places at different speeds). In PCs, the selection of clock speeds is determined by various factors. Unless you are overclocking, you will never deal with them directly. Microcontrollers, on the other hand, are usually very flexible with clocks and require that the designer specify what clocks will be used. Usually this means that both hardware and software aspects of clocks be considered during the design stage. For example, the MSP430 accepts a low frequency crystal (typically 32.768KHz), which must be added externally. Upon initializing the microcontroller, the clock system of the MSP430 must be configured to take advantage of this clock. In the EZ430-RF2500, no crystal is available and the only oscillator used is the internal oscillator. This saves space but clock precision is reduced.

Why are clocks important? One important effect is on the frequency of the CPU. The speed at which it runs depends on a clock called MCLK. This means that the speed of execution of instructions will depend on the clock. Do you have an application which needs to move data fast? The slow default speed might be too slow for your needs. On the other hand, the power consumption of the microcontroller is very dependent on the CPU speed. Although peripherals consume current, the CPU, when running, is the major offender. Current consumption usually varies linearly with clock speed and therefore one way to keep consumption to a minimum is to get the clock speed the lowest possible (or turn it off completely when not needed). MSP430 Microcontrollers generally use two categories of clocks, Fast and Slow. The fast clocks source the CPU and most modules and varies usually from several hundred kHz to Several MHz (25MHz currently in the new MSP430F5xx family). The slow clocks belong to the low kHz range.

4.2 Clock Generation

How do we generate the clocks? There are many ways to do so, but in microcontrollers you (and in the MSP430 specifically) there are generally two types of oscillators:

- Internal Oscillators
- External Crystals

4.2.1 Internal Oscillators

Internal Oscillators are usually an RC network, with an addition of various systems to try and improve the quality of the clock (making it more accurate, for example). Trimming and Calibration can significantly improve the precision

of the clock. Remember that temperature does affect the clock frequency and if this dependancy isn't reduced, it can wreak havoc on an application. Therefore, it is usually important to select The benefit of this type of oscillators is that their frequency can be easily changed and they don't occupy any more space on the PCB. On the MSP430, the internal oscillator is usually called Digitally Controller oscillator

4.2.2 External Crystal

External Crystals allow a much more precise frequency at the output , but are not tunable (they are fixed frequency sources). Instead, most microcontrollers allow dividing the frequency and this can be used to change it if necessary.

When combining these two, we get much more flexibility. Note, however, that crystals can be a bit more complicated. Selecting a crystal can be complicated because it has many parameters such as loading capacitance and tolerance. Moreover, they might require the use of external capacitors. If the capacitance issue is not addressed, the actually frequency can deviate from the nominal.

In literature 32 kHz crystal actually has a 32.768 kHz frequency. The 32 kHz crystal to be used can be of the watch crystal type. It must have (in most MSP430 derivatives) a 6pF loading capacitance and 32.768 kHz . Don't purchase the ones with 12.5pF. Mouser and Digikey have several acceptable crystals available in several packages. Cost for this crystal can be \$0.30, and will usually be less than \$1.

Refer to the following for more information about using Crystals:

User's Guide

Data Sheet

MSP430 32-kHz Crystal Oscillators Application Note - slaa322b

4.3 Clock Sources

The following are the clock sources. Not all of them exist in all MSP430 derivatives. The MSP430F2274 does not include the XT2CLK (probably due to the low number of pins in the package).

- LFXT1CLK: Low-frequency/high-frequency oscillator that can be used with low-frequency watch crystals or external clock sources of 32,768-Hz. or with standard crystals, resonators, or external clock sources in the 400-kHz to 16-MHz range.
- VLOCLK: Internal low frequency oscillator with 12-kHz nominal frequency. Its low frequency means very low power.
- DCOCLK: Internal digitally controlled oscillator (DCO).
- XT2CLK: Optional high-frequency oscillator that can be used with standard crystals, resonators, or external clock sources in the 400-kHz to 16-MHz range.

All these sources give you great flexibility. You could avoid using crystals altogether by using the internal DCO and VLO, or, if we need more precision, we use the external crystals at the expense of PCB space. It is standard practice to use LFXT1 with a 32.768 kHz crystal, leaving XT2 to be used with a high frequency crystal. However, you can do whatever you require.

4.4 Clock Signals

The clocks available are actually the last element in a long chain. That is, first we begin with the clock generator, followed by some circuitry which further helps improve clock performance and change its parameters (such as dividing the frequency by some constant). Finally, clock distribution in the MSP430 allows for different clock sources . As an example, a Timer Module in the MSP430 can be source either from the fast clock or the slow one. The CPU,

however, can only be sourced from the fast clock. The User's Guide section for each peripheral will specify what are the allowable clock sources.

Note that the flexibility of the Clocks is limited. In the MSP430F2274 we have 3 clock signals available to CPU and Modules:

- **ACLK:** Auxiliary clock. ACLK is software selectable as LFXT1CLK or VLOCLK. ACLK is divided by 1, 2, 4, or 8. ACLK is software selectable for individual peripheral modules.
- **MCLK:** Master clock. MCLK is software selectable as LFXT1CLK, VLOCLK, XT2CLK (if available on-chip), or DCOCLK. MCLK is divided by 1, 2, 4, or 8. MCLK is used by the CPU and system.
- **SMCLK:** Sub-main clock. SMCLK is software selectable as LFXT1CLK, VLOCLK, XT2CLK (if available on-chip), or DCOCLK. SMCLK is divided by 1, 2, 4, or 8. SMCLK is software selectable for individual peripheral modules.

The CPU is always sourced by MCLK. Other peripherals are sourced by either SMCLK, MCLK, and ACLK. If, for example, ACLK is set at 32.768kHz, all modules using ACLK will operate at 32.768 kHz. Therefore, it is important to consider all the modules in the system and their frequency requirements to select the actual frequency of

Why do we differentiate between Clock signals and Clock sources? This is because between the actual source (oscillator, crystal) and the actual signal there is circuitry that can perform some operations on it. The most important function of this circuitry is clock division. Clock sources such as 32 kHz source can be divided by 1, 2, 4, or 8.

4.5 Considerations for Using Clocks

Why do clocks matter and why do we not use them in their default speed? As always, there are tradeoffs. Imagine an application where the MSP430 is required to process some information quickly. Since the clock frequency determines the amount of time each CPU cycle takes, the amount of time to process information or perform a task is directly related to the clock frequency of the CPU. We therefore would like to do so at the fastest frequency possible. On the other hand, energy consumption varies almost linearly with clock speed. With these two in mind, we see that we must choose the lowest speed that we require, unless power consumption is not an issue. While this would be probably satisfactory in applications where the system has a fixed power source (AC), battery applications require saving energy whenever possible. The faster the clock, the more energy we use. The specific application will dictate what clock we use.

There is more to this, of course. Clocks can be shut off to save energy. This is especially important for the fast clocks. As we go higher in the Low Power Mode scale, more peripherals are shut off. DCO and Fast clocks are shut off first. The last clocks to be shut off are the 32kHz and VLO. This is because wakeup from low power mode might be done from a peripheral that requires a clock, such as a timer. When all clocks are off, only interrupts from unlocked peripherals, such as I/O, can wake the system up. If we need a Real Time Clock, similar to a watch clock, to wake the MCU up at some determined point, we must have the Timer or RTC used for this operating with a clock source.

4.6 Basic Clock Module In EZ430-RF2500

Since the EZ430-RF2500 uses an MSP430F2274, we must refer to MSP430F2274 documents, especially example code and User's Guide. An important feature of the MSP430F2274 is that Texas Instruments has included very useful calibration constants that reside in the Information Flash Memory (It is similar to where the program code resides but it exists to allow the user to store things like calibration constants and not get erased when we erase code and flash it).

The first consideration is to check which Clock Sources we have for the EZ430-RF2500. Inspecting the board (and checking the schematic) shows that there is in fact no crystal for the MSP430F2274. Therefore, all our sources for the Basic Clock Module are internal oscillators:

- **DCOCLK** - Internal High Speed Oscillator up to 16MHz

- VLOCLK - Very Low Frequency (12kHz) oscillator

The DCOCLK is the main clock for the system and supplies the CPU. The VLOCLK is extremely useful when our application goes to one of the sleep modes (which we discuss in 6.1). This clock can maintain timers for waking up the system at a certain point in the future. Selecting the frequency for DCOCLK is a matter of changing the settings of some registers, as shown in Figure 4.1.

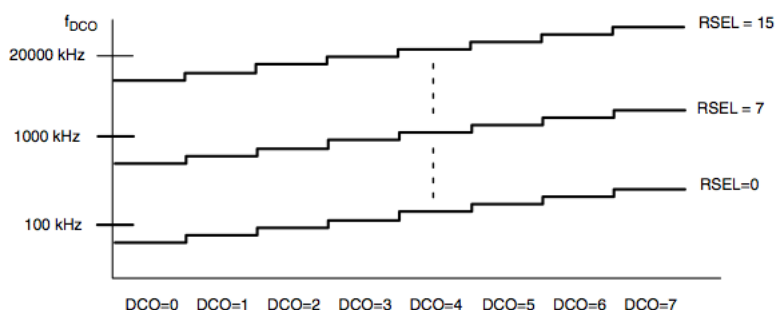


Figure 4.1: Selecting the DCO Frequency

As you can see, It is difficult to select the correct frequency precisely. To avoid issues, TI provides a set of calibrated constants that have a 1% deviation from the desired frequency.

A general function to initialize DCOCLK to 8MHz and enable the VLOCLK is provided next.

```
void configureClocks()
{
    // Set system DCO to 8MHz
    BCSCTL1 = CALBC1_8MHZ;
    DCOCTL = CALDCO_8MHZ;

    // Set LFXT1 to the VLO @ 12kHz
    BCSCTL3 |= LFXT1S_2;
}
```

You can easily take the above function and integrate it in your code(or simply use the actual lines of code). The following are the list of all calibration constant provided in the Flash:

```
CALDCO_16MHZ
CALDCO_12MHZ
CALDCO_8MHZ
CALDCO_1MHZ
```

Although these are only 4 constants (they don't include other possibly useful frequencies), they will suffice for most applications. You can still use the registers to obtain another frequency.

Listing 4.1: MSP430 Hello World with Clock control

```
#include "msp430x22x4.h"

void configureClocks();
volatile unsigned int i;           // volatile to prevent optimization

void main(void)
```

```

{
    WDTCTL = WDTPW + WDTHOLD;    // Stop watchdog timer
    configureClocks();
    P1DIR |= 0x01;                // Set P1.0 to output direction

    for (;;)
    {
        P1OUT ^= 0x01;           // Toggle P1.0 using exclusive-OR
        i = 50000;               // Delay
        do (i--);
        while (i != 0);
    }
}

void configureClocks()
{
    // Set system DCO to 8MHz
    BCSCTL1 = CALBC1_8MHZ;
    DCOCTL = CALDCO_8MHZ;

    // Set LFXT1 to the VLO @ 12kHz
    BCSCTL3 |= LFXT1S_2;
}

```

Listing 4.1 shows the Hello World application with the added clock routine. If you run this code you will immediately notice that the LED will flash much faster. The reason for this is that the default DCO clock frequency is much lower than 8MHz. It is the CPU which tells pin 1.0 to toggle, and so the speed of the CPU determines the toggle speed (although there is a small delay because of the I/O Module and other things). Also notice that whenever you declare a function, you must have both the declaration and prototype and then call it from the actual code.

8MHz will be the standard speed we will use in our EZ430-RF2500 application. This is fast enough to run most things, and is good for communicating with the CC2500. If you look at the CC2500 datasheet, you'll see that they specify the maximum clock frequency possible. The 8MHz clock will source the SPI module that communicates with the radio and although it can be divided further to reduce the clock speed to comply with the required specifications if we have a faster clock, 8MHz will work without any modifications.

Chapter 5

General Purpose Input Output - GPIO

Digital pins are the main communication element that a microcontroller has with the world. They can go HIGH (1) or LOW(0) and the actual voltage that this results in depends primarily on the system voltage (VCC). The datasheet will provide details about the voltages this represents.

Take a look at this schematic representation of the MSP430F2274:

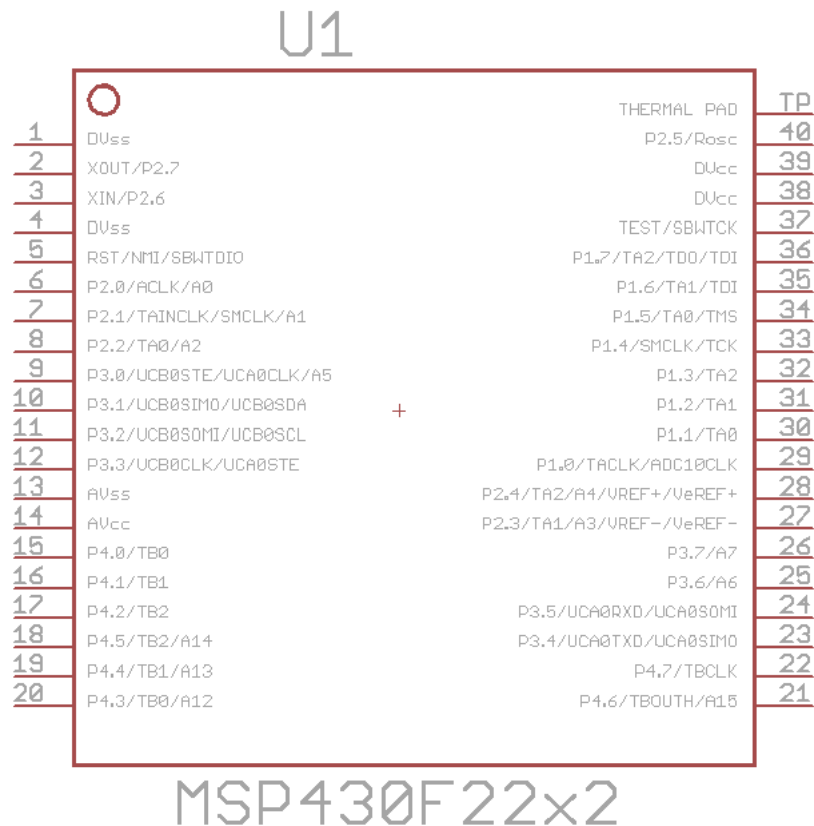


Figure 5.1: MSP430F2274 Schematic

You will notice that each pin has a long name, with several designations separated by a slash. Because microcontrollers have a limited number of pins, the manufacturer has to multiplex the functionality of the internal modules. That is, each pin can only do one thing at a time and you have to select what it is (on startup there are some defaults).

Hopefully, you can realize your design without having the multiplexing cause issues. It can become a problem because if you need more GPIO pins and you're using pin 6 to be the input to the analog to digital converter, you will have to sacrifice them. Usually, this means that in your design stage you have to take all of this into account and see which pins you'll use (and with what functionality) and what you'll sacrifice, such as access to monitor one of the internal clocks.

For the GPIO functionality of the pins, each pin numbering is written as PX.Y where the X represents the port number and the Y the pin number in that port. Different MSP430s have different number of ports. Port 2 is represented by P2.0 P2.1, P2.2, P2.3, ..., P2.7. Each port is assigned several 8 bit registers that control them and these are:

- PxSEL - This register selects what functionality is shown at the pin, GPIO or Internal Modules
- PxDIR - Used if Pin is GPIO. Selects the pin direction - Input or Output
- PxOUT - If pin is GPIO and has output direction, this selects whether it is HIGH or LOW
- PxIN - If pin is GPIO and has input direction, this represents the level at the input (HIGH or LOW)

For all these registers, the x in their name For example, port 2 has the registers P2SEL, P2DIR, P2OUT, and P2IN. P2OUT controls the output level of each pin (1 or 0) in the port because they are mapped as follows:

BIT 7	P2.7
BIT 6	P2.6
BIT 5	P2.5
BIT 4	P2.4
BIT 3	P2.3
BIT 2	P2.2
BIT 1	P2.1
BIT 0	P2.0

The 8 bits each can be 1 (HIGH) or 0 (LOW), representing VCC or 0 (or close to it). The port is controlled by P4OUT. If, for example, we wished to turn on pin 0 of the port HIGH, i.e. P4.0 we would need to set :

```
P2OUT |= 0x01;  \\ P4.0 High
```

This would set bit 0 to 1 and therefore VCC would appear on that pin. The grouping of the port makes it easy to control 8 pins. For example, if an IC were connected to all the of the pins it could be used to seamlessly transfer bytes since the input register of Port 2 (P4IN) can be read as a whole byte and no bit manipulation is needed there. It would appear obvious that if we can set the pins to HIGH or LOW, there can be no input and therefore I/O (input output) is only Input at that moment. Each pin on the microcontroller is actually multiplexed. That is, each pin can be both input or output, but it can only be one of them at any one point in time. The selection of functionality is done using the P4DIR register, where X represents the port. In P4DIR a 0 indicates an input while a 1 indicates an output. Therefore, if we wanted to set pin 0 of Port 4 to be an input we would need to do:

```
P2DIR &= ~BIT0;
```

This leaves all the bits alone while making bit 0 to be 0. There is another level of complexity. Every pin can also have a secondary function besides I/O. This is microcontroller dependant and specified in the datasheet of the specific microcontroller. The complexity of the pin is show hierarchically as follows:

First, we select between I/O and secondary function using PxSEL Then we select between Input and Output using PxDIR If we selected input, PxIN will contain the binary representation of the voltages at each pin in the port. If we selected output, P4OUT will enable setting the pins of each port to the corresponding HIGH or LOW, depending on the application.

Remember, these registers have start up defaults and they are specified in the data sheet.

For GPIO pins, there are several other important Registers. Some of these deal with interrupts and this topic will be discussed later in Chapter 6.

PxREN - This register controls the internal pullup and pulldown resistors of each pin. These can be important when using switches. Instead of using an external pullup resistor, we can simply configure everything internally, saving space. The corresponding bit in the PxOUT register selects if the pin is pulled up or pulled down.

In PxREN, a 0 in a bit means that the pullup/pulldown resistor is disabled and a 1 in any bit signifies that the pullup/pulldown resistor is enabled. Once you select any pin to have the pullup/pulldown resistor active (by putting a 1 in the corresponding bit), you must use PxOUT to select whether the pin is pulled down (a 0 in the corresponding bit) or pulled up (a 1 in the corresponding bit).

The pullup/pulldown is useful in many situations, and one of the most common one is switches connected to the MSP430.

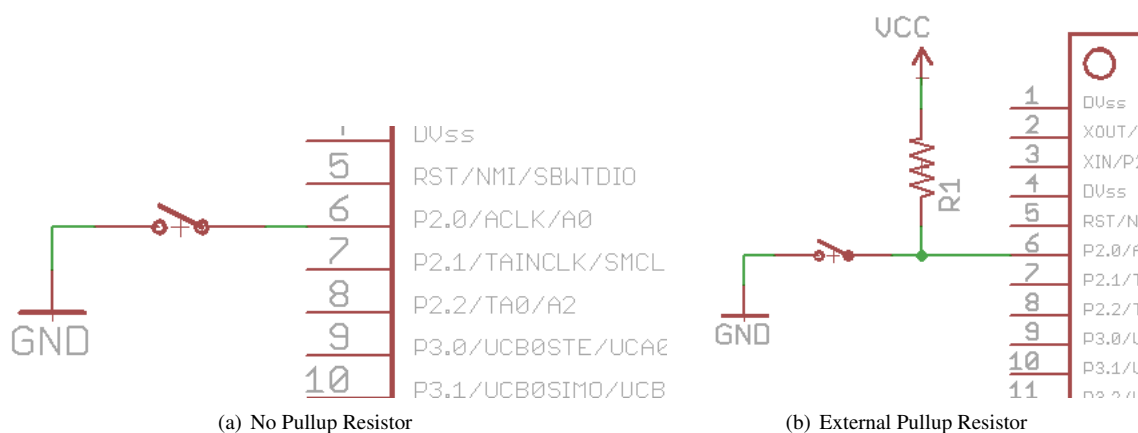


Figure 5.2: Connecting Switches to MSP430

The switch is connected to P2.0 (a GPIO with interrupt capability) in both figures. Let's think a bit about when we just connected the switch in case 5.2(a). Here, the switch is normally open and the P2.0 is left floating. Therefore, it depends on the actual microcontroller. When the switch is pressed, P2.0 is connected to ground. However, this will not trigger an interrupt because the transition is not guaranteed. We can't be sure that P2.0 was high before it went to ground.

Let's analyze the second case 5.2(b), where the designer included an external pullup resistor. Here, the switch is also open most of the time, but P2.0 is connected to VCC through the resistor. The resistor limits the current significantly (considering a resistor with 100k or so). When the user presses the switch, one end of the resistor and P2.0 are both connected to ground. P2.0 experiences a clean transition (usually and if the switch is good) from high to low and an interrupt can occur. The resistor has current flowing through it and although it is a waste, it is very small and the duration of the switch press is usually very quick (less than a second).

If you look at the schematic of the EZ430-RF2500, you will see that the switch is connected to P1.2, another GPIO.

There is no pullup resistor, so we must use the internal MSP430 pull up resistor. In fact, from my own experience when the pullup resistor isn't used, the performance of the switch is extremely bad.

Chapter 6

Interrupts and Low Power Modes

In this section I will not go into the details of the instruction set of the CPU, but rather will discuss some important features every MSP430 user should know about, the Interrupts and the Low Power Modes.

6.1 Interrupts

Interrupts represent an extremely important concept often not covered in programming since they are not readily used in a PC by most programmers. However, it is critical for anyone who programs microcontrollers to understand them and use them because of the advantages they bring.

Imagine a microcontroller waiting for a packet to be received by a transceiver connected to it. In this case, the microcontroller must constantly poll (check) the transceiver to see whether any packet was received. This means that the microcontroller will draw a lot of current since the CPU must act and use the communications module to do the polling (and the transceiver must do the same). Worse, the CPU is occupied taking care of the polling while it could be doing something more important (such as preparing to send a packet itself or processing other information). Clearly, if the CPU and the system are constantly polling the transceiver, both energy and CPU load are wasted. Interrupts represent an elegant solution to this problem. In the application above, a better solution would be to set things up so that the microcontroller would be informed (interrupted) when a packet has been received. The interrupt set in this case would cause the CPU to run a specific routing (code) to handle the received packet, such as sending it to the PC. In the meantime and while no interrupt has occurred, the CPU is left to do its business or be in low power mode and consume little current. This solution is much more efficient and, as we will later cover, allows for significant power savings. Interrupts can occur for many reasons, and are very flexible. Most microcontrollers have them and the MSP430 is no exception.

There are in general three types of interrupts:

- System Reset
- Non-maskable Interrupts (NMI)
- Maskable Interrupts

The interrupts are presented in decreasing order of system importance. The first two types are related to the microcontroller operating as it is supposed to. The last case is where all the modules allow interrupt capability and the user can take advantage of for the application. System Reset interrupts simply occur because of any of the conditions that reset the microcontroller (a reset switch, startup, etc.). These reset the microcontroller completely and are considered the most critical of interrupts. Usually you don't configure anything that they do because they simply restart the microcontroller. You have no or little control of these interrupts.

The second type of interrupts is the non maskable ones. Mask ability refers to the fact that these interrupts cannot be disabled collectively and must be disabled and enabled individually. These are interrupts in the category where the

error can possibly be handled without a reset. Just like a normal PC, a microcontroller is a machine that has to be well oiled and taken care of. The supply voltage has to be satisfied, the clocks have to be right, etc. These interrupts occur for the following reasons:

- An edge on the RST/NMI pin when configured in NMI mode
- An oscillator fault has occurred because of failure
- An access violation to the flash memory occurred

These interrupts do not usually occur but they should be handled. By handled I mean code needs to be written to do something to deal with the problem. If an oscillator has failed, a smart thing to do would be to switch to another oscillator. The User's Guide for the MSP430F2274 provides more information about these type of interrupts. The last type of interrupts is the most common one and the one you have a large control over. These are the interrupts that are produced by the different modules of the MSP430 such as the I/O, the ADC, the Timers, etc. To use the interrupt, we need the following procedure:

1. Setup the interrupt you wish and its conditions
2. Enable the interrupt/s
3. Write the interrupt handler

When the interrupt happens, the CPU stops executing anything it is currently executing. It then stores information about what it was executing before the interrupt so it can return to it when the interrupt handler is done (unless the interrupt handler changes things). The interrupt handler is then called and the code in it is executed. Whenever the interrupt ends, the system goes back to its original condition executing the original code (unless we changed something). Another possibility is that the system was in a Low Power Mode, which means the CPU was off and not executing any instructions. The procedure is similar to the one detailed above except that once the interrupt handler has finished executing the MSP430 will return to the Low Power Mode. Note that the CPU is always sourced from the DCO (or an external high speed crystal) and this source must be on for the interrupt handler to be processed. The CPU will activate to run the interrupt handler. The process of going from CPU execution (or wakeup) to interrupt handler takes some time. This is especially true whenever the system is originally in sleep mode and must wakeup the DCO to source the CPU. Normally, it is not critical but some applications might have issues with it taken longer than desired. We will now discuss a useful example: Using the switch of the EZ430-RF2500 to turn on and off the LED. This is best done by example. Two different listings are provided, the traditional way and an interrupt driver. You'll realize the immense benefits of using the interrupt solution quickly

Listing 6.1: EZ430-RF2500 Toggle LED using Switch - Polling

```
#include "msp430x22x4.h"

void configureClocks();
volatile unsigned int i;           // volatile to prevent optimization

void main(void)
{
    WDTCTL = WDTPW + WDTHOLD;      // Stop watchdog timer
    configureClocks();

    // Configure LED on P1.0
    P1DIR = BIT0;                   // P1.0 output
    P1OUT &= ~BIT0;                 // P1.0 output LOW, LED Off

    // Configure Switch on P1.2
    P1REN |= BIT2;                  // P1.2 Enable Pullup/Pulldown
```

```

        P1OUT = BIT2;                                // P1.2 pullup

        while(1)
        {
            if(P1IN & ~BIT2)                          // P1.2 is Low?
            {
                P1OUT ^= BIT0;                        // Toggle LED at P1.0
            }
        }
    }

void configureClocks()
{
    // Set system DCO to 8MHz
    BCSCTL1 = CALBC1_8MHZ;
    DCOCTL = CALDCO_8MHZ;

    // Set LFX1 to the VLO @ 12kHz
    BCSCTL3 |= LFX1S_2;
}

```

The code in listing 6.1 uses a technique called polling and is a terrible way to check if the switch has been pressed. The reason for this is apparent if you consider what is going on. The CPU is in a constant loop checking for whether the switch went low - remember that the switch is usually high until we press on it. It checks whether the pin is zero at any time. At the speed the CPU is going (8MHz), it will poll millions of times a second. There is no need to such a high rate of polling, but we're forced to do that because that's what the CPU is using. Second, let's assume that the program correctly detects that the pin has zero. It will toggle the LED just as we want. The problem is that a user's press of the switch will last many milliseconds. During this time the polling will continue rapidly and the LED will toggle again and again because the switch is still pressed, not something we want. We can add some code for delay that will prevent this, but it causes issues. How long should the delay be? This is an added problem we wish to avoid. Another big issue is that during all this time, the CPU is continuously working and it doesn't do anything else, a complete waste of a microcontroller that is wasting a lot of energy. The next listing shows an improved program. In this case, we use the interrupt flag, the indicator that an interrupt has occurred.

Listing 6.2: EZ430-RF2500 Toggle LED using Switch - Interrupt Flag

```

#include "msp430x22x4.h"

void configureClocks();
volatile unsigned int i;    // volatile to prevent optimization

void main(void)
{
    WDTCTL = WDTPW + WDTHOLD;    // Stop watchdog timer
    configureClocks();

    // Configure LED on P1.0
    P1DIR = BIT0;                // P1.0 output
    P1OUT &= ~BIT0;              // P1.0 output LOW, LED Off

    // Configure Switch on P1.2
    P1REN |= BIT2;               // P1.2 Enable Pullup/Pulldown
    P1OUT = BIT2;               // P1.2 pullup
    P1IES |= BIT2;              // P1.2 Hi/lo edge
    P1IFG &= ~BIT2;             // P1.2 IFG cleared just in case
}

```

```

while(1)
{
    if(P1IFG & BIT2)                // P1.2 IFG cleared
    {
        P1IFG &= ~BIT2;            // P1.2 IFG cleared
        P1OUT ^= BIT0;             // Toggle LED at P1.0
    }
}

void configureClocks()
{
    // Set system DCO to 8MHz
    BCSCTL1 = CALBC1_8MHZ;
    DCOCTL = CALDCO_8MHZ;

    // Set LFXT1 to the VLO @ 12kHz
    BCSCTL3 |= LFXT1S_2;
}

```

In this case we don't have an interrupt handler that is called whenever the interrupt occurs. Rather, the CPU continuously polls for the flag to see if the interrupt occurred. There are several advantages to this technique: we correctly toggle the LED only when the switch is pressed and don't have the continuous toggle issue since not until the next transition do we toggle the LED. Another benefit is that, as opposed to interrupts, the CPU continuously executes code and doesn't go to an interrupt handler. This allows us to continue processing some information and check the flag later, in which case we can act on it, rather than be forced to do so immediately. However, some of the previous disadvantages exist, mainly: The CPU still works too hard on something trivial like polling and we are still wasting energy. An important part of the code is clearing the interrupt flag for the pin. This ensures any previous flags set are cleared and we won't immediately jump into the interrupt handler when we activate the interrupts. Many programmers have made an error of not clearing the flags and then having the code run continuously. Always make sure you clear it when it is necessary.

The configuration of the LED pin is quite simple. It is the configuration of the switch that is important to us. First, we enable the pullup resistor by using P1REN and P1OUT. Afterwards, we enable the interrupt on P1.2 using P1IE (interrupt enable). Another important issue is that we can control the condition of the interrupt. P1IES controls whether the interrupt occurs on the rising edge (0 to 1) or on the falling edge (1 to 0). Only transitions cause interrupts not just static levels.

The next listing shows yet another listing that finally uses the interrupt method.

Listing 6.3: EZ430-RF2500 Toggle LED using Switch - Interrupt

```

#include "msp430x22x4.h"

void configureClocks();
volatile unsigned int i;           // volatile to prevent optimization

void main(void)
{
    WDTCTL = WDTPW + WDTHOLD;      // Stop watchdog timer
    configureClocks();

    // Configure LED on P1.0
    P1DIR = BIT0;                  // P1.0 output
    P1OUT &= ~BIT0;                // P1.0 output LOW, LED Off
}

```

```

        // Configure Switch on P1.2
        P1REN |= BIT2;           // P1.2 Enable Pullup/Pulldown
        P1OUT = BIT2;           // P1.2 pullup
        P1IE |= BIT2;           // P1.2 interrupt enabled
        P1IES |= BIT2;          // P1.2 Hi/lo falling edge
        P1IFG &= ~BIT2;         // P1.2 IFG cleared just in case

        _EINT();

        while(1)
        {
            // Execute some other code
        }
    }

void configureClocks()
{
    // Set system DCO to 8MHz
    BCSCTL1 = CALBC1_8MHZ;
    DCOCTL = CALDCO_8MHZ;

    // Set LFXT1 to the VLO @ 12kHz
    BCSCTL3 |= LFXT1S_2;
}

// Port 1 interrupt service routine
#pragma vector=PORT1_VECTOR
__interrupt void Port_1(void)
{
    P1OUT ^= BIT0;              // Toggle LED at P1.0
    P1IFG &= ~BIT2;            // P1.2 IFG cleared
}

```

Listing 6.3 shows the interrupt solution. The interrupts are activated using `eint()`; At this point you might be asking yourself: Wait, I activated the interrupt when I called the line with `P1IE`. Why do I have to enable it again? The answer lies in the name of these interrupts. These interrupts are maskable, which means that although they are controller individually, a second layer exists which can enable or disable them together. Therefore, we must disable the masking to allow our interrupts to run. The major benefit of the interrupt solution is that code continues to be executed after the interrupt is enabled. Rather, the while loop is executed. Whenever the user presses on the button, the interrupt handler is executed and then the CPU returns to its state of execution before the interrupt occurred.

An important note regarding switches is in order. Switches are not perfect. It is possible that the mechanical system inside it will toggle several times when the user presses on it. This is of course undesirable and caused by the internal mechanism vibrating and connecting/disconnecting several time. This effect is appropriately called bouncing. To avoid this, most programmers add code called a Debouncer. Simply put, once the interrupt for the switch fires, you wait a certain amount of time before you allow any other interrupt to be called. You may also wait to perform that actual action of turning on the LED, but simply avoiding multiple interrupt calls within a short period of time that no person can press on a with multiple times is the objective. I have not implemented a de-bouncer since the performance of the switch is not that bad, but a small delay of 50mS would probably suffice to prevent and bouncing problems.

Something we have not discussed yet is the case of multiple interrupts. In your application you'll likely use many interrupts and you can't predict when they occur. This means that you can't ensure by design that one interrupt will not be called while an interrupt handler is being executed. This issue is called nested interrupts and is a complex topic. Program execution can spiral out of control and unexpected things happen. Keeping interrupt routines short limits

the time they are active and so avoids any nesting. However, this might not be sufficient. A simple solution, which might or might not be acceptable depending on your application is to disable the interrupts during an interrupt handler and re-enable them after the interrupt handler is done. If the loss of an interrupt during that time is acceptable, this represents a simple solution. To do this we can use the following:

```
_DINT(); // GIE flag clear disabling general interrupts
```

This solution might not be acceptable. Improving on this technique is to create flags which, upon the return of an interrupt, are checked and acted upon. These flags can be variables that can indicate if and how many times a certain interrupt occurred. To do this simply make interrupt handlers increment the variable. This is very quick and is similar to what some operating systems use. As I mentioned, this topic is very complex and will not be further addressed.

We work next on augmenting the above software by reducing the power consumption. To do this we take advantage of the Low Power Modes provided by

6.2 Low Power Modes

Low Power Modes (LPMs) represents the ability to scale the microcontroller's power usage by shutting off parts of the MSP430. The CPU and several other modules such as clocks are not always needed. Many applications which wait until a sensor detects a certain event can benefit from turning off unused (but still running) parts of the microcontroller to save energy, turning them back on quickly when needed.

Each subsequent LPM in the MSP430 turns off more and more modules or parts of the microcontroller, the CPU, clocks and . T covers the LPM modes available in the MSP430F1611, which are similar to the ones available in most MSP430 derivatives. For the most accurate information, refer to the User's Guide and Datasheet of your particular derivative. Not mentioned here are LPM5 and some other specialized LPMs. These are available only in select MSP430 devices (in particular the newer F5xx series). However, the basic idea behind LPMs is the same , i.e. gradual shut down of the microcontroller segments to achieve power reduction.

- Active - Nothing is turned off (except maybe individual peripheral modules). No power savings
- LPM0 - CPU and MCLK are disabled while SMCLK and ACLK remain active
- LPM1 - CPU and MCLK are disabled, and DCO and DC generator are disabled if the DCO is not used for SMCLK. ACLK is active
- LPM2- CPU, MCLK, SMCLK, DCO are disabled DC generator remains enabled. ACLK is active
- LPM3 - CPU, MCLK, SMCLK, DCO are disabled, DC generator is disabled, ACLK is active
- LPM4 - CPU and all clocks disabled

It is important to note that once parts of the microcontroller are shut off, they will not operate until specifically turned on again. However, we can exit a low power mode (and turn these parts back on), and interrupts are extremely useful in this respect. Another Low Power Mode, LPM5, is available in some derivatives of the MSP430

As an example of the savings that can be achieved by incorporating Low Power Modes into the software design, I present Figure 6.1, which is shown in the MSP430F2274 User's Guide:

We usually enter a low power mode with the interrupts enabled. If we don't, we will not be able to wake up the system.

```
_bis_SR_register(LPMx_bits + GIE);
```

The x represents LPMs from 0 to 4 in the MSP430F2274. Another important thing to know is that the low power modes don't power down any modules such as the ADC or timers and these must be turned off individually. This is

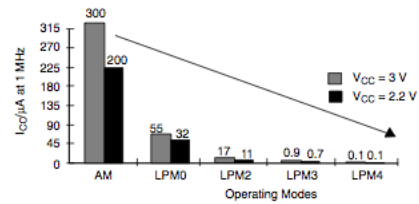


Figure 6.1: Low Power Mode Savings

because we might want to combine the low power mode with them. A simple example is going in a Low Power Mode while the ADC is sampling and then waking up from the ADC interrupt when the sample has been obtained.

Example of Interrupts and Low Power Modes: The following listing is a modification of Listing 6.3. It incorporates entering a LPM and gaining significant energy savings.

Listing 6.4: EZ430-RF2500 Toggle LED using Switch - Low Power Mode

```
#include "msp430x22x4.h"

void configureClocks();
volatile unsigned int i;           // volatile to prevent optimization

void main(void)
{
    WDTCTL = WDTPW + WDTHOLD;      // Stop watchdog timer
    configureClocks();

    // Configure LED on P1.0
    P1DIR = BIT0;                   // P1.0 output
    P1OUT &= ~BIT0;                  // P1.0 output LOW, LED Off

    // Configure Switch on P1.2
    P1REN |= BIT2;                  // P1.2 Enable Pullup/Pulldown
    P1OUT = BIT2;                   // P1.2 pullup
    P1IE |= BIT2;                   // P1.2 interrupt enabled
    P1IES |= BIT2;                  // P1.2 Hi/lo falling edge
    P1IFG &= ~BIT2;                 // P1.2 IFG cleared just in case

    // Enter Low Power Mode 4 (LPM4) w/interrupt
    __bis_SR_register(LPM4_bits + GIE);
}

void configureClocks()
{
    // Set system DCO to 8MHz
    BCSCCTL1 = CALBC1_8MHZ;
    DCOCTL = CALDCO_8MHZ;

    // Set LFXT1 to the VLO @ 12kHz
    BCSCCTL3 |= LFXT1S_2;
}

// Port 1 interrupt service routine
#pragma vector=PORT1_VECTOR
__interrupt void Port_1(void)
```

```

{
    P1OUT ^= BIT0;           // Toggle LED at P1.0
    P1IFG &= ~BIT2;         // P1.2 IFG cleared
}

```

In this case, we use the interrupt and we go into a low power mode. You can see that after entering the LPM there is not code (such as a while loop).

Why did we enter LPM4? Consider what we need during the microcontroller's time in the Low Power Mode. We don't need anything running really (in this application). There are no timers or any other peripherals that need a clock and so we can safely turn all of them off. LPM3 leaves the VLOCLK on (VLOCLK is the source for ACLK), which we don't need. LPM4 disables all clocks and the only interrupts that can activate the system from sleep are GPIO interrupts. If you go to LPM4 expecting that a Timer will activate the interrupt, it will never happen! The Timer's clock source was disabled. All you'll have is an MSP430 that's stuck in LPM forever. This is not our case so we can use LPM4 without any problems.

6.2.1 Exiting Low Power Modes

As previously mentioned, after executing the interrupt handler the system returns to the previous state. If the CPU was on and executing instructions, it will continue from where it left. If the system was called to a Low Power Mode, it will return. Suppose we don't want to return to a Low Power Mode and that perhaps we want to wakeup to do a lot of processing that wouldn't be good to put in an interrupt. To do this we must exist the low power mode in the interrupt handler. This is done with the following code:

```
__bic_SR_register_on_exit(LPMx_bits); // Clear LPMx bits from 0(SR)
```

Where x again represents the LPM that we want to exit from.

An example of this application is shown in the following listing.

Listing 6.5: EZ430-RF2500 Toggle LED using Switch - Low Power Mode

```

#include "msp430x22x4.h"

void configureClocks();
volatile unsigned int i; // volatile to prevent optimization

void main(void)
{
    WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer
    configureClocks();

    // Configure LED on P1.0
    P1DIR = BIT0;           // P1.0 output
    P1OUT &= ~BIT0;         // P1.0 output LOW, LED Off

    // Configure Switch on P1.2
    P1REN |= BIT2;          // P1.2 Enable Pullup/Pulldown
    P1OUT = BIT2;           // P1.2 pullup
    P1IE |= BIT2;           // P1.2 interrupt enabled
    P1IES |= BIT2;          // P1.2 Hi/lo falling edge
    P1IFG &= ~BIT2;         // P1.2 IFG cleared just in case

    // Enter Low Power Mode 4 (LPM4) w/interrupt
    __bis_SR_register(LPM4_bits + GIE);
}

```

```

        // Interrupt ran and we left LPM4, we end up here
        while(1)
        {
            // Execute some important code
        }
    }

void configureClocks()
{
    // Set system DCO to 8MHz
    BCSCTL1 = CALBC1_8MHZ;
    DCOCTL = CALDCO_8MHZ;

    // Set LFXT1 to the VLO @ 12kHz
    BCSCTL3 |= LFXT1S_2;
}

// Port 1 interrupt service routine
#pragma vector=PORT1_VECTOR
__interrupt void Port_1(void)
{
    P1OUT ^= BIT0;                // Toggle LED at P1.0
    P1IFG &= ~BIT2;              // P1.2 IFG cleared
    __bic_SR_register_on_exit(LPM4_bits); // Exit LPM4
}

```

The above code functions as follows: After setting up the LED and switch we go to LPM4, where energy is greatly conserved. The MSP430 waits for the interrupt. Once the switch is pressed, the LED is toggled and we exit LPM4. The next code that is executed is the while loop immediately after the LPM4 line. Note that in this case I just put a while loop to . You can put any code that you want.

Chapter 7

Analog to Digital Converters - ADCs

7.1 Introduction

The ADC is arguably one of the most important parts of a Microcontroller. Why is this? The real world is analog. To be able to access it, an ADC is required. The ADC is also probably the most difficult module to master. It is so because despite the simplicity that is assumed during courses that cover it (in which we just let the ADC sample at a particular frequency and it does so perfectly), there are many real world imperfections we must cover.

We will cover the ADC present in the MSP430F2274, the 10-bit SAR ADC. Most of the information will easily carry over to other ADCs, even if these ADCs have different features.

What is an ADC? It is a device which converts the analog waveform at the input to the equivalent representation in the digital domain. It does so by a variety of techniques which depend on the architecture of the ADC itself. SAR stands for Successive Approximation and you can understand from this what is the general idea behind the conversion. There are many architectures out there. The architecture usually dictates the capabilities of the ADC.

The ADC samples and quantizes the input waveform. Sampling means that it measures the voltage of the waveform at a particular moment. Quantization means it assigns a particular value, usually binary, to represent that voltage.

7.2 ADC Parameters

Probably the most important parameter of the ADC is its resolution, usually expressed in bits. Above we said that the MSP430F2274 has 10-bits of resolution. This is an extremely important parameter because the resolution dictates how much we can differentiate between different levels of voltage. Usually ADCs up to 12-bits are considered low to medium resolution, while those with 16-bits or higher are considered to have high resolution. This is so because the actual resolution varies exponentially with the bits as follows:

$$\text{Quantization Levels} = 2^{\text{bits}}$$

For a 10-bit ADC we have a maximum of $2^{10} = 1024$ possible levels of voltages we can represent. That's why higher bit ADCs such as 16 and even 24 are used: you can readily distinguish between smaller moves in the signal and your digital representation better approximates the original. For 16-bits we have $2^{16} = 65536$, a large increase in the number of levels we can represent. The noise in the system limits how low we can go, and there is usually a requirement for careful routing of the PCB to ensure we take full advantage of the ADC's capabilities. This is especially the fact for high resolution ADCs.

If we have a 1V on the ADC's input pin and assuming the MSP430 is operating with a reference voltage of 3V, each successive level from 0V to 3V is:

$$\frac{3V}{1024} = 0.002929687V = 3mV$$

The accuracy means that if the signal changes by less than 3mV we can't distinguish the difference, but that every 3mV change in signal level translates into an addition of 1 to the code. If we have 0V at the input (without any noise), then we have 10 zeros. However, if we have 3mV at the input then we have 9 zeros followed by a one.

$$0V \rightarrow 0000000000 = 0x00$$

$$3mV \rightarrow 0000000001 = 0x01$$

You can think about the ADC as a table. For each voltage entry we assign an equivalent binary number (usually represented by hex).

The following shows a simple 3-bit ADC. It is not realistic but simply shows how each new level is represented in binary:

3V	_____	111
2.625V	_____	110
2.25V	_____	101
1.875V	_____	100
1.5V	_____	011
1.125V	_____	010
0.75V	_____	001
0.375V	_____	000
0V	_____	

Figure 7.1: 3-bit ADC Example

The Sampling Frequency is another critical parameter of the systems. Usually the ADC manufacturer specifies this in SPS (samples per second)

The waveform at the input of the ADC might or might not stay constant. It might remain constant for long periods of time or change continuously (as with a voice at the input). When we sample the data, if it doesn't change, we can sample once. However, it is more likely that we will setup the ADC to sample regularly. For speech we would likely adopt an 8kHz

7.2.1 ADC Clock

In the case that we are using a simple resistive sensor that forms As with any digital device, the ADC itself also requires a clock, which is used for timing purposes.

The actual voltage used by the ADC as reference can vary and can be the supply voltage of the MSP430, an internal reference, or an external reference provided by the user. This results in a great flexibility

The ADC inputs of the MSP430 are usually denoted by an Ax, with the x representing the channel number. Most commonly, 8 or 16 channels are available (A0 - A15).

7.2.2 ADC Modes

The ADC with most MSP430 devices is very flexible. Although normally we think of sampling a single source (channel), the MSP430 allows us several modes which enable:

- Sample one channel once
- Sample one channel multiple times sequentially
- Sample Multiple channels (a range of channels) once
- Sample multiple channels multiple times sequentially

This can greatly simplify applications since we do not need to sample each channel individually, but can group them together. In the case of a Tri axial analog output accelerometer, for example, all axes (X,Y , and Z) can be sampled one after the other sequentially, saving time in the process.

Chapter 8

Digital Interfaces

Imagine two people try to communicate. Do you tell the other person "Hello" by spelling out "H" "E" "L" "L" "O", each letter separately? Of course not. Doing so would be slow and inefficient (and for humans difficult to understand). Similarly, when two digital devices attempt to communicate (to transmit and receive information and commands), they do so by exchanging their binary alphabet (0 and 1). However, to make communication more efficient, they group these binary numbers into groups. Moreover, there is a certain protocol used. The protocol is an agreed mechanism by which two parties communicate. It could be possible that we were to say the word hello by saying "O" "L" "L" "E" "H". We don't do that of course, but it is possible (it is done in computers where we determine whether to send the Most significant bit or Least significant bit first in the group of bits). Another reason this is done is because we can represent more things with more bits (one bit can represent two things and this is a limitation). This will become more obvious when we discuss the use of the SPI interface to access the registers of other devices (and when there are more than two registers, as is most always the case, we must be able to represent them all). The protocol allows two devices to communicate by agreeing on a set of common rules. This is the basic idea about digital interfaces.

However, in most cases, you don't get to choose any rules (you could do what's called bit banging which is simulating a protocol by simple I/O manipulation). There are a distinct set of protocols available and you will choose from what is available. For example, the CC2500 transceiver in the EZ430-RF2500 uses the SPI interfaces, which is a serial interface (bits are sent serially), is synchronous (uses a clock to indicate when bits are sent/received, and allows multiple devices on the bus. For the transceiver to successfully understand what we are telling it (isn't that the purpose of all communications), we must configure the MSP430 to the parameters which the transceiver will understand (and which the MSP430 will understand back). We will cover this in the following section.

Three of the most common digital interfaces (and the two that are available in most MSP430 devices (and in most embedded microcontrollers in general) are Serial Peripheral Interface (SPI), the Universal asynchronous receiver/transmitter (UART), and I²C.

Another popular interface is USB, which is available in almost all computers. Increasingly, USB is being added to microcontrollers because of the ubiquity of connecting to a computer using it. However, it is outside of our scope and won't be covered. There are many sources out there about it.

Another note of importance. The MSP430 (and other microcontrollers) group digital interfaces such as SPI, UART and I²C together in one module (or even in different modules). Therefore, don't be surprised if one of these communication modules can do only one type of communications at a time (although it is possible to switch). As mentioned before, microcontrollers have limited pins and therefore require multiplexing between different functionalities.

In the MSP430, the digital interface modules are called Universal Serial Communication Interface (USCI) and Serial Communication Interface (USART). Different derivatives have different combinations of these two modules. The MSP430F2274 in the EZ430-RF2500 includes two of these modules: USCLA0 and USCLB0. USCLA0 is capable of running UART, LIN, IrDA, and SPI USCLB0 is capable of running SPI and I²C.

However, TI designed the EZ430-RF2500 and already assigned the functionality of the modules. USCLA0 is allocated for UART communications with the PC, while USCLB0 is connected to the CC2500 and needs to be in SPI mode.

8.1 Serial Peripheral Interface (SPI)

SPI is a synchronus data link included in the USART module. Using the 4 wires (sometimes 3) it allows a microcontroller to communicate with any number of sensors or devices that support it. As previously mentioned, SPI works on the basis of bytes (8 bits) and not just bits.

The SPI protocol deals with the concept of master and slaves. A master is the device (like a microcontroller) which initiates the communications and provides the clock for synchronization. The slave is a device (like the CC2500) which responds to the communications, receives commands, and sends data, all as requested by the microcontroller. Note that it is possible for a microcontroller to be a slave (as long as it supports that functionality). Such can be done for two microcontrollers that are communicating with each other. One will necessarily be the master and the other the slave.

The following shows a diagram of the signals used by SPI:

Table 8.1: SPI Signals

Signal Name	Alternative Names	Description
SCLK	SCK, CLK	Serial Clock
MOSI/SIMO	SDI, DI, SI	Master Output Slave Input
MISO/SOMI	SDO, DO, SO	Slave Output Master Input
SS	nCS, CS, nSS, STE, CE	Slave Select, Chip Enable

The serial clock signal is simply a clock provided to synchronize the communications between the master and slave. SOMI and SIMO are very descriptive names. They indicate exactly how they should be connected (The Master IC's output to the slave's input). Finally, a fourth signal, which can be tied to low (Ground) at the slave, is necessary to select the slave with which we want to communicate. SPI can't communicate with all the slaves at once, so the slave select signal is used. Only the currently active slave receives the information because its slave select line is selected to low. The rest of the slaves must have their Slave select line high or else they will receive the commands. Figures 8.1 shows SPI with only 1 slave and with multiple slave, so that the use of the Slave Select (or Chip Enable) is apparent.

8.1.1 Configuring SPI

Before we can use SPI to communicate with another device, we must configure the MSP430's USART module. Depending on the MSP430, it may have 1 or 2 of such modules (or in small versions it might not include one at all). Check with the User's Guide and Datasheet to ensure that the USART module is SPI capable. In the datasheet of the MSP430F2274 we find the following pins to be the ones for SPI:

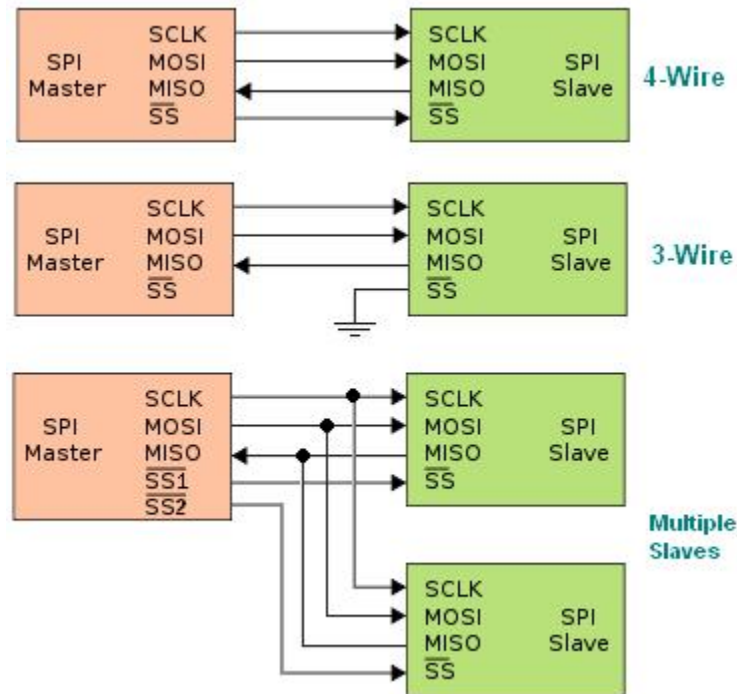


Figure 8.1: SPI Configurations

All the pins are also multiplexed and can be either a general I/O or serve the function of a peripheral module. In this microcontroller, there is an USCLA0 and USCLB0 modules, both of which are capable of SPI. To configure the SPI interface we need to make use of the register which controls it. We will assume that we are using the ez430-RF2500, which uses SPI only on USCLB0 since USCLA0 is used with the UART to communicate with the PC. The ez430-RF2500 uses a 4-wire configuration (the Chip Select pin is part of the module). This is incompatible with multiple slaves on the bus. In that case, we will need to configure the SPI to use 3-wire with P3.0 as a general purpose pin. This will be discussed later. This configuration is the most common one, as the 4-wire configuration allows for multiple masters, which we won't deal with. Pin 3.0 is used as a general I/O to control the CSn (same as slave select). Pins P3.1 and P3.2 are used as SIMO and SOMI, respectively. Pin P3.3 is the clock which allows synchronization between the master (MSP430) and the slave. We will first show the code and then explain it in detail:

Table 8.2: SPI Pins in the MSP430F2274

Pin	Peripheral Module Functionality(Multiplexed)
P3.0/ UCB0STE/ UCA0CLK/ A5	General-purpose digital I/O pin USCLB0 slave transmit enable USCLA0 clock input/output ADC10, analog input A5
P3.1/ UCB0SIMO/ UCB0SDA	General-purpose digital I/O pin USCLB0 slave in/master out in SPI mode SDA I ² C data in I ² C mode
P3.2/ UCB0SOMI/ UCB0SCL	General-purpose digital I/O pin USCLB0 slave out/master in in SPI mode SCL I ² C clock in I ² C mode
P3.3/ UCB0CLK/ UCA0STE	General-purpose digital I/O pin USCLB0 clock input/output USCLA0 slave transmit enable
P3.4/ UCA0TXD/ UCA0SIMO	General-purpose digital I/O pin USCLA0 transmit data output in UART mode slave in/master out in SPI mode
P3.5/ UCA0RXD/ UCA0SOMI	General-purpose digital I/O pin USCLA0 receive data input in UART mode slave out/master in in SPI mode

```

P3SEL |= 0x0C;           // P3.3,2 USCLB0 option select
P3DIR |= 0x01;           // P3.0 output direction
UCB0CTL0 |= UCMSB + UCMST + UCSYNC; // 3-pin, 8-bit SPI mstr, MSB 1st,
UCB0CTL1 |= UCSSEL_2;    // SMCLK as clock source
UCB0BR0 = 0x02;
UCB0BR1 = 0;
UCB0CTL1 &= ~UCSWRST;    // **Initialize USCI state machine**

```

Some of the comments explain what is done, but we will explain in more detail. The first thing done is that the pins' functionality has to be determined. P3.2 and P3.3 are selected to be the Primary peripheral module function (in this case USCLB0). This is necessary because the default functionality for these pins is general I/O. P3.1 is presumably already set for SPI. P3.0 needs to be set as an output because it is the pin that controls the Chip Select (Slave select). The code afterwards sets settings for the SPI. These are conveniently put together on one line. UCMSB selects the Most Significant Bit(MSB) first. UCMST selects this SPI module as the master. UCSYNC simply indicates that we are selecting a synchronous mode. By default, the UCB0CTL0 register is set for 3-wire communications.

As with most MSP430 peripherals, the USCI needs a clock to source it. In this case, the code selects SMCLK, which has to be previously set by the code. UCB0BR0 and UCB0BR1 are set the same for most any SPI code, so we will ignore those statements. Finally, we enable the SPI using UCSWRST.

8.1.2 Using SPI for Communications

The first thing we must do to send information over SPI is to enable the chip. In this case P3.0 is the I/O pin connected to the CSn (Chip Select) of the CC2500. For this we must put the CSn low (logic 0):

```
P3OUT &= ~0x01;           // Pull CS low to enable SPI
```

Once this is done, we can simply send the information by filling the buffer:

```
UCB0TXBUF = 0x00;         // Dummy write to start SPI
```

Finally when we are finished using SPI we must pull Chip select high to deselect the CC2500:

```
P3OUT |= 0x01;           // Pull CS high to disable SPI
```

This turning on and off of the Chip Select doesn't have to be done after every writing to the buffer. It is done when we are finished with the chip.

Let's assume we have the CC2500 RF IC using the SPI and that we've enabled CSn line by pulling it low. The following code writes to a register:

```
void TI_CC_SPIWriteReg(char addr, char value)
{
    TI_CC_CS_n_PxOUT &= ~TI_CC_CS_n_PIN;           // /CS enable by pulling CSn low
    while (TI_CC_SPI_USCIB0_PxIN & TI_CC_SPI_USCIB0_SOMI); // Wait for CCxxxx ready
    IFG2 &= ~UCB0RXIFG;                             // Clear flag
    UCB0TXBUF = addr;                                // Send address by putting in SPI buffer
    while (!(IFG2 & UCB0RXIFG));                     // Wait for TX to finish (status byte received)
    IFG2 &= ~UCB0RXIFG;                             // Clear flag
    UCB0TXBUF = value;                                // Send data
    while (!(IFG2 & UCB0RXIFG));                     // Wait for TX to finish (status byte received)
    TI_CC_CS_n_PxOUT |= TI_CC_CS_n_PIN;             // /CS disable
}
```

The function accepts two bytes as parameters. The first is the address of the register and the second is the value we want to put in the register. From the CC2500 datasheet you can see that there is a register called IOCFG2. It's address is 0x00. This means that if we wanted to change the value of this register, we would call the function as follows:

```
TI_CC_SPIWriteReg(0x00, 0x0F);
```

An analysis of the function is now discussed. As in all previous code, the first thing to do is to enable the CS line by pulling it low. Afterwards, the while loop waits until the slave is ready by checking the input pin. The CC2500 pulls SOMI low when it is ready to receive. We clear the flag of the SPI because perhaps we received a byte before and we want to know when we receive one. Again as before, we send the address by just filling the buffer with the byte of the address. Now we wait for the flag that a byte has been received. This is because when writing to registers, the status byte is sent on the SO pin each time a header byte or data byte is transmitted on the SI pin. so in fact the CC2500 sends this byte back and we just check for its presence to know that our byte has been sent. Once we know this happened, we again clear the flag so that we will know the next time it is sent. We fill the buffer, this time with the actual data to be written to the register and wait to receive the status byte again (indicating the status byte was received. After this we pull CSn (Slave or chip enable) high to turn off the SPI interface.

The code covers much of what needs to be done. It is important to read the datasheet of the IC to be communicated, as each has its own requirements. More information is available at the CC2500 datasheet from the TI website. Also code is available as part of the ez430-RF2500 demo code.

Chapter 9

UART Module and Communications with a PC

In this section we will cover the UART in the MSP430 and more specifically how we can configure the EZ430-RF2500 to communicate with a computer using this interface. The Universal Asynchronous Receiver/Transmitter module incorporated into the MSP430 is a communication peripheral that is part of the USART. The USART provides UART, SPI, and I²C capabilities. In this section we will focus on the UART functionality. Generally, It uses 2 lines: TX and RX (Transmit and Receive) which can connect to RS232 (The commonly used 9 pin Serial port on a computer). It can also connect to at USB to UART convertor such as the FT232RL and FT2232 from FTDICHP as is in our case. From the name of the peripheral you can easily guess that UART sends things serially, bit after bit. However, for convenience it groups the bits together into one byte. That is, the bits are always sent one after the other (1 or 0) but we don't select to send bits, we tell it to send a byte (or if we want a series of bytes). In the MSP430, the pins for the TX and RX are shared with the other capabilities of the USART. The byte we will send to through the UART will go to the FTDICHP IC and then to the USB. The computer will assume it is a serial port. A program such as Hyperterminal can read the bytes as ASCII. For example, if we send a 0x41 (this is one byte), the computer will show its ASCII equivalent which is the letter A. OF course the hyperterminal could show us the hexadecimal number sent (0x41) as well. This only depends on the terminal program we use on the PC. The MSP430F1611 has 2 USART modules, each with the capability to use UART independently.

Actually the USART module sends more than a byte. It sends a bit group which consists of a start bit, seven or eight data bits, an even/odd/no parity bit, an address bit (address-bit mode), and one or two stop bits. The extra stuff depend on settings.

9.1 Configuring the UART

Before we can use the UART successfully to send data, it has to be configured and the USB to UART chip and the terminal computer must have the same configuration or else communications will fail. You can thing the same as with humans, we must have a common language whose form we understand to be able to interpret what is being said. If not, things will get lost (In the UART, a 0x41 send can be misinterpreted as something else). Although common configuration information is available from TI in their example code, we will explain all the elements to give the reader a better understanding. It is important for you to read the UART section in the Family User's Guide of the MSP430F1611. It shows things in great detail and contains more information. The following are some of the main things to configure for the UART to be operating successfully.

1. The I/O pins have to be selected for the UART function
2. A clock with a certain frequency must be sourced to the UART module
3. Enable UART TX or RX or Both

4. Select the byte format as 7- or 8-bit data with odd, even, or non-parity
5. Set the baud generator correctly so as to get a correct baud rate from the clock sourced
6. Enable the module

We will address each of these extensively.

9.1.1 Selecting the UART Function for Pins

As we mentioned previously, the MSP430 (and every other microcontroller) multiplexes the various physical pins' functionality between general I/O and the internal modules. This means that we can either use a pin as a General Purpose Input/Output (HIGH or LOW), or, one of the internal modules. This can be determined by PxSEL registers, where x can represent 1,2,3 ... ports, depending on which member of the MSP430 family we are using. For the MSP430F1611 we have 6 Port and therefore we have:

P1SEL
P2SEL
P3SEL
P4SEL
P5SEL
P6SEL

Each of these 8 bytes registers controls the specific functionality assigned to each pin. We will focus on the port that has the UART functionality in the MSP430F1611. Page 7 and 8 of the MSP430F1611 tells us the job of each pin. We look for UART. We find that:

P3.4/UTXD0 32 I/O General-purpose digital I/O pin/transmit data out USART0/UART mode
P3.5/URXD0 33 I/O General-purpose digital I/O pin/receive data in USART0/UART mode
P3.6/UTXD1 34 I/O General-purpose digital I/O pin/transmit data out USART1/UART mode
P3.7/URXD1 35 I/O General-purpose digital I/O pin/receive data in USART1/UART mode

Where P3.4 indicates port 3 pin 4 and so on. The functionality is indicated by USART0/UART mode. If we would have been looking for SPI we would find other pins with USART0/SPI mode. So as you can see, it can be a General Purpose digital I/O pin or a UART pin, either TX or RX. Also shown is the fact that there are two such modules, USART0 and USART1.

Therefore, port 3 is the port we need to change.
To select UART functionality for USART0 we use the following c statements:

```
P3SEL —= 0x30; // P3.4,5 = USART0 TXD/RXD
```

0x30 is the equivalent of 110000 or 00110000 with leading zeroes.

The 1 is at 5 and 4, which represent the pins for which we want to choose the UART function.

In the User's Guide we find that:

Bit = 0: I/O Function is selected for the pin

Bit = 1: Peripheral module function is selected for the pin

This is from section 9.2.4 Function Select Registers PxSEL When the peripheral module is selected (in this case the USART and soon we will tell USART to use the UART), we don't need to worry about P3IN and P3OUT for pins 4 and 5.

The same as we did above for USART0 we can do to USART1:
P3SEL —= 0xC0; // P3.6,7 = USART1 option select

We could also combine them so both USART modules be active on the pins: `P3SEL == 0x30 + 0xC); // P3.4,5 = USART0 TXD/RXD`

2) A clock for the UART module Just like any module in the MSP430, a clock must be chosen. Remember that the UART sends bits every x amount of time. This is determined from the baud rate, which is derived from a division of the clock time. We will assume the clock has already stabilized and is working. Also, it should operate at a specific frequency (which we need to know). This is covered in a previous section of this manual.

To select the clock BRCLK we need to set the SSELx bits (bits 5 and 4) of the UTCTL1 (for USART1) or UTCTL0 (for USART0). As indicated in the datasheet, bits 5 and 4 will cause the following selection:

Binary	Hex	Define	Clock Source
00	0x00	SSEL0	UCLKI
01	0x10	SSEL1	ACLK
10	0x20	SSEL2	SMCLK
11	0x30	SSEL3	SMCLK

Notice that 01 equals 0x10 because the 1 is in bit 4. the define Column indicates an easy to use replacements defined in the header file `mcp430x14x.h` SSEL1 for example is defined to be 0x10. This is so as to avoid using number. Therefore the following do the same thing:

`UTCTL1 == SSEL1; // UCLK = SMCLK UTCTL1 == 0x10; // UCLK = SMCLK`

Also, both SSEL2 and SSEL3 give the same clock source (SMCLK). As you might remember from the discussion about system clocks, ACLK refers to the clock generated by the 32kHz crystal (can actually be other crystals too but normally 32kHz is used) while MCLK and SMCLK are usually clocks sourced from the internal Digitally Controlled Oscillator (DCO). We assume that we previously put the DCO to 1998848 or approximately 2MHz. This is so we can refer to the `fet140_uart03_09600.c` example code file. Remember that UTCTL1 can also be UTCTL0, depending on which pins we are using (P3.4 and P3.5 or P3.6 and P3.7). In the above table I am always referring to only the 5th and 4th bits, the rest remain as they are (until we change them for some reason). UCLKI is a clock which allows using an opposite polarity. We will ignore it as a possibility.

3) Enable UART TX or RX or Both To enable UART we use the ME1 register (for USART0) or ME2 (for USART1). I believe there is no ME0. The header file specifies `UTXE0 URXE0` and `UTXE1 URXE1` to enable the module.

UTXE0 enables the transmit functionality for USART0 while URXE0 does the same for the receive functionality. Therefore

`ME1 == UTXE0 + URXE0; // Enable USART0 TXD/RXD`

or for USART1

`ME2 == UTXE1 + URXE1; // Enable USART1 TXD/RXD`

4) Select the byte format as 7- or 8-bit data with odd, even, or non-parity These settings determine the type of data that is sent. When we send a byte, it can have several options. This will not be explained further than to say that we chose an 8-bit data with no parity.

This is done using the following statements: `UCTL0 == CHAR; // 8-bit character`

or

`UCTL1 == CHAR; // 8-bit character`

5) Setting the baud rate generator The baud rate we get is derived from the clock sourced to the UART module. Standard baud rates are used and include:

This is a little complicated. The user's guide specifies some simple configurations and a formula to calculate the registers. Also, a baud rate calculator is provided at: <http://mspgcc.sourceforge.net/baudrate.html> This is the most convenient as it calculates everything including errors (discussed later). We will chose 9600 baud. Although this is

Baud Rate
2400
9600
19200
115200

slow, it will send characters. Faster baud rates can be obtained although the higher the baudrate the higher the error rates. The baudrate necessary depends on the application and how fast one needs to send information. Using this calculator with 1998848 Hz and 9600 baud we obtain:

clock: 1998848Hz

desired baud rate: 9600bps

division factor: 208.2

effective baud rate: 9600.61bps

maximum error: 0.2468us 0.24%

UBR00=0xD0; UBR10=0x00; UMCTL0=0x84; /* uart0 1998848Hz 9600bps */

UBR01=0xD0; UBR11=0x00; UMCTL1=0x84; /* uart1 1998848Hz 9600bps */

The registers to be configuration can be UBR00, UBR10, and UMCTL0 or UBR01, UBR11. or UMCTL1 depending on whether USART0 or USART1 are used.

5) Finally we enable the module

```
UCTL0 &= ~SWRST;           // Initialize USART state machine
```

Putting all of this together we have the code:

```
WDTCTL = WDTPW + WDTHOLD;    // Stop WDT
P3SEL |= 0x30;               // P3.4,5 = USART0 TXD/RXD
ME1 |= UTXE0 + URXE0;        // Enabled USART0 TXD/RXD
UCTL0 |= CHAR;               // 8-bit character
UTCTL0 |= SSEL1;             // UCLK = SMCLK
UBR00 = 0xD0;                // 2MHz 9600
UBR10 = 0x00;                //
UMCTL0 = 0x00;               // no modulation
UCTL0 &= ~SWRST;             // Initialize USART state machine
```

The first line stops the watchdog timer, as previously discussed. The rest of the code was just described. The only difference is UMCTL0 = 0x00; // no modulation The authors decided no to use modulation. Information 9s be provided in the Determining the Modulation Value section of the user's guide. The mspgcc calculator more than likely calculated the modulation to reduce errors.

9.1.2 Sending and Receiving Information with the UART

Now that the UART is correctly configured and the hardware is set (including the PC software), we can send data using the UART.

Generally, two buffer registers are available for this in each UART module:

RXBUF0 and TXBUF0 for UART0 (part of USART0)

RXBUF1 and TXBUF1 for UART1 (part of USART1)

To send a byte of data we simple put it in TXBUF0 or TXBUF1:

TXBUF0 = 0x41;

or TXBUF1 = 0x41;

Which if the computer interprets as ASCII would show up as the letter A. The UART module takes the data put into

the buffer and sends it automatically without any further intervention. Sending multiple bytes isn't much harder. We simply have to ensure that the sending of the previous byte was finalized before putting new information in the buffer, and this is done by:

```
while (!(IFG1 & UTXIFG0));    // USART0 TX buffer ready?
```

This microcontroller will keep executing the while loop (do nothing) until the sending flag has been cleared. So we can put this together to send 2 bytes, one after the other:

```
TXBUF0 = 0x41;
while (!(IFG1 & UTXIFG0));    // USART0 TX buffer ready?
TXBUF0 = 0x42;
```

However, sending in this form is actually of limited use. Say for example we wanted to automate and send the alphabet from A to Z (no small caps).

This can be done as follows:

```
for(int i = 0x41; i < 0x5B; i++)
{
    TXBUF0 = i;
    while (!(IFG1 & UTXIFG0));    // USART0 TX buffer ready?
}
```

This similar approach can be used to iterate through an array. Suppose we have an array called information with the following information in it:

4	8	15	16	23	42
---	---	----	----	----	----

The above array is a collection of decimal numbers with 4 at position 0 and 42 at position 5. The IAR compiler can accept decimal number, hexadecimal numbers and binary. However, hexadecimal is unintuitive with sensor data. microcontroller doesn't care really. The array is of size 6 and to send it we can use the following code:

```
for(int i = 0; i < 6; i++)
{
    TXBUF0 = information[i];
    while (!(IFG1 & UTXIFG0));    // USART0 TX buffer ready?
}
```

The for loop accesses the array iteratively and sends the information, waits for the byte to be sent, sends the next one and so on.

Receiving data on the microcontroller is done automatically. The respective RXBUF0 or RXBUF1 buffers will contain the byte of data to be sent. This is easy if the computer sends one byte only, but is obviously not logical for many bytes. Besides, the buffer will fill with the information but we need to know when it happens. This is where an interrupt or flag come in. The USART module can generate an interrupt to immediately execute code when some data is received. For this to occur we must first set the correct interrupt to be active, using the following code:

```
IE1 |= URXIE0;    // Enable USART0 RX interrupt
```

OR

```
IE2 |= URXIE1;    // Enable USART1 RX interrupt
```

When a character is received, the interrupt vector will be called and the code defined is executed. Such an interrupt vector can be, for example, made to echo a character (send the character that was received back):

```
#pragma vector=USART0RX_VECTOR
__interrupt void usart0_rx (void)
{
    while (!(IFG1 & UTXIFG0));           // USART0 TX buffer ready?
    TXBUF0 = RXBUF0;                     // RXBUF0 to TXBUF0
}
```

where USART0RX_VECTOR can also be USART1RX_VECTOR. The interrupt does exactly what its name means, it interrupts the microcontroller by stopping the current statements it's processing, and redirects it to the interrupt code. This might or might not be done quickly enough, depending on what is done at the interrupt.

If an interrupt vector is not the right solution, a flag can also be used. A flag is a bit set by a module to indicate that something has occurred. IFG1 and IFG2 each contain a bit, URXIFG0 and URXIFG1 respectively, that indicate whether a character has been received. this can be checked by:

```
if(IFG1 & URXIFG0)
{
    \\Do Something with the recieved character
}
```

OR

```
if(IFG2 & URXIFG1)
{
    \\Do Something with the recieved character
}
```

It is important to clear the flag after it has been checked because it is not automatically.

```
IFG1 &= ~URXIFG0;           // Clear flag URXIFG0
```

OR

```
IFG2 &= ~URXIFG1;           // Clear flag URXIFG1
```

9.1.3 Sending Data bigger than a byte

In case a number from a sensor is larger than the byte allowed by the buffer, the number can be split up. Consider an integer that is 16 bits long. We can send it using two bytes (two 8 bits) as follows:

```
int number = 20000;
char part1 = (number & 0xFF00) >> 8;    \\ MSB
char part2 = number & 0x00FF;           \\ LSB
```

In the above code, a 16 bit variable is filled with the decimal number 20000. Since it's greater than 255 (which requires 8 bits to represent), it obviously needs 16 bits.

When we AND a number with 0x00FF, we are masking the upper byte since the upper 8 bits are ANDed with 1 while the lower byte(its bits) are made to be 0. This is assigned to part 1 using a right shift. The LSB (Least Significant Byte) is simply put into part2 by masking the upper byte and putting what is left (the lower byte) into part 2. Then we simply transmit part1 and part2 and reassemble the original number by shifting part1 to the left 8 positions and adding it with part2.

9.1.4 Sending Strings

The UART can be used to send strings of characters, allowing us to form complete sentences for a variety of reasons. Since terminal programs routinely support ASCII, any byte sent by the UART of the MSP430 can be interpreted as an ASCII character.

9.1.5 UART Errors

The UART module depends greatly on its source clock. Any clock error (a change in frequency relative to the baud rate divisors) sends the characters at different times and therefore changes the effective baudrate even if both the computer and the microcontroller have selected the same baud rate. This causes errors because bits arrive at a different times. For example, a bit error can make an A that was sent to be received as a B. The Digitally Controlled Oscillator is the main problem. Since some MSP430 derivatives have only a simple DCO, it drifts significantly with temperature. Possible fixes to this issue include adding a HF (High Frequency) crystal to source SCLK because the crystal is much more precise than the RC circuit used for the DCO. Other possibilities include adding an external resistor ROSC (about 100k) for the DCO, or lowering the baud rate(gives more error tolerance). ROSC (Oscillator resistor) allows reducing the frequency drift of the DCO (which drifts a lot because of temperature).

Errors are of course more pronounced at higher baud rates since the timing difference is smaller and it becomes more difficult to distinguish between the bits.