Beginning Microcontrollers

with the

MSP430

Tutorial

Gustavo Litovsky

Version 0.3

What's new in this version?

- A major update to the UART section describing more in detail how to configure the interface, including an algorithm on accurately calculating the registers. Also included are details on communicating with the Host PC such as framing.
- More information about the selection and use of crystals with the MSP430 and other micro-controllers
- Major editing improvements that better convey the information

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

List of Figures

List of Tables

Chapter 1

Introduction

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

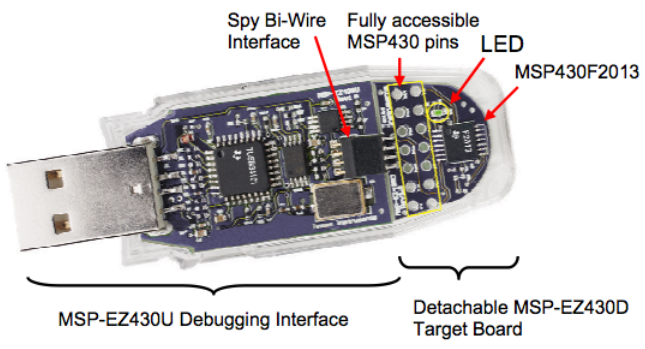

Spy Bi-Wire Interface
Fully accessible MSP430 pins
LED
MSP430F2013
MSP-EZ430U Debugging Interface
Detachable MSP-EZ430D Target Board

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.

1.2.2 EZ430-RF2500

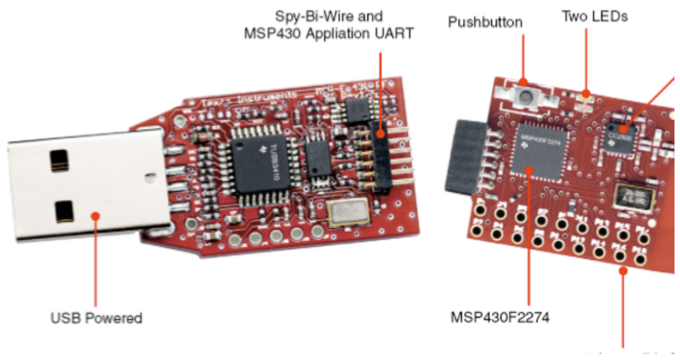


Figure 1.2: EZ430-RF2500 Development Stick

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 Programer 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 the most basic language for controlling computers since it represents binary directly and it is easier to understand. Knowledge of assembly is not completely necessary to program the MSP430, but 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 this tutorial. In general, a compiler translates the C code into the binary code and we will not worry about how this is performed initially.

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 computer with some specialized functions. Of course, it cannot compare to a modern PC in aspects such as speed and processing capability for all but the high performance CPUs, but it is useful in a wide variety of applications where larger processors are not feasible. 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 control 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 occur. Table 2.1 shows the data types and their sizes as used in the IAR compiler:

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 for what data types are available. 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 for char, for example), while if the char variable was declared to be a Signed char its values go from -128 to 127.

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

It is possible to specify whether by default a variable is Signed or Unsigned. This is done at the compiler's options menu.

When creating a variable, you must be aware of how large a number you will need to store and choose the data type appropriately.

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 *h* is used. 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 zeros 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 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. 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 zeros and these zeros 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 convert from, enter the number and press the 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.

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

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.

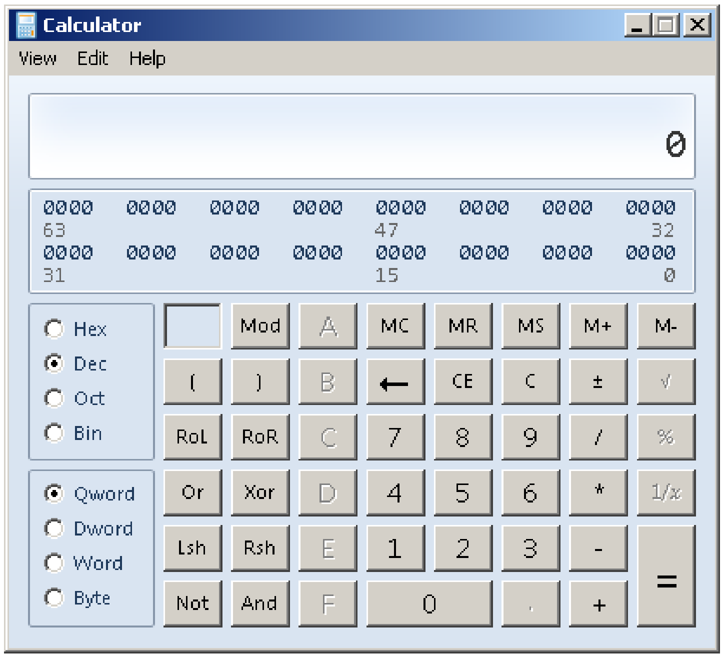
A	B	A | B
0	0	0
0	1	1
1	0	1
1	1	1

(a) OR

A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

(b) AND

Table 2.4: Digital Operations - OR and AND



Hex [✓ Dec] Oct Bin

[✓ Qword] Dword Word Byte

Figure 2.1: Converting Numbers Using the Windows Calculator

A	Ã
0	1
1	0

(a)
NOT

A	B	A ∧ B
0	0	0
0	1	1
1	0	1
1	1	0

(b) XOR

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 RAM 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

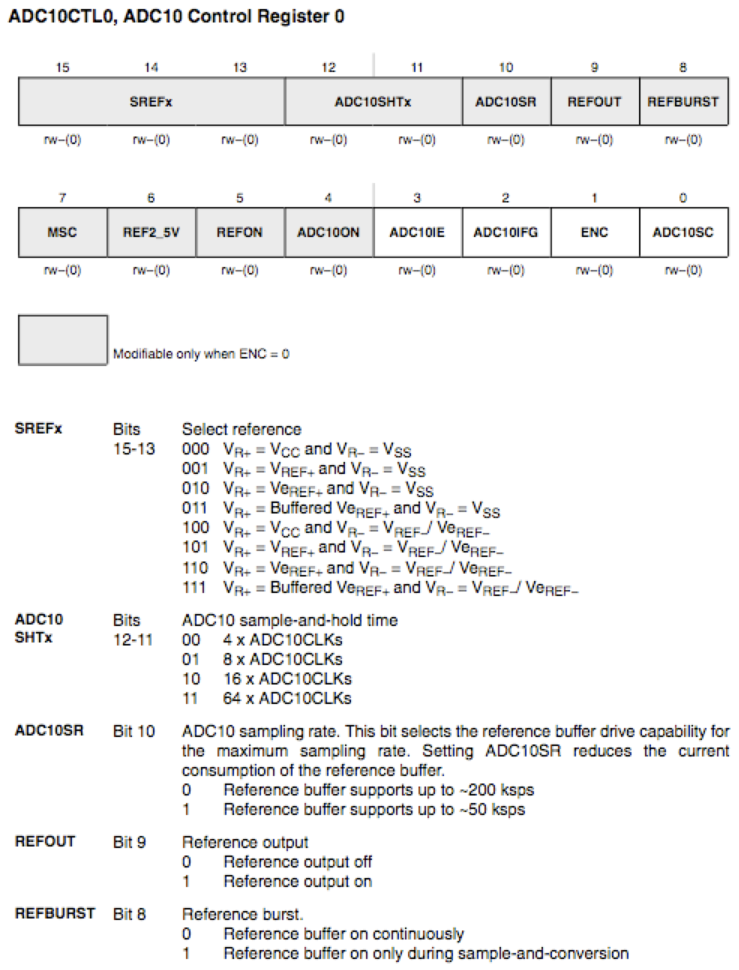
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

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 2.3. 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 them can be easily found in the User's Guide of each MSP430. 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 a User's Guide:

The register in this case is 16-bit. Visually, bits are grouped together when they share a common function. Therefore, Bits 15 to 13 are designated as 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 T̈he 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. It is good programming practice to replace odd numbers with names since humans are much more comfortable with them (have you ever visited Amazon.com by going to it's IP number? Not likely). We haven't declared these registers, so how do we know their names and how does the compiler know? 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 also defined and can be easily used to make code more readable. 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 drive them HIGH or LOW). Bit 0 of P1OUT controls P1.0, bit 1 controls P1.1 and so



ADC10CTL0, ADC10 Control Register 0

15	14	13	12	11	10	9	8
SREFx			ADC10SHTx		ADC10SR	REFOUT	REFBURST
rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)

7	6	5	4	3	2	1	0
MSC	REF2_5V	REFON	ADC10ON	ADC10IE	ADC10IFG	ENC	ADC10SC
rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)

▨ Modifiable only when ENC = 0

SREFx	Bits 15-13	Select reference 000 $V_{R+} = V_{CC}$ and $V_{R-} = V_{SS}$ 001 $V_{R+} = V_{REF+}$ and $V_{R-} = V_{SS}$ 010 $V_{R+} = Ve_{REF+}$ and $V_{R-} = V_{SS}$ 011 V_{R+} = Buffered Ve_{REF+} and $V_{R-} = V_{SS}$ 100 $V_{R+} = V_{CC}$ and $V_{R-} = V_{REF-}/ Ve_{REF-}$ 101 $V_{R+} = V_{REF+}$ and $V_{R-} = V_{REF-}/ Ve_{REF-}$ 110 $V_{R+} = Ve_{REF+}$ and $V_{R-} = V_{REF-}/ Ve_{REF-}$ 111 V_{R+} = Buffered Ve_{REF+} and $V_{R-} = V_{REF-}/ Ve_{REF-}$
ADC10 SHTx	Bits 12-11	ADC10 sample-and-hold time 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 1 Reference output on
REFBURST	Bit 8	Reference burst. 0 Reference buffer on continuously 1 Reference buffer on only during sample-and-conversion

Figure 2.2: ADC10 Register Example

forth. Suppose we want to make P1OUT equal to 0xFF, which would set all 8 bits in it to 1 and will cause all pins in Port 1 to go 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 zeros again (this will make all pins in port 1 to become LOW):

```
P1OUT = 0x00;          // Set all the bits of P1OUT to 1
```

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 ërased̈) Perhaps another part of the program controls one of the bits in the register and if we use the assignment (=) symbol we will affect it adversely.

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 that OR is performed bit by bit, each position with the same position, so that position 0 is added to the other position 0, position 1 with 1, and so forth. Our previous 1 in position 0 still remains unaffected.

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
OR
0x02
=
0x03

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 (because we're performing an OR). Therefore:

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

The OR is not a summation. It is a bit by bit assignment as per its truth table.

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 AND 0x02 = 0x02
```

or

```
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, a '1' in 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 and 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. By the way, Binary can be input to the code by first putting 0b before the bits representing the number:

```
P1OUT |= 0b11111111;
```

2.4.1 XOR Operator

Finally, there exists also the XOR operator whose truth table is show below:

A	B	$A \wedge B$
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 f̈lipä 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 LEDs since it provides a toggle without needing to check what was the previous value.

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

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

The operations discussed above form the basic elements for bit manipulation necessary to control all the registers of the microcontroller and its peripherals, even allowing communication with other devices connected using SPI,I^{2}C 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 some ASCII characters and their equivalents in Hex, Binary and Decimal:

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, you can look up 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 Ḧello Worldẅe 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.

2.6 Conclusion

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 on your own.

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

Listing 3.1: MSP430 Hello World

```c
#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 and it is very simple. The first line includes the header file with 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 given a particular derivative. Notice the letter x in the header name can be one or multiple numbers or a letter (commonly F for Flash).

Following this, there is a variable declaration. i is declared to be unsigned int. The volatile modifier tells the compiler not to eliminate it in its optimizations routines. If a variable is declared and never modified in main() the compiler will likely eliminate it because it doesn't know about an interrupt routine accesing it.

Now comes the main routine of the code. main() is common to among all programs. 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. Check the Datasheet and user's guide for information about the defaults. 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.

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? 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 before initalization has ended and this will restart the system, in an endless loop. This can happen if we have a lot of memory, greater that 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 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. We then include an infinite loop. There is no condition for breaking 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 (except the check for the condition). Once this loop is done, we begin again with the XOR of P1OUT.

Usually, when we write an MSP430 application we do as follows:

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
5. start the application specific code

Information about the Watchdog timer is included 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 run the CPU since asynchronous operation is not possible in computer proccesing (it presents many difficulties such as when the data to be processed from 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 and at what speeds. Usually this means that both hardware and software aspects of clocks be considered during the design stage.

As an 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. Because of this, the speed of instruction execution 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 do consume current, the CPU, when running, is in most cases the major offender. Current consumption usually varies linearly with clock speed and therefore one way to keep consumption to a minimum is to set the clock speed as low as 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 and are usually either 12kHz or 32.768kHz.

How do we generate the clocks? There are many ways to do so, but in the MSP430 (and many other microcontrollers) there are generally three types of clock sources:

- Internal Oscillators

- External Crystals
- External Oscillators

4.2 Internal Oscillators

Internal Oscillators are usually an RC network with circuitry to try and improve the accuracy of the clock. Trimming and Calibration by the manufacturer can significantly improve the precision of the clock. Remember, however, that temperature does affect the clock frequency and if this dependancy isn't reduced by some sort of calibration, 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, A fast Digitally Controller Oscillator (DCO) oscillator is avaialble, with another slow 12kHz oscillator optional and usually available in newer MSP430 variants.

4.3 External Crystals

External Crystals add a large measure of accuracy to oscillators and should be used as much as possible unless cost and area considerations are more important. They are not tunable but with PLL or other circuitry can generate other frequencies based on the crystal.

The selection and use of crystals is not always so straightforward because they are specified using several parameters. Aside from the frequency,accuracy, drift and aging factors, the most overlooked part of adding a crystal to the circuit is the loading capacitance. A detailed explanation for the need and purpose of the loading capacitance is available from most crystal manufacturers.

In general, every crystal used in a parallel configuration, which is the case in the MSP430 and many other microcontrollers, requires capacitors to provide the capacitance that is necessary for the oscillator to oscillate at the right frequency. These capacitors are usually connected to the two crystal pins as follows:

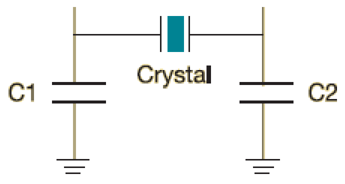


Figure 4.1: Loading Capacitors on Crystal

The effective loading required is specified by the crystal manufacturer, not by the microcontroller manufacturer (although the microcontroller manufacturer could make recommendations or force you to use a certain loading because of built in capacitors). This loading is what the crystal needs to oscillate at the frequency it is specified at. Any change in the effective loading changes the crystal's effective frequency, affecting the system using the crystal. The loading capacitors provide this capacitance, but they're not the only ones. Every pair of parallel plates with voltage across them forms a capacitor. Two parallel pins such as microcontroller pins have voltage and therefore introduce parasitic capacitance from the circuit. Internal microcontroller circuitry does the same because of the way the oscillator is constructed. The PCB on which the microcontroller is mounted also adds parasitic capacitance. From this you should realize that to get the effective loading capacitance you must take into account all these extra parasitics. Some of these are specified in the datasheet and others are usually assumed to be in a certain range.

Lets begin a simple exercise by assuming there are no parasitics. If we assume that we selected a 4MHz crystal with a parasitic capacitance of 7pF. We need the combination of C1 and C2, which are usually selected to have the same value, to give a loading of 7pF.

$$C_Leff = \frac{C1 * C2}{C1 + C2} = \frac{C1 * C2}{C1 + C2}$$

Therfore if C1 = C2, we have

$$C_Leff = \frac{C1^2}{2C1} = \frac{C1}{2}$$
$$C1 = 2 * C_Leff = 2 * 7 = 14pF$$

Therfore, both C1 and C2 should be 14pF to give 7pF of loading capacitance.

As was previously mentioned, this does not take into account the parasitics and the microcontroller effect. The oscillator inside the microcontroller introduces two capacitances generally called C_{XIN} and C_{XOUT}. Each appears in parallel with one of the external loading capacitors, resulting in the effective load capacitance of:

$$C_Leff = \frac{(C1 + C_{XIN}) * (C2 + C_{XOUT})}{(C1 + C_{XIN}) + (C2 + C_{XOUT})}$$

We are not finished, however. The PCB itself and the pins introduce a parasitic capacitance that should be taken into account as well. This parasitic capacitance appears in parallel to the previous parasitic capacitance and therefore is simply added (as is done for parallel capacitors):

$$C_Leff = \frac{(C1 + C_{XIN}) * (C2 + C_{XOUT})}{(C1 + C_{XIN}) + (C2 + C_{XOUT})} + C_{par}$$

PCB Parasitic capacitances are not always easy to measure. Because of this, it is common to assume a PCB parasitic capacitance of 2-5 pF.

It is worth the note that in the work above we assumed that we must provide external capacitors to load the crystal. This is not always true. The MSP430 accepts a 32kHz crystal for low power application. If you look at several designs with the 32kHz crystal you will notice there are no loading crystals. In these cases the MSP430 provides the loading capacitance internally to save space. The internal capacitance provided can be fixed, as is the case with MSP430 derivatives such as the MSP430F1611, or they can be software selectable from among several values. For example, the MSP430F1611 datasheet page 38 specifies C_{XIN} and C_{XOUT}, the Integrated input capacitances, to be both 12pF, meaning that the effective loading capacitance is 6pF and the crystal required should have a loading capacitance as close to 6pF as possible if parasitics are not taken into account.

Note that a 32 kHz crystal actually has a 32.768 kHz frequency (which is useful because 32768 is a power of 2, the base for binary numbers). The 32 kHz crystal to be used is usually of the watch crystal type. 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.

References:

MSP430 32-kHz Crystal Oscillators Application Note - slaa322b

Crystal Oscillators Using MSP430

Pierce-gate oscillator crystal load calculation

What is frequency at load capacitance?

4.4 Clock Sources

Clock sources are the term encompassing both crystals and internal oscillators. Their availability depends on your particular MSP430 derivative. For example, The MSP430F2274 does not include the XT2CLK (probably due to the low number of pins in the package). The following is a summary of the nomeclature of the clock sources:

- 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 you need more precision, use the external crystals at the expense of PCB space and some money. It is standard practice to use LFXT1 with a 32.768 kHz crystal, leaving XT2 to be used with a high frequency crystal.

4.5 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 from either the fast clock or the slow one. The CPU, however, can only be sourced from the fast clock called MCLK. 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. It is important to consider all the modules in the system and their frequency requirements to select their source. ADC sampling at high speeds cannot be done with ACLK.

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 7.1). 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 4.2.

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.

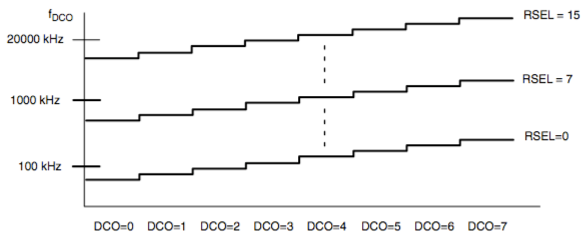


Figure 4.2: Selecting the DCO Frequency

Listing 4.1: Configuring EZ430 Clocks

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

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

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 many applications.

Listing 4.2: MSP430 Hello World with Clock Control

```c
#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 4.2 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 calls). The function itself adds some overhead.

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.

4.7 Considerations for using clocks

Clocks are among the most critical aspects of microcontrollers and the MSP430 because they sometimes require significant engineering effort to use. This is especially true for battery powered applications where we don't have the luxury of an electrical outlet. After all, these are the kind of applications for which the MSP430 was designed.

As previously mentioned, there is a direct tradeoff betwen clock speed and energy consumption. During the development process a bottom-up and top-down aproaches are usually used. We start the initial development at a limited clock speed and scale up as necessary to achieve each one of our tasks. On the other hand, the top-down approach begins with a fast clock and scales the clock down after functionality is proven in an attempt to conserve energy. In many cases, it is immediately apparent where we should begin. A hand held watch would require a bottom-up approach, while audio sampling would require top-down. In many cases, the use of modules and external devices complicates the clock selection process because of conflicing requirements. The flexible clock system of the MSP430 simplifies things, but tradeoffs will still be required.

Although this topic will be covered in later chapters, energy can be saved by turning clocks off. This will prevent us from using any modules for which the clock has been turned off, but should be done in cases where it's advantageous.

Chapter 5

General Purpose Input Output - GPIO

Digital pins are the main communication element that a microcontroller has with the outside world. They can go HIGH (1) or LOW(0) depending primarily on the system voltage (VCC). The datasheet for your particular MSP430 will provide details about the voltage swing used to represent these two positions, and they are usually several hundred millivolts from the system voltage.

5.0.1 Pin Multiplexing

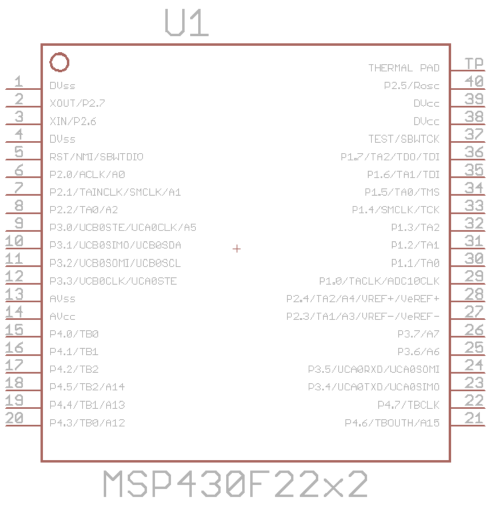


Figure 5.1: MSP430F2274 Schematic

Looking at the schematic representation of the MSP430 (MSP430F2274 in Figure 5.1) 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 pins among the internal modules. That is, each pin can only do one things at a time and you have to select what that function will be (upon startup there are some defaults which you will likely override with your code). Multiplexing can become a problem if you're in short supply of I/O pins and you're using others for specialized functions. In your design stage you should take all of this into account and plan which pins will be used (and with what functionality).

Most pins on the MSP430 devices include GPIO functionality. In their naming, this is indicated by PX.Y, where the X represents the port number and the Y the pin number in that port. MSP430 devices have several different ports, depending on the actual device, and these ports each have 8 pins (making byte access easy). For example, Port 1 is represented by pins P1.0 P1.1, P1.2, P1.3, ..., P1.7.

Each port is assigned several 8 bit registers that control them and provide information about their current status. Each register is designated PxYYY, where the lowercase x represents the port number and YYY represent the name of the functionality of the register.

- PxSEL - This register selects what functionality is shown at the pin, GPIO or an internal Module
- PxDIR - If the pin is set for GPIO in PxSEL, this register selects the pin direction - Input or Output
- PxOUT - If the pin is set for GPIO in PxSEL and If the pin has output direction as set by PxDIR, this selects whether it is HIGH or LOW
- PxIN - If the pin is set for GPIO in PxSEL and has input direction as set by PxDIR, this represents the level at the input (HIGH or LOW)
- PxIFG - Interrupt Flag for the corresponding I/O pin and it is set when that pin experiences the appropriate signal edge transition(either high to low or low to high).
- PxIES -Interrupt transition selection register. If a bit is 0, the pin will generate an interrupt and a flag is the transition is low-to-high. A 1in the bit will generate an interrupt and a flag if the transition is high-to-low.
- PxIES - Interrupt enable register. If a bit is set to 1, a pin will generate an interrupt if the appropriate transition has occurred.

In addition, for those MSP430 devices that support it, there are registers related to pullup/pulldown resistors:

- PxREN - Internal Pullup/pulldown resistor is enabled for the specified pin if a bit is set to 1. A 0 disables the resistor.
- PxOUT - If a pins pullup/down resistor is enabled, the corresponding bit in the PxOUT register selects pull-up(bit = 1) or pull-down(bit = 0).

Critical Note - In almost all MSp430 devices I've seen, only ports 1 and 2 have interrupt capability. Make sure to take this into account when designing the circuit. Following the convention, Port 1 has the registers P1SEL, P1DIR, P1OUT, and P1IN, among others. P2OUT controls the output level of each pin (1 or 0) in the port because they are mapped as follows:

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 :

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

```
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 the port, 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 PXDIR register, where X represents the port. In PXDIR 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 dependent 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 7.

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.

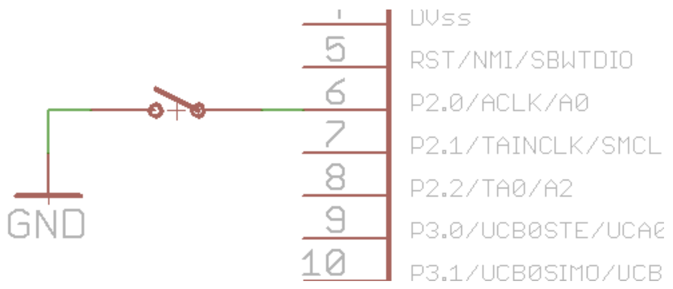
In PxREN, a 0 in a bit means that the pullup/pulldown resistor is disabled and a 1 in any bit signifies that the pullup/pulldown resistor is enabled. Once you select any pin to have the pullup/pulldown resistor active(by putting a 1 in the corresponding bit), you must use PxOUT to select whether the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

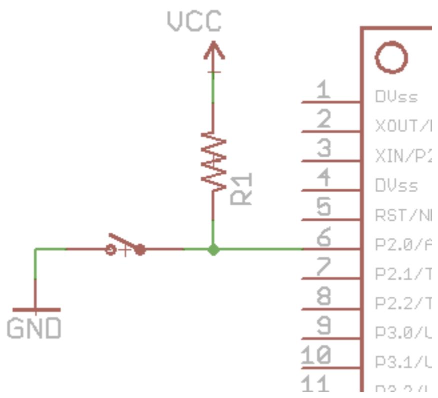
5.1 Switches

Adding switches to a microcontroller is one of the most popular ways to communicate with it. Many situations require you to let the microcontroller perform some operation. It's relatively simple to do as well.

Consider first that your ultimate objective is to generate a clean high to low or low to high transition. The switch at an initial moment is either high or low and the pressing of the momentary button (you could use a different kind of switch as well) results in a change in the voltage level.



(a) No Pullup Resistor



(b) External Pullup Resistor

Figure 5.2: Connecting Switches to MSP430

Figures above show how a switch is connected to P2.0 (a GPIO with interrupt capability) of the MPS430. Lets think a bit about what happens when the system is powered up, as in case 5.2(a). Here, the switch is normally open and the P2.0 is left floating and this is very bad . When the switch is pressed, P2.0 is connected to ground. There is no clear voltage transition because the initial condition is floating. Similar designs have been used by many newcomers and should be avoided. A design such as that is possible if there is an internal pull-up resistor (more about this later).

Lets analyze the second case 5.2(b), where the designer included an external pullup resistor. Here, the switch is also open initially, 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 get connected to ground. P2.0 experiences a transition from high to low. The resistor has current flowing through it and although it is wasted as heat, it is very small and lasts for only a few moments for the duration of the switch press.

Many microcontrollers include internal pull-up or even pull-down, which free the designer from having to include them. In this case, simply connect the button to either VCC and GND and it should work.

5.1.1 Debouncing

Up to now, we've assumed that the switch is perfect. In reality that is not so. The mechanical construction of the switch results in internal vibrations (bouncing) in which the metals bounce until they settle and make contact. During this time, a switch can result in many connections/disconnections that a microcontroller will interpret as multiple presses when in fact a user only pressed once. The end result is that multiple interrupts will be generated and can cause issues in the end application.

What to do? Well, we know that human beings cannot press a button every millisecond. Several approaches are possible, each with its own advantages/disadvantages. The most basic one is to accept the first transition/interrupt and ignore the rest by waiting for a set amount of time (using a timer or a for loop to waste cycles). The amount of time to wait depends on the switch to be used, but 40ms seems like a safe bet in many cases. If you can't wait that long, measure the bouncing time using an oscilloscope and create a worse case estimate.

A much better way to address bouncing is to check the switch's logic level after several milliseconds have been passed, ensuring that the first interrupt was not a momentary malfunction.

5.2 LEDs

Another important GPIO use is to control LEDs as indicators. The first way to perform this is to simply connect the led to the GPIO and drive it using the pin. A resistor of an appropriate value should be added in series with the LED to limit the current because the microcontroller will source (provide) the current and its capabilities in this respect are limited. Typical LEDs require on the order of tens of milliamperes. For low power microcontrollers this figure reaches the upper limit of the current sourcing capability of the microcontroller. Because this current is shared among the GPIO, adding LEDs can limit the drive capability of I/O. This might require buffering and complicate the circuity. A solution commonly used is to connect the LED in reverse, as shown in figure **??**. Here, the microcontroller is used to sink the current, which it is usually able to do more readily. Unfortunately TI does not provide much information in this respect and some experimentation is in order.

The value of the series resistance varies depending on the current required to drive the LED and the VCC of the system. Typical values are 220, 330 and 470 Ohm.

5.3 Bit banging

Bit banging refers to the implementation of a digital communications protocol from GPIO primitives. In other words, using a predefined sequence of high and low on the GPIO, we can emulate a protocol such as SPI. When dedicated hardware is unavailable (such as when you've used up all possible modules that are available), bit banging may be one of few choices left without resorting to using a different device. Because the implementation is purely software, it is very inexpensive. However, it has several drawbacks, primarily its limited .

As you previously learned, each module on the MSP430 requires a clock which is used for timing. When you perform bit banging, the CPU is manually modifying the GPIO registers to perform the bit banging. Therefore, for each bit to be sent, several cycles are required. This is in contrast to the mostly single cycle nature of dedicated hardware. In effect, the rate at which bit banging can be performed is several times slower than dedicated hardware and results in increased latency.

Another major issue is the processing time required from the CPU. Instead of working on more useful things, the CPU spends it time communicating. This takes a huge toll, especially for large amounts of data

It is advisable to stay away from bit banging as much as possible and only to use it as a last resort or if latency is not critical in the application. One application in which I've adopted bit banging is reading the serial ID from a one wire IC. Because this is only necessary at bootup, there's no load on the system while it's running and it simplifies the hardware design significantly.

Chapter 6

Timers

6.1 Introduction

Timers are common on all microcontrollers . They usually come in a variety of flavors and with different capabilities, however their working is relatively simple.

The timer is provided a clock source and counts a certain amounts of steps. When it reaches the limit of those steps it "fires". The number of steps and the frequency of the clock source determine the time that it takes the timer to fire. For example, if our timer is sourced using a 32768Hz crystal and we were to count 32768 times it would fire in 1 second.

Using a certain clock frequency we can create a trigger that is a subset of that frequency. In the EZ430, no external clock is provided. There are two sources: the VLO and the fast DCO. The VLO is usually running at 12kHz. If we were to count 12000 times, it would take us one second. Note that the timer is a peripheral and does not require the CPU to count anything. Rather it can inform the CPU with an interrupt.

6.2 Accuracy

A major issue with the selection of the clock source arises that affects precision. Using the VLO we are limited because of its speed because of the binary nature of the register that holds the counter. The fastest speed that the VLO would allow us to do is: 12000/1 = 12kHz. In fact it will be a bit less because of execution time. However it's reasonable to see that 1 is the minimum number of counts.

Chapter 7

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.

7.1 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 they 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 two. 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 resets 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 possible 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 nor 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 7.1: EZ430-RF2500 Toggle LED using Switch - Polling

```c
#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 LFXT1 to the VLO @ 12kHz
  BCSCTL3 |= LFXT1S_2;
}
```

The code is listing 7.1 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 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 are CPU is using. Second, lets assume that 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 7.2: EZ430-RF2500 Toggle LED using Switch - Interrupt Flag

```c
#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 7.3: EZ430-RF2500 Toggle LED using Switch - Interrupt

```c
#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 7.3 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 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

7.2 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 7.1, 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(LPMx_bits + GIE);
```

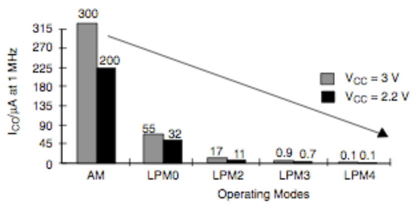


Figure 7.1: Low Power Mode Savings

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 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 7.3. It incorporates entering a LPM and gaining significant energy savings.

Listing 7.4: 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
  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
}
```

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.

7.2.1 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 7.5: EZ430-RF2500 Toggle LED using Switch - Low Power Mode 2

```
#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 8

Analog to Digital Converters - ADCs

8.1 Introduction

The ADC is arguably one of the most important parts of a Microcontroller. Why is this? The real world is analog and to be able to manipulate it digitally some sort of conversion is required. Although many techniques are possible, using an Analog to Digital Converter (ADC or A2D) is the most common when good signal fidelity is necessary. A more simple interface could be some threshold circuit that would allow you to know whether a signal passed a certain threshold. The resolution is obviously very poor and would not be used to sample real signals unless it 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 assume the ADC samples at a particular frequency and it does so perfectly to the available resolution.).

We will cover the ADC present in the MSP430F2274, a 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 this name indicates the general idea behind the conversion. There are many architectures out there.

The ADC samples and quantizes the input waveform. Sampling means that it measures the voltage of the waveform at a particular moment in time. In reality, the ADC cannot use an impulse but rather a larger time frame to sample so that the value is stable. When we talk about the sampling frequency, we refer to the fact that the ADC will be continuously sample. This is the most common mode discussed in DSP courses. However, an ADC can sample one or in many other ways, depending on the application. Quantization means it assigns a particular value, usually binary, to represent the analog voltage present. Both of these important concepts will be discussed next.

8.2 ADC Parameters

8.2.1 ADC Resolution

Probably the most important and most cited parameter of the ADC is its resolution, usually expressed in bits. The MSP430F2274 in our case has 10-bits of resolution. The resolution is an extremely important parameter because it dictates how the lowest difference we can differentiate. The higher the resolution, the more we can distinguish between minute differences in the voltage.

The resolution is specified in usually bits. This tells us how many different voltage levels we can represent.

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

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 changes in the signal and your digital representation better approximates the original. For 16-bits we have $2^16 = 65536$ and for 24-bit we have more than 16 million different levels, a huge increase due to the exponential nature. Note however that noise is a prevalent issue in ADCs and becomes the major constraint at a certain point

The question now comes to how the number of levels is related to the input voltage? This is simple. The number of levels possible is divided equally for the voltage range. If we have a 1V on the ADC's input pin and assuming the ADC is operating with a voltage range of 3V, each successive level from 0V to 3V is:

$$\frac{3V}{1024-1} = 0.00293V = 3mV$$

If the signal changes by less than 3mV we can't distinguish the difference, but every 3mV change in signal level translates into an addition of 1 to the binary number. If we have 0V at the input (without any noise), then we have 10 zeros (since it's a 10-bit ADC and no signal at the input). 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 staircase. For each range of voltages we assign an equivalent binary number usually referred to as the "code". The binary representation is usually replaced by hex since it is easier to read, as discussed in previous chapter. Note however that this is the ideal case and no offset or errors are taken into account.

An example of this is is shown in the next figure

The following shows a simple 3-bit ADC. Such ADC does not exist but serves to show how each new level is represented in binary:

Unfortunately, we cannot cover all the important details of ADCs. The interested reader is referred to the following excellent sources of information:

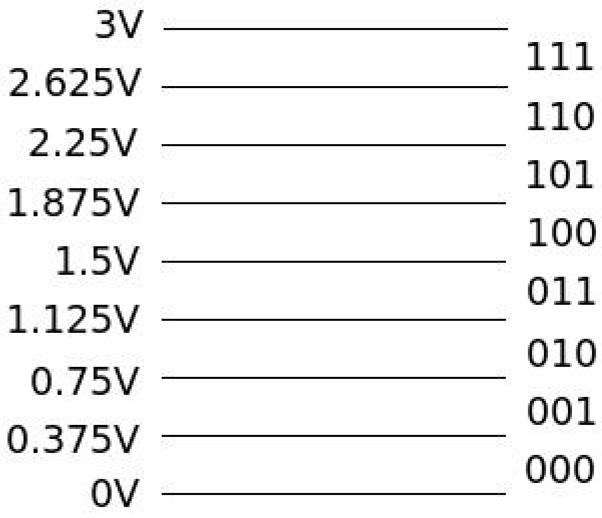


Figure 8.1: 3-bit ADC Example

- Application Note: SLAA013: ”Understanding Data Converters”. Texas Instruments : http://www.ti.com/litv
- Book: ”Data Converter Handbook”. Newnes 2004. ISBN - 0750678410

The general equation of the MSP430F2274 (which can be assumed for most other ADCs given a few changes) is:

$$N_{ADC} = 1023 \times \frac{V_{in} - V_{R-}}{V_{R+} - V_{R-}}$$

Notice that 1023 is is $2^10 - 1$.

A simple example follows: Lets assume that the input voltage is 1V. V_{R+} and V_{R-} represent the upper and lower limits of the voltage range. In our case we will have $V_{R+} = V_{CC}$ and $V_{R+} = V_{CC} = V_{SS} = 0V$.

Using the equation above we have:

$$N_{ADC} = 1023 \times \frac{1V - 0V}{3V - 0V} = 1023 \times 0.333 = 341 = 0x155$$

The binary representation is not shown because it is large and hexadecimal represents it nicely. Notice that 1V is exactly a third of the voltage range and therefore the output code will be a third of all the maximum number, 1023 in this case.

8.2.2 ADC Sampling Frequency

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 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. Different signals at the input of the ADC will require different sampling speeds. It is possible to oversample by sampling at a very high speed and therefore be able to capture different signals. Realize that if not sampling fast enough, very quick events can not be detected. Speech signals for example are generally considered to have important information for the region of 300Hz to 4000Hz. Using Nyquist's theorem we realize that we must therefore sample at a frequency twice the highest frequency (This is for low pass signals, i.e. signals where the lowest frequency is relatively close to 0Hz).

8.3 ADC Example - Temperature and Voltage

Instead of going over the details of the ADC, we will first see how a simple ADC application can be done using the MSP430F2274. This is done to give a good general picture of how the ADC can be quickly configured. Later more advanced features will be discussed.

The code to sample the integrated temperature sensor of the MSP430F2274 and the VCC is listed next:

Listing 8.1: EZ430-RF2500 ADC Sampling of Temperature and VCC

```
#include "msp430x22x4.h"


volatile unsigned int i;      // volatile to prevent optimization

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

  // Configure MCU Clocks
  BCSCTL1 = CALBC1_1MHZ;
  DCOCTL = CALDCO_1MHZ;
  // Set LFXT1 to the VLO @ 12kHz
  BCSCTL3 |= LFXT1S_2;

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

  volatile long temp;
  int delay;
        int temp, volt;

      // Configure ADC to Sample Temperature Sensor
      ADC10CTL1 = INCH_10 + ADC10DIV_4;       // Temp Sensor ADC10CLK/5
      ADC10CTL0 = SREF_1 + ADC10SHT_3 + REFON + ADC10ON + ADC10IE + ADC10SR;
      for( delay = 240; delay > 0; delay-- );     // delay to allow reference to settle
      ADC10CTL0 |= ENC + ADC10SC;             // Sampling and conversion start
      __bis_SR_register(CPUOFF + GIE);        // LPM0 with interrupts enabled

      // Read the conversion data
      temp = ADC10MEM;
33       ADC10CTL0 &= ~ENC;      // Stop ADC Conversion
34
35       // Reconfigure ADC to Sample VCC
36       ADC10CTL1 = INCH_11;                         // AVcc/2
37       ADC10CTL0 = SREF_1 + ADC10SHT_2 + REFON + ADC10ON + ADC10IE + REF2_5V;
38       for( degC = 240; degC > 0; degC-- );      // delay to allow reference to settle
39       ADC10CTL0 |= ENC + ADC10SC;               // Sampling and conversion start
40       __bis_SR_register(CPUOFF + GIE);          // LPM0 with interrupts enabled
41       volt = ADC10MEM;
42       ADC10CTL0 &= ~ENC;
43       ADC10CTL0 &= ~(REFON + ADC10ON);          // turn off A/D to save power
44 }
```

Listing 8.1 is heavily commented for easier understanding. As with any program, we begin by setting up the basic elements, clocks and I/Os. Line 25 is where the actual ADC use begin. We first must setup various parameters. In this case the channel and the clock of the ADC. The channel refers to the pin which will be sampled. The temperature sensor is integrated internally and therefore in this case does not belong to any pin. Channel 10 is the channel for the temperature sensor. On the same line we also configure the clock divider. This takes the clock source of the ADC and divides it by 5. More on why this is done later.

Line 26 configures other parameters. In this case we select the reference to be 1.

8.3.1 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).

8.3.2 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 9

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 Periheral Interface (SPI), the Universal asynchronous receiver/transmitter (UART), and I^2C.

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 I^{2}C 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: USCI_A0 and USCI_B0. USCI_A0 is capable of running UART,LIN, IrDA, and SPI USCI_B0 is capable of running SPI and I2C.

However, TI designed the EZ430-RF2500 and already assigned the functionality of the modules. USCI_A0 is allocated for UART communications with the PC, while USCI_B0 is connected to the CC2500 and needs to be in SPI mode.

9.1 Serial Peripheral Interface (SPI)

SPI is a synchronous 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 9.1: 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 9.1 shows SPI with only 1 slave and with multiple slave, so that the use of the Slave Select (or Chip Enable) is apparent.

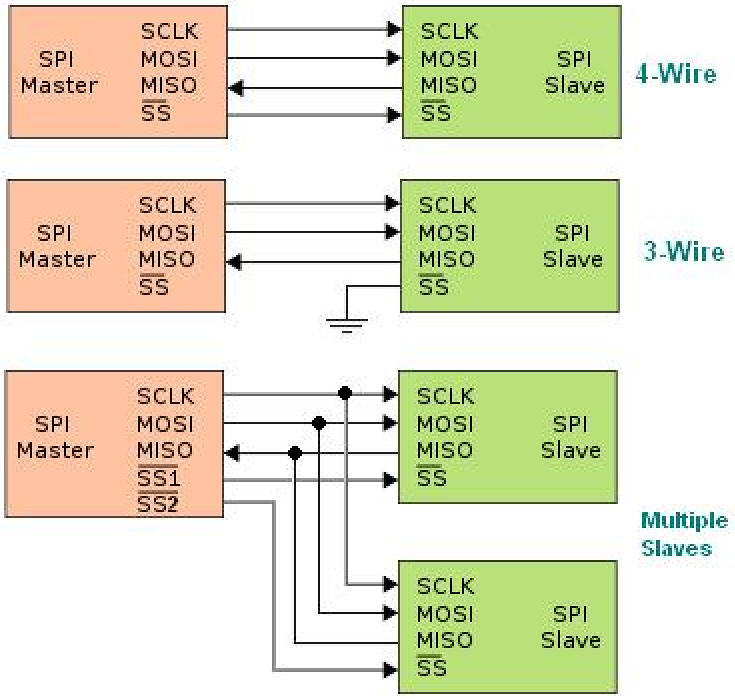


Figure 9.1: SPI Configurations

9.1.1 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:

Table 9.2: SPI Pins in the MSP430F2274

Pin	Peripheral Module Functionality(Multiplexed)
P3.0/ UCB0STE/ UCA0CLK/ A5	General-purpose digital I/O pin USCI_B0 slave transmit enable USCI_A0 clock input/output ADC10, analog input A5
P3.1/ UCB0SIMO/ UCB0SDA	General-purpose digital I/O pin USCI_B0 slave in/master out in SPI mode SDA I^2C data in I^2C mode
P3.2/ UCB0SOMI/ UCB0SCL	General-purpose digital I/O pin USCI_B0 slave out/master in in SPI mode SCL I^2C clock in I^2C mode
P3.3/ UCB0CLK/ UCA0STE	General-purpose digital I/O pin USCI_B0 clock input/output USCI_A0 slave transmit enable
P3.4/ UCA0TXD/ UCA0SIMO	General-purpose digital I/O pin USCI_A0 transmit data output in UART mode slave in/master out in SPI mode
P3.5/ UCA0RXD/ UCA0SOMI	General-purpose digital I/O pin USCI_A0 receive data input in UART mode slave out/master in in SPI mode

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 USCI_A0 and USCI_B0 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 USCI_B0 since USCI_A0 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:

```
P3SEL |= 0x0C;                          // P3.3,2 USCI_B0 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 USCI_B0). 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.

9.1.2 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_CSn_PxOUT &= ~TI_CC_CSn_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_CSn_PxOUT |= TI_CC_CSn_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 10

UART

In this chapter we will cover the UART module in the MSP430 to enable communications with a computer. The Universal Asynchronous Receiver/Transmitter (UART) module is part of both the Universal Serial Communication Interface (USCI) Module and the universal synchronous/asynchronous receive/transmit(USART) Module. Depending on which derivative of the MSP430 you are using, you will use either USCI or USART and enable one of the modes of operation such as UART, SPI, I2C, etc. In this section we will focus on the UART functionality in either of these. Note that you may only use the module to do one type of communication at a time. However, it might be possible to have several connections to the USART/USCI Modules and switch between protocols, but this can be problematic at times and will not be addressed. Also note that although the names of registers may differ when using the UART in either the USCI or USART modules, the major approaches are the same and only minor modifications should be required such as register names.

10.1 Hardware Connection

UART is simple to connect and it uses 2 lines: TX (Transmit) and RX (Receive). No clock is required because it's asynchronous.

When interfaced to another module that uses UART, you will need to cross connect the lines. The TX of one device will connect to the RX of another device, while the RX will be connected to the TX of the second device, as shown in figure 10.1.

RS232 is a common connection between PC and devices, and it can be easily converted to RS232 using a level converter. Another popular solution is to connect the UART of the MSP430 to a USB to UART converter such as the FT232RL and FT2232 from FTDICHIP. These ICs creates a virtual COM port on the host machine that allows you to seamlessly transfer bytes between the host machine and the MSP430.

10.1.1 UART connectivity on the MSP430F1611

The MSP430F1611 includes two USART modules, USART0 and USART1. Both are capable of using UART independently of each other.

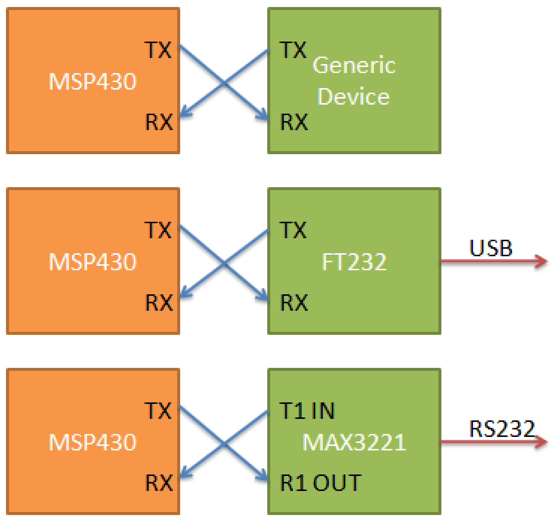


Figure 10.1: Connecting UART

UART0 uses P3.4/UTXD0 as TX and P3.5/URXD0 as RX
UART1 uses P3.6/UTXD1 as TX and P3.7/URXD1 as RX

10.1.2 UART connectivity on the MSP430F2274

The MSP430F2274 includes two USCI modules, USCI_A0 and USCI_B0. Only USCI_A0 is has UART functionality and and uses P3.4/UCA_0TXD /UCA_0SIMO as TX and P3.5/UCA_0RXD /UCA_0SOMI as RX.

10.2 Using UART

From the name of the peripheral you can easily guess that UART sends bit serially, one bit after the other, as opposed to parallel. However we don't usually send individual bits because of the fact that they convey little information by themselves. Rather, the UART uses a buffer that is one byte (8 bits) long to which information is written.

On a deeper level, the module sends more than just the byte a user places in the buffer. UART Typically sends 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 bits depend on the configuration of the UART module, but the most common configuration is 8 data bits, one stop bit and no parity. Extra bits that are sent are ultimately discarded and not stored at the destination buffer. The result of the parity check is however available. Figure 10.2 shows the bit arrangement in the MSP430's UART Module:

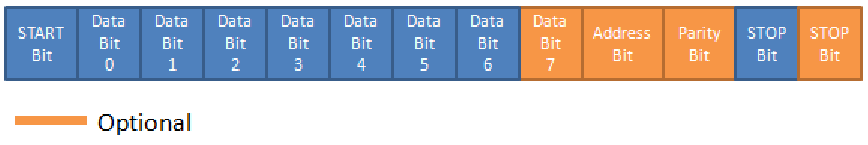


Figure 10.2: UART Bits

10.3 Configuring the UART

Before we can use the UART successfully to send data, both the host computer and the UART Module must be configured. Because the communication is asynchronous, if the two are not configured correctly then errors will occur because the interpretation of the timing of the bits will be off.

To better explain how to configure the UART module, we will use the following code as an example.

Listing 10.1: Configuring UART

```
P3SEL |= 0x30;              // P3.4,5 = USART0 TXD/RXD
ME1 |= UTXE0 + URXE0;       // Enable USART0 TXD/RXD
UCTL0 |= CHAR;              // 8-bit character
UTCTL0 |= SSEL0;            // UCLK = ACLK = 32.768kHz
UBR00 = 0x03;               // 32.768kHz/9600 - 3.41
UBR10 = 0x00;
UMCTL0 = 0x4a;              // Modulation
UCTL0 &= ~SWRST;            // Initialize USART state machine
IE1 |= URXIE0 +  UTXIE0;    // Enable USART0 RX/TX interrupt
```

The code above demonstrates that the following steps are needed:

1. Select the I/O pins for UART
2. Enable UART TX, RX, or Both
3. Select the character format - 7 or 8 bits
4. Source the UART Module with a clock of a certain frequency
5. Set the baud generator correctly so as to get a correct baud rate from the clock sourced
6. Enable the UART module state machine
7. Enable Interrupts if required

We will take the code and improve upon it to understand it and adapt it to most any MSP430 UART modules.

10.3.1 Selecting the UART Function for Pins

As we mentioned previously, the MSP430 (and many other microcontrollers) multiplexes the functionality of various modules at the physical pins. In our case the we can select 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.

The functionality selected is determined by PxSEL registers, where x can represent 1,2,3 ... ports, depending on which port the UART module is located. Each of the 8 bits in the register controls the functionality of the pin. As an example, the MSP430F1611 has two modules which support UART functionality. From the MSP430F1611 datasheet, pages 7 and 8, you can see that the UART pins that can be used:

P3.4/UTXD0
P3.5/URXD0
P3.6/UTXD1
P3.7/URXD1

P3.4 indicates pin 4 on port 3, while P3.5 indicates the same port but pin 5 and as you can see, they can be a General Purpose digital I/O pin or a UART pin. Because the pins are in port 3, we need to modify P3SEL. The question is how should we modify it? In the User's Guide for the MSP430F1611 we find that:
Bit = 0: I/O Function is selected for the pin
Bit = 1: Peripheral module function is selected for the pin

To select UART functionality for P3.4 and P3.5 we use the following c language statement that sets the two bits for pins 4 and 5, selecting them for the UART module:

```
P3SEL |= BIT4 + BIT5;
```

10.3.2 Enabling UART RX and TX

The UART module is very flexible. It can control the RX and TX functionalities individually.In our case we will enable both. To do this the following code is used:

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

Of course, for different UART modules we will need to use a different register settings. This can be easily found by looking at the user's guide for the particular MSP430.

10.3.3 Select the character format

Different settings are available for the communications between the two devices. Among these are the number of data bits, parity and stop bits. However, the most common setting, designated 8N1, is 8 data bits, no parity and 1 stop bit. Because of the default settings of the registers, parity is disabled by default and the stop bit is 1. The only thing left to do is to make the data bits 8 and not 7 as follows:

```
UCTL0 |= CHAR;    // 8-bit character
```

Some other options include PENA for enabling parity, PEV for selecting Odd or Even parity, or SPB for selecting two stop bits. An 8-bit character is normally used because it allows us to use ASCII, as well as allowing for a larger range of numbers to be represented

10.3.4 Selecting A Clock

Just like most modules in the MSP430, a clock must be chosen for the UART, and this is arguably the most critical task. Remember that the UART module sends bits every x amount of time. The timing for the bits 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).

In general, the clock frequency dictates the maximum baud rate allowed. ACLK, which is the auxiliary clock and is usually 32kHz or less (although in some cases it can be sourced from an external high frequency crystal). This means that in reality, and to keep errors low, we should select a baud rate of 9600 baud or less. Higher baudrates require a higher clock and this is normally sourced from SMCLK and MCLK, which themselves are sourced by the internal DCO or an external high frequency crystal. A crystal is always prefrable because it has a much better frequency stability and would almost guarantee better performance as far as error rates. It's only drawback is the added price and space requirements.

To select the clock source for 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

The define Column indicates an easy to use definitions defined in the header file msp430x16x.h that simplifies reading the code.

To select the 32kHz ACLK source we simply do:

```
  UTCTL0 |= SSEL0;    // UCLK = ACLK = 32.768kHz
```

If we wanted to select SMCLK we could do:

```
UTCTL0 |= SSEL1;    // UCLK = ACLK = 32.768kHz
```

```
UTCTL0 |= SSEL1;    // UCLK = SMCLK
```

UCLKI allows reversing the polarity, which is useful in certain situations but in fact rarely used.

10.3.5 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:

Baud Rate
1200
2400
9600
19200
57600
115200
230400

Using the desired baud rate and the clock rate we can calculate the divisors required to generate the baud rate. Extensive information about this is available in the User's Guide and other sources. A simple way to calculate the registers is to use one of the many available calculators:

http://mspgcc.sourceforge.net/baudrate.html

These calculators use the same methodology that will be presented.

Given a baud rate **b = 9600** and a clock speed **c = 32768** we have:

divisor = c/b = 32768/9600 = 3.413

The integer part of this divisor is easily implementable and you can see so in the code:

```
UBR00 = 0x03;    // 32.768kHz/9600 -> 3 integer part
```

16-bit Integer divisors are possible because there are two registers, UBR00 and UBR01, both 8-bit. UBR00 contains the upper byte while UBR00 contains the lower byte (lower 8 bits). In this case 3 is not greater than 255 so that we place 0x00 in the upper byte and 3 (or its hex equivalent 0x03) on the lower byte.

However, we need to get as close as possible to 3.413 or else our baudrate will be off and we would like it to be as close as possible. In cases where there is no fractional part we ideally don't need to deal with it, but it occurs in most every case because the clock used is never perfect. Using the modulation register we can approximate the fractional part of 3.413 to some degree of accuracy.

Multiplication	Result	Bit Result
0.413 * 2	0.826	0
0.413 * 3	1.239	1
0.413 * 4	1.652	0
0.413 * 5	2.065	1
0.413 * 6	2.478	0
0.413 * 7	2.891	0
0.413 * 8	3.304	1
0.413 * 9	3.717	0

To calculate the number of bits we need to set in the modulation register we multiply the fractional part by 8: $8 * 0.413 = 3.304$ Rounding gives us 3, so that tell us 3 of the 8 bits in the modulation register need to be set. To determine which bits to turn on we to successively multiply the fractional part:

The important thing to note is whether, for every addition of one to the multiplying factor, we jump to the next integer. For example, multiplying by 2 resulted in 0.826 which has an integer of 0. Multiplying by 3 means 1.239 which has an integer part of 1. Because they are not the same we say that we set it that bit because it carried. On the other hand from 3 to 4 we still have 1 as the integer part, so we don't set. Notice that we have a total of 8 multiplications so we can set the 8 bits of the modulation register. Starting from the multiplication by 2, we consider the first bit to be 0 because multiplication by 1 previously has the same integer as multiplying by 2. The second multiplication by 3 changes the integer part so we do set that one. The multiplication by 4 doesn't so we don't set that bit.

Note that when we first multiplied 8 * 0.413 = 3.304 we said that we needed to have 3 bits set (1) when rounding down in the register and from the table it is apparent that we have obtained only 3 bits set. All we need to do is to consider where do we start with the bits, from the bottom or top? The answer is that the bit for which we multiplied by 2 is the Least Significant,i.e we start from the bottom of the table and go up, resulting with the following:

$$01001010 = 0x4A$$

NOTE: Some sources state that we should start from the top, but this does not seem to comply with TI's own algorithm.

10.3.6 Enabling the Module

Once all the settings are complete, we can enable the module as follows:

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

10.3.7 Enabling Interrupts

Although it is possible to perform UART operations like as receiving bytes using polling techniques, it is most commonly done using interrupts and these must be enabled. The UART reset state does not have to be enabled to turn them on or off. This can be done as follows:

```
IE1 |= URXIE0 + UTXIE0;    // Enable USART0 RX/TX interrupt
```

Although both interrupts are enabled in this case, it is more common to use the RX interrupt.

10.4 Configuring the Host Computer

Once the MSP430's UART is configured, we need to configure the software on the Host computer, if it is used. This varies depending on the software. Some free software options are:

Name	URL
Putty	http://www.chiark.greenend.org.uk/ sgtatham/putty/
RealTerm	http://realterm.sourceforge.net/

Both of these programs are free and open source, although RealTerm is more terminal oriented and has many more useful options than Putty.

To configure the terminal, find the COM Port that is connected to the MSP430. In Linux this is usually done a command such as **ls /dev/ttyUSB*** Which will show a list of all USB VCP ports. Note that in some cases the chipset used maps differently, but the above technique works well in most cases, especially for FTDI Chip USB to UART ICs. In Windows finding the COM port is done either by using the Device Manager in the Control Panel. RealTerm itself shows all possible COM Ports.

Set the terminal to this port and to use the baud rate you've previously selected. Remember to set the data bits to 8, the parity to none and the stop bits to 1 unless you've configured the UART differently.

10.5 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. This is in fact very simple for single bytes. Each UART module contains two buffers, a receive buffer and a transmit buffer. For the sending buffer, the user places data (a byte in this case) and the module sends it by serializing it and sending the appropriate bits. For the receive buffer it is the opposite, the bits are deserialized and converted into a byte.

10.5.1 Sending Bytes

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

```
TXBUF0 = 0x41;

TXBUF1 = 0x41;
```

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 checking the appropriate flag:

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

The MSP430 will keep executing the while loop (do nothing because of the colon) until the sending flag has been cleared.
This means we can put this together to send 2 bytes, one after the other:

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

Even when sending an individual byte, always use the while loop to ensure the buffer is ready before trying to transmit it.

10.5.2 ASCII

Since the UART buffer is one byte(when it is configured to use 8 bits), we can send any number from 0 to 255, or 0x00 to 0xFF. A major use of sending data is to send readable information. ASCII is a standard encoding supported by terminal emulators and the C compilers for the MSP430, which allows us to send readable characters. In this case ASCII maps a byte with a specific value to a specific character. For example, the lowercase 'a' corresponds to 0x61 while 'b' corresponds to 0x62.

It is not necessary to learn the ASCII table. Any compiler will automatically replace 'a' with the corresponding number without the user needing to explicitly specify that value. In the case of IAR and other compilers, it is also legal to perform the following comparison:

```
char buf = 'a';
if(buf == 'a')
// Successful -This will execute
else
// This will not run

// Some other code
```

More will be discussed about using ASCII to send readable strings to the Host computer.

10.6 Sending Multiple bytes

Sending single bytes can be useful in some cases, but real applications require sending many bytes. We will delve into how to send and receive multiple bytes and strings. It is your job as the designer to create a protocol if necessary to send useful data. Some of this will be covered later.

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 = 'A'; i < 'Z'; i++)
{
   TXBUF0 = i;
   while (!(IFG1 & UTXIFG0));    // USART0 TX buffer ready?
}
```

A simple for loop can be used to send bytes.

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

Value	4	8	15	16	23	42
Byte	0	1	2	3	4	5

The above array is a collection of decimal numbers with 4 at position 0 and 42 at position 5. The array has 6 elements 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.

10.6.1 Sending Strings

Sending strings to the PC is useful in several situations. One of them is when you wish to use a terminal program such as Putty to receive data or debug the microcontroller by printing readable information. This eliminates the need to build a custom application to read and write information to the port. This requires time and is not always possible(although since Putty and RealTerm are open source, adapting the code simpler than starting from scratch). To simplify our work of sending the strings we will create a simple function:

```
void sendstring(char * str, int len)
{
    int i = 0;
    for(i = 0; i < len; i++)
    {
        while (!(IFG1 & UTXIFG0));    // USART0 TX buffer ready?
        TXBUF0 = str[i];
    }
}
```

We can use the function above to send the string Hello World as follows:

```
sendstring("Hello World\\n",12 );
```

Note that
n is not two characters but rather one character, the newline character. Therefore only 12 bytes should be sent and not 13.

The function can be simplified to avoid counting the length of the string by using the compiler's string functions, specifically strlen, which gives the size of the string.

Listing 10.2: Send String Function

```c
void sendstring(char * str)
{
    int i = 0;
    for(i = 0; i <  strlen(str); i++)
    {
        while (!(IFG1 & UTXIFG0));     // USART0 TX buffer ready?
        TXBUF0 = str[i];
    }
}
```

Be cautious in using this function. Calling the function as follows is fine: sendstring("Hello World
n"); However, sending a fixed array with characters will fail. This is because the compiler automatically appends a
0 to the end of the string, which is how strlen finds the length. if the parameter to the function is an array that does not contain
0, strlen will not find the end and the results are unpredictable (yet they will always be disastrous).

10.7 Receiving Data

Another function that is very common is receiving data. On the MSP430 this is done almost automatically if it is correctly configured and the UART RX functionality is enabled. The only issue that must be resolved is that multiple bytes received successively overwrite the buffer. If the host computer sends two bytes and the user does not fetch the first one from the RX buffer before the second byte arrives, it will be lost since the buffer will be overwritten with the second byte.

The complexity of handling this problem depends on your application. In all cases,however, every time a byte arrives it should be copied to another buffer(an array for example). Once enough bytes have been received, we process them as needed. The number of bytes received can be fixed (processing every 5 bytes received for example) or flexible (such as using delimiters to indicate the start and end).

We will now discuss how to receive individual bytes and show how we can place them in a buffer for processing. The USART module can generate an interrupt to immediately execute code when a byte 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 in the interrupt service routine (ISR) is executed. Such an interrupt vector can be, for example, made to echo a character (send the character that was received back):

```c
#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 or the USCI RX vectors. The interrupt does exactly what its name means, it interrupts the microcontroller by stopping the current statement it is processing, and redirects it to the interrupt code, returning to the original statement that was being executed once the interrupt handling is finished. This might or might not be done quickly enough, depending on what is done at the interrupt. It is usually best to keep interrupts as short as possible, raise a flag to indicate that the data needs to be processed and end the interrupt service routine.

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 received character
}
```

OR

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

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

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

OR

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

To demonstrate receiving of multiple bytes, we will assume everything is initialized and interrupts are enabled.

Listing 10.3: Buffering UART Data

```
char buffer[256];
int bufpos = 0;
char process_flag = 0;
void main()
{
// System and UART Initialization Here

__EINT();

while(1)
{
   if( process_flag)
  {
    \\ Process bytes

     process_flag = 0;
  }
}
}

#pragma vector=USART0RX_VECTOR
__interrupt void usart0_rx (void)
{
     buffer[bufpos] = RXBUF0;
    if(bufpos == 255)
    {
        process_flag = 1;
        bufpos = 0;
    }
}
```

The code above is relatively simple. 256 bytes are collected, a flag raised and the information set for processing.

10.8 Framing

Using fixed sizes of bytes is not always a good idea. Whenever the size of the data to be communicated can vary, for example commands and data can be of different. In this case a more flexible method is desirable. This can be realized using delimiters, which is just using special characters before and after the actual data to indicate frame the data.

The methodology for using delimiters is in fact very similar to the Serial Line Internet Protocol (SLIP). Although we won't be using IP on top of the delimiters, it will be useful as a comparison. More information on SLIP is available here:

http://en.wikipedia.org/wiki/SLIP http://tools.ietf.org/html/rfc1055

We must first choose two bytes that will represent the start and end. This can be any value between 0x00 and 0xFF. It is better to use values that will not be used in any other way. 0x00 and 0xFF are all too common when faults and should be avoided. We will therefore choose the following and define them:

```
#define START_BYTE 0x7A
#define END_BYTE    0x7E
```

Listing 10.4: UART Framing

```
#define START_BYTE 0x7A
#define END_BYTE    0x7E
#define MAX_BUF_LEN 256

char uart_buffer[MAX_BUF_LEN];
int bufpos = 0;
char process_flag = 0;
int uart_packet_len = 0;
char frame_received_flag = 0;

void main()
{
  // System and UART Initialization Here

  __EINT();

  while(1)
  {
     if(frame_received_flag)
    {
      \\ Process bytes

       process_flag = 0;
    }
  }
}

#pragma vector=USART0RX_VECTOR
__interrupt void usart0_rx (void)
{

    /* Protect against buffer overflow */
    /* In this case overwrite */
     if(buffer_counter == MAX_BUF_LEN-1)
    {
        buffer_counter = 0;
    }

     /* Received Starting Byte */
    if(!packet_start_flg && (SERIAL_RX_BUF ==  START_BYTE))
    {
        packet_start_flg = 1;
        buffer_counter = 0;
    }
    /* Store Frame Bytes without SFB and EFB */
    else if(packet_start_flg && (SERIAL_RX_BUF != END_BYTE))
    {
        // Store in buffer
        uart_buffer[buffer_counter] = SERIAL_RX_BUF;
        buffer_counter++;
    }
    /* Received Stop Byte so process Frame */
    else if(packet_start_flg && (SERIAL_RX_BUF == END_BYTE))
    {
            frame_received_flag = 1;
            uart_packet_len = buffer_counter-2; // Discard START and END bytes
            packet_start_flg = 0;
    }
}
```

The code above is commented so it is easier to understand. The code starts by checking if the START_BYTE (0x7A in this case) was received. If it was, then every subsequent byte is stored in the buffer, as long as the byte to be stored is not the END_BYTE. If the END_BYTE is received, we set the flag indicating that the frame was received and reset the process of receiving the frames. In addition to this, the first for loop protects against frames larger than the buffer. This is necessary since writing past the array is dangerous and should never be done. To avoid losing frames we would need to increase the size of the buffer, which means more memory would be required.

10.9 Future Additions

The code presented above is of course only the beginning. There are, as always, issues with the code presented above:

1) Although we've received a frame, we don't know if all of the bytes received are correct. Using a Cyclic Redundancy Check (CRC) we can have much better certainty that the bytes have not been modified unintentionally.

2) If a frame is lost there is no retransmission.

3) It is possible that the START and END bytes are not unique and the frame delimiters detected are incorrect. Therefore, a more unique pattern such as using two bytes that are unlikely to occur should be considered.

Some of these options are likely necessary if the UART Error rate is higher than it should. If so, consider adding a crystal for a stable clock or reducing the baudrate.

Chapter 11

Wireless Communications with CC2500

11.1 Introduction

If you're using the EZ430-RF2500 you're likely attracted by its wireless transceiver. In fact, many of today's applications use some form of wireless communications, from WiFi to bluetooth. The part of the tutorial will discuss using the EZ430-RF2500's wireless capabilities at a basic level to transmit data from one node to another.

It is not possible to cover all the aspects of the transceiver, as there are many options. Rather, the main focus will be to discuss the most common features to transmit and receive data. The code provided is based on the SLAA325 application note from Texas Instruments. This application note concerns communications between an MSP430 and several of TI's transceivers which share many common aspects.

Although we will focus on using the CC2500, many issues in this section will be applicable to many digital sensors and devices that use the SPI interface or even another digital bus.

The EZ430-RF2500 includes the CC2500 transceiver. This is a 2.4GHz transceiver capable of up to 250kbps. It is an all digital transceiver that is accessed using the SPI bus.

Transmission of packets at its most basic level consists of:

1. Initialize the SPI interface
2. Configure the CC2500 with the correct frequency, output power
3. Fill the TX FIFO with the data to be sent
4. Tell the CC2500 to initiate transmission

For reception, an interrupt based method will be used as follows:

1. Initialize the SPI interface if necessary
2. Configure the CC2500 with the correct frequency, output power, and modulation
3. Configure the interrupt pins to alert MSP430 that a packet has arrived
4. When a packet is received, retrieve it from the RX FIFO

11.2 SPI Interface

Initialization of the SPI interface is similar to what was previously discussed. We must ensure that the SPI clock used does not go above the maximum frequency supported by the transceiver.

The transceiver has registers internally similar to the MSP430. However, since these are external to the MSP430, we must use the SPI interface to read and write to them. When a register is read, we send the name of the register to be read. The transceiver responds with whatever value that is stored in that register. When writing, we must provide two things, the register's address and the value which we want to write.

Physically the SPI interface of the EZ430-RF2500 consists of:

- CSN P3.0 UCB0STE
- SIMO P3.1 UCB0SIMO
- SOMI P3.2 UCB0SOMI
- SCLK P3.3 UCB0CLK

These 4 pins of port 3 of the MSP430F2274 are for the SPI interface connectivity using the USCI_B0 module. Because of the limited number of I/O on the board, it is possible to use pins P3.1 to P3.3 for another SPI device, although this device will require a CSN line of its own and both devices can't be talking to the Microcontroller at the same time.

The spi.h and spi.c include all functions that are needed to communicate with the CC2500. The TI_CC_SPISetup configures the SPI interface.

Note that this implementation uses a 3-wire interface, leaving us to manually control the CSN line.

The CC2500 improves on this even further. Two access modes are available for reading and writing: Single and Burst. Single access means we can only read/write to one register, with each following read requiring a completely new operation. Burst Access is used to access multiple registers sequentially without the need to give the each register's position. What the transceiver does is simply assume that the next value you provide will go to the next register. Table 37 in page 60 of the CC2500 datasheet shows the different registers available, what modes of access is possible, and whether they are read/write or just read.

Two registers are special, these are the TX and RX FIFO. A FIFO can be thought of as a large array. The data to be transmitted or data that was received is placed here by the microcontroller. These FIFOs are fixed at 64 bytes. This can result in some problems when the data to be sent exceeds 64 or so bytes. A solution to this will be discussed later.