MSP430 Tutorial

Very Important Low Power Processor For Embedded Systems Applications

Written By,

IIT Bombay Alumni Foundation's

Embedded Technosolutions

Introduction

Although there are many resources dedicated to teaching microcontrollers and the MSP430 in particular, we have always found that they failed to cover the basics, or made too many assumptions that were not valid when transitioning from a computer programming environment. Some seemed to assume you already had years of experience using microcontrollers. While this might be true, the MSP430, like any other microcontroller, has peculiarities that need to be explained and understood.

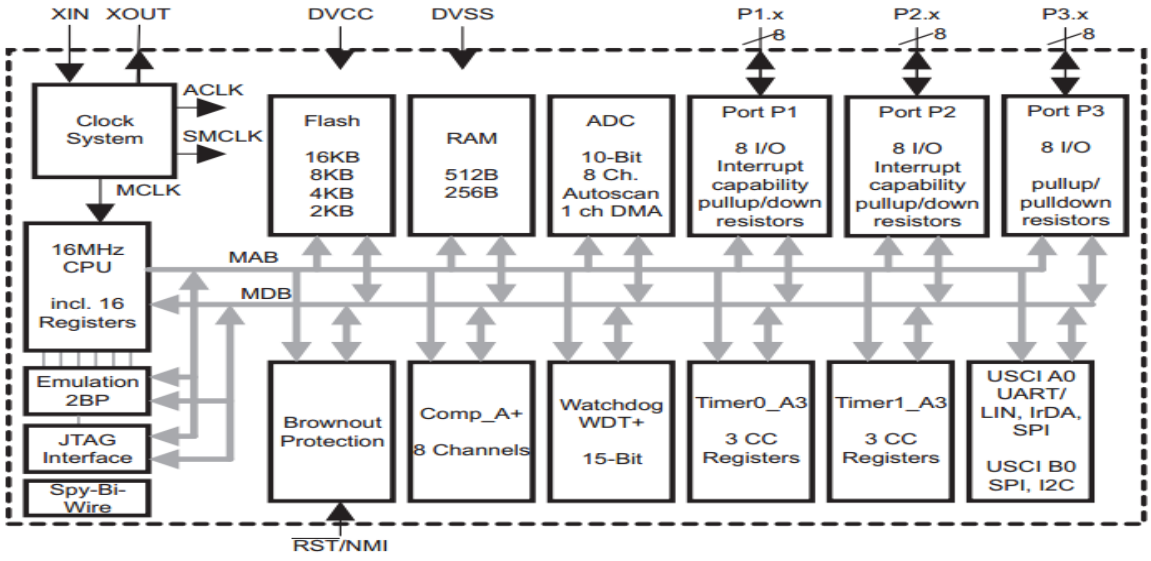
Programming for embedded systems and the MSP430 is not more difficult than programming an x86 on Windows or Linux. In fact, it is much better in that it exposes us to the little details of how the system operates (the clocks, I/O) at a level that anyone can learn. However, it does force us to make critical 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, 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 number system theory necessary to manipulate binary digits and digital logic as used in the microcontroller and all computers. Once this is understood, 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. We 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 we also wanted to keep the tutorial to be accessible for beginners and so the first part of the tutorial looks to lay a good foundation.

Introduction

To those who are unfamiliar with microcontrollers, they 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. Below we show a block digram of everything integrated inside a MSP430G2553 Microcontroller. You can see that it contains not just a CPU, but memory, non-volatile storage and many peripherals that are useful in real life applications.



A wide array of microcontrollers exists, far surpassing the capabilities of full-fledged computers in the 70s, 80s, and 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 hundreds 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. Low cost development tools, both hardware and software, allow anyone to start building a system. The knowledge of doing so is also more widespread. Still, no matter how good are the tools, they can never substitute real knowledge about the inner workings of microcontrollers. This is what this tutorial is about.

Chapter 1

MSP430 Development Hardware

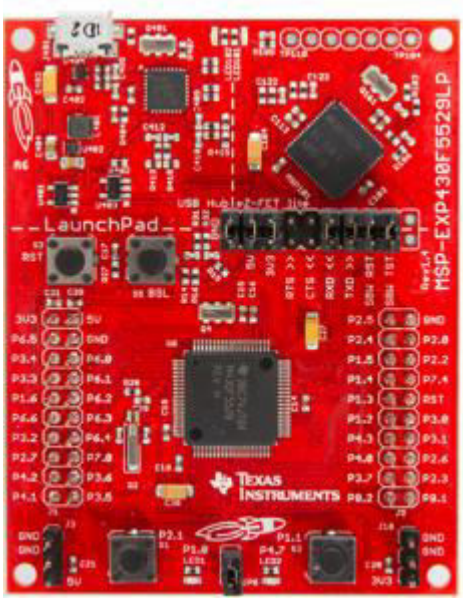
The MSP430 contains several families, from the low cost Value family to the most advanced F6xx family. Each family is usually targeted at a set of applications and contains a certain mix of peripherals. In a family, however, devices vary as far as the amount of Flash and RAM available. It is not unusual to develop using relatively large devices in a family, only to migrate down to reduce costs. Because of this, the tutorial will end up discussing several platforms, but will attempt to cover the lowest denominator first to make it accessible. The MSP430 Launchpad is the most accessible MSP430 platform and although the devices supported do not have all the peripherals of some of the mode advanced MSP430, it does cover so many peripherals that it makes it an ideal platform for starting.

The MSP430 Launchpad is an easy way to get started with the MSP430. For a long time the board was sold at a promotional cost of $4.30, although it's now available for $9.99. Still, the cost makes it difficult to say no.

The board contains a DIP socket capable of accepting most variants of the MSP430Gxx family. The most common device used is the MSP430G2553, which is a part running up to 16MHz with 16kB of flash and 512B of RAM. The USB connectivity on the board allows both programming with the on-board JTAG programmer, as well as UART communications for data transfer.

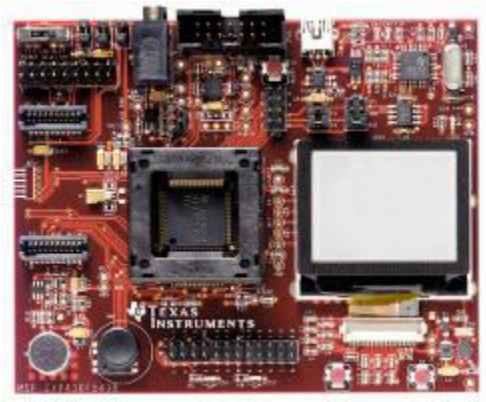
The board can be augmented with booster packs designed by TI and third parties that enable Wireless Communications, Wi-Fi, Batteries, Displays, and other elements.

MSP430F5529 Launchpad



This MSP430F5529 Launchpad is one of the latest Launchpads. The F5529 along with the F55xx family integrate a USB Controller, opening the door to new applications and possibilities previously requiring a dedicated USB to UART converter. For a price of $12.99 you get an MSP430 that can go up to 25MHz, has 128kB of Flash and 8kB of RAM. With USB you can implement CDC, HID and MSC classes so you can be a serial port, a mouse or act as an SD card. This board uses the same form factor as the classic MSP430 Launchpad, so booster boards can be reused.

MSP4305438 Experimenter Board



The MSP430F5438 and MSP430F5438A are quite popular devices due to the fact that they have 256kB of flash. Being some of the largest MSP430 devices makes them ideal for prototyping to support a wide range of applications. Some of the features of the board:

- 100 pin socket enabling quick insertion and removal of devices
- Dot-Matrix LCD with Backlight
- Audio Jack output with on-board Audio Amplifier
- 3-Axis Analog Accelerometer
- 5 position Joystick for navigation
- EM Connector headers supporting TI Low Power Transceiver Modules

EZ430-RF2500 Kit

Designed as a USB Stick, it is one of the most popular kits available from TI for those looking to try the MSP430 with a Wireless Transceiver. This inexpensive kit allows you to start using the CC2500 transceiver, a 2.4GHz radio that has become quite popular, especially given the kit's $49 price.

One side of the stick contains the FET programming circuitry allow Spy-Bi-Wire communications with the MSP430 for debugging. The other contains the MSP430F2274 with the CC2500 and all circuitry needed for a connection. The MSP430F2274 is a relatively small device, 32kB of Flash and 1kB of RAM, so applications can be limited (especially by the RAM). Out of the box demo shows connecting the EZ430-RF2500 wirelessly to monitor temperature and voltage.

JTAG Programmer

Programming, Debugging and Flashing the MSP430 is done via the JTAG interface, or its pin reduced version called Spy-Bi-Wire. Although this JTAG is based on the IEEE 1149.1 Joint Test Action Group (JTAG) specification, TI has made modifications which mean that a MSP430 specific programmer must be used. The USB-FET is probably the most common programmer. A parallel port version was once available but has since become extinct given the disappearance of parallel ports and the emergence of USB.

The USB FET programmer is supported by practically all compilers and IDEs, including Code Composer Studio, IAR Workbench and Open source tools. The USB-FET programmer can be purchased from TI.

Physically the USB FET uses a 14 pin ribbon cable to connect to a target board, with two rows of 7 pins each. A small arrow indicates pin 1, and it is usually hard to connect it incorrectly on boards that have the shrouded header. The pinout of the significant pins is as follows:

Pin #	Name
1	TDO
3	TDI
5	TMS
7	TCK
9	GNDs
11	RST/NMI

2	VCC_FET

All pins are required except for pin 2, VCC_FET. This pin provides a controllable voltage to the MSP430 if desired and if connected to the MSP430's VCC pins. One must be careful about drawing too much current due to current limiting of the USB FET itself. 50mA is typically the maximum recommended current. It is typically best to power the MSP430 separately if possible to ensure clean voltage rails.

In Some cases, pin 2 and pin 4 are both required by certain third-party programmers to ensure target has proper voltage. Refer to the connection diagram for your programmer to ensure that all the signals are connected.

MSP430 Supporting Circuitry

Although the MSP430 Launchpad and Development boards contain everything that is needed to run an MSP430, it is important to know what external components are needed to ensure reliable operation. The MSP430 is relatively self-contained, but it still requires a few external components. The list below is general. For best performance, follow TI's Hardware Tools Guide

Pull-up resistor of 47k with a on RST/NMI to ensure the MSP430 is not held in reset

- Decoupling capacitors on VCC, VCORE, and other pins located physically near the MSP430 for best performance. 0.1uF and decade values work best, but some applications require a filter network if transients are expected.
- JTAG connectivity as described in the JTAG Programming section, or Spy-Bi-Wire for devices without JTAG support
- MSP430 Core operates with 1.8V to 3.6V, but some modules may require 2.2V or 2.7V for proper operation
- Crystals as required for Timer/RTC/UART accuracy or USB operation

We will elaborate on some of these elements later, especially with respect to crystal selection and clock sourcing. In general, voltage requirements and crystal selection require careful consideration of the end application. MSP430 development platforms typically run at 3.6V, which makes them capable of running the MSP430 at the maximum frequency and support the operation of all modules. They typically also have footprints where 32kHz or high frequency crystals can be mounted.

Chapter 2

Getting Started with MSP430

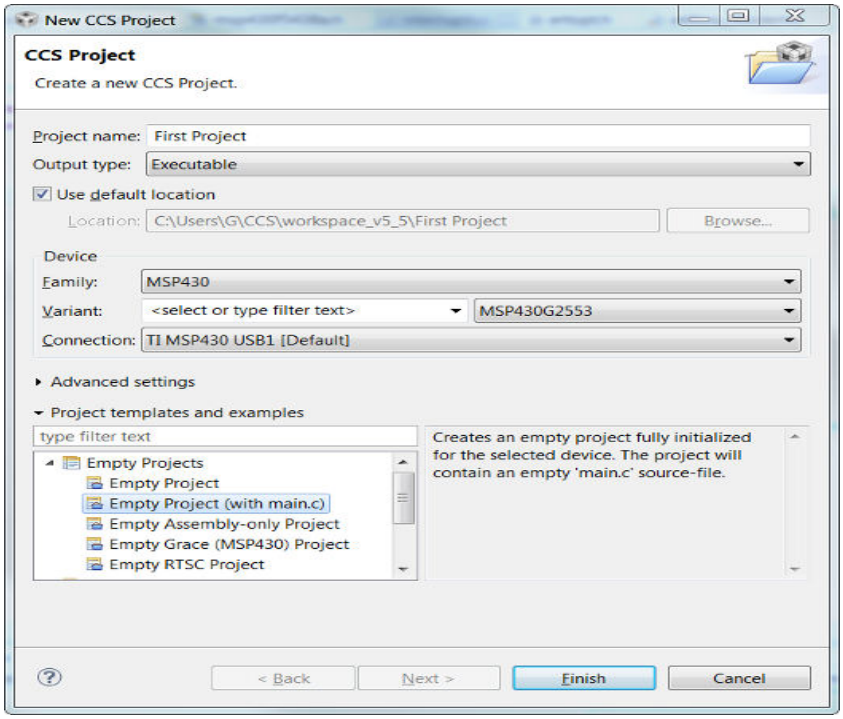
This tutorial is heavy with practical aspects of using the MSP430, and we hope you can follow the code, try and play with it. But, before we dive into number systems and operations we want to show you the reason we need to understand them. We believe that only by seeing code and running it will you begin realizing what pieces need to be understood to use the MSP430. This chapter will take you through a complete sample project. You might not understand all the code right away, but it will become clearer as we progress.

First Project in Code Composer Studio

We assume that you have been able to download and install Code Composer Studio, preferably version 5.5 or above. You can get a 90-day evaluation version from TI, or run it as a code size limited version. Although we're using version 5.5, most v5.x versions will work. The most important thing to ensure is that support for the MSP430 was enabled during installation, since CCS supports many TI devices simultaneously. Installation isn't particularly difficult, but the download can take a while.

The first step is to open Code Composer Studio and create a workspace. In the **Select a Workspace** dialog specifies a folder name, even if one doesn't exist. Although for this first

project we will place the project with the workspace for simplicity, during development it is best to separate them. The workspace is not usually needed for development. A project can be added to any workspace. CCS will open with several windows, but they are full of information we don't need. Go to the **Project** menu and then **New CCS Project**. We will create a project for the classic MSP430 Launchpad, which contains a MSP430G2553, so the dialog should be configured as follows:



Creating an Empty project (with main.c) simplifies creating the initial main.c file. The project will compile but do nothing. Replace the contents of main.c with the code below:

```
#include <msp430.h>
int i = 0;
void main()
{
   WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer
   P1DIR |= 0x01;
   while(1)
   {
      P1OUT ^= 0x01;
      for(i = 0; i < 20000; i++);
   }
}
```

Press CTRL+B or go to **Project-> Build All**. CCS will build the project and since our code is valid and it will compile and link. If all goes well, CCS will inform you in the console window that linking is complete. Note that CCS takes care of both compilation and linking specifically for the

MSP430 we specified, so that all the code will be placed in the correct flash locations when downloaded. If you have a Launchpad, connect it to the computer. It should automatically detect and install the drivers needed to run. CCS provides the drivers in its installation directory, so point windows there if needed.

Now press F11 or go to **Run->Debug**. If asked about low power, just press Proceed. CCS will download the code to the MSP430 and stop at the first line of main(). Press run and you will begin to see the LED blinking on the Launchpad.

Chapter 3

Number Systems & Operations

Although we are used to the base 10 decimal system for everyday counting, computers operate using a different system that uses only two digits, 1 and 0, to represent numbers. Counting in Binary, however, is quite unnatural for humans to use because even small numbers become long quickly. Using binary to represent numbers can be sometimes confusing but we will be using binary and hexadecimal values and operations extensively when programming the MSP430, so it's critical to understand how they operate.

A CPU, and by extension a microcontroller, is a number cruncher. The first thing we must understand is how to represent numbers in a manner that the compiler, and therefore the CPU, can understand.

Hexadecimal Number System

Because binary numbers tend to get long, hexadecimal is often used to represent the same values. Instead of base 2 as in binary, or base 10 as in decimal, hexadecimal uses base 16 which is still a power of 2. A compiler easily hexadecimal numbers, so using hexadecimal can make

code more readable. The following table shows the relationship between decimal, binary and their equivalent representation in binary:

Decimal	Binary	Hexadecimal
0	0	0
1	1	1
2	10	2
3	11	3
4	100	4
5	101	5
6	110	6
7	111	7
8	1000	8

9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

The hexadecimal system uses the letters A through F for digits above 9. So with one hexadecimal digit we can represent 4 binary digits, also called a nibble. But, how is a compiler to understand whether 10 is a decimal number, a hexadecimal number or a binary number? In the C language, which we will be using extensively, a prefix is used. To represent a hexadecimal number, 0x is placed before the number. For example, if we wanted to represent F in hexadecimal we will write 0xF.

It's important to note that leading zeroes don't change the actual value represented. However, the presence of leading zeroes does indicate the total range available. For example, 0xF indicates a total of 4 bits, while 0x0F indicates that a total of 8 bits are available, but the upper 4 bits are 0. We will see that setting a register to 0x0F or 0xF has the same result.

If you look at the code we used for the first application, you will notice the line:

```
P1DIR |= 0x01;
```

0x01 is a hexadecimal number. We are setting bit 0 (the lowest bit) high.

Bit Positions

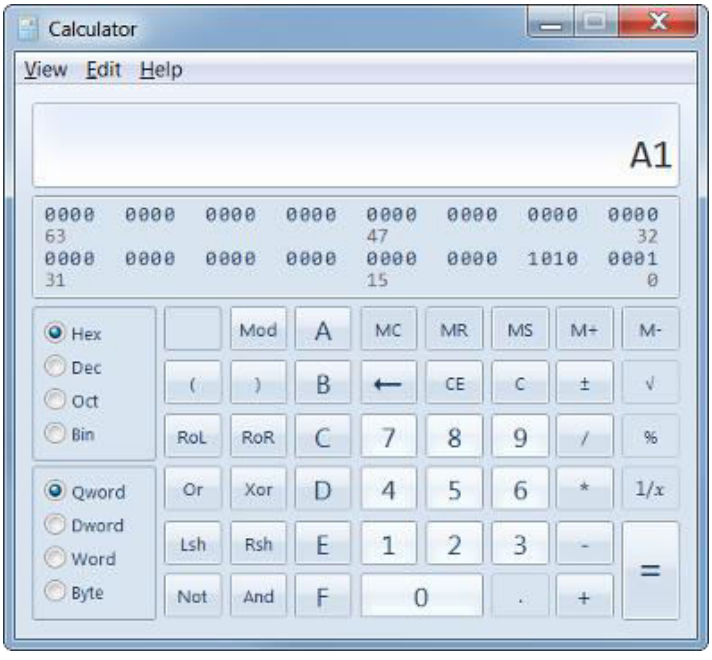
One very common operation done in microcontrollers is setting and clearing of bits. For example, we might want to set the 3rd bit. In binary this is represented as 100, or 0x04 in hexadecimal. The following table shows the relationship between bits and locations and their hex values. You might notice that as we cycle through individual bit locations we go through powers of 2, namely: 1, 2 4 and 8 (note that we begin counting from 0):

Bit Position	Binary	Hex Value
0	00000001	0x01
1	00000010	0x02
2	00000100	0x04
3	00001000	0x08
4	00010000	0x10
5	00100000	0x20
6	01000000	0x40
7	10000000	0x80

We will use these values all the time when programming the MSP430

Converting between numbers

At times you'll have to convert between the number systems for numbers like 0x5378 which are not easy to convert to decimal. Most operating systems provide a calculator capable of converting between decimal, binary, and hexadecimal. The windows calculator available in Windows XP and Windows 7 is one such application. Enter the calculator programmer mode and select the initial number system. For example, we want to know what the number 0xA1 is in decimal. So, select hexadecimal and type A1. Then click on the radio button called Dec to select the decimal number system.



Contact : 8828222688 / 8080097128

E-mail : info@embeddedtechnosolutions.com

Website : www.embeddedtechnosolutions.com

Our Branches: Thane, Nerul & Dadar

Note that due to 2's complement representation of numbers, one must be careful when dealing with numbers that are negative. In 2's complement, the uppermost bit indicates whether the number is negative. Positive numbers are not a problem, but in the calculator you need to select the right number of digits as indicated by the Byte, Word, Dword and Qword options. Let's assume we're working with signed 8-bit numbers. Then the uppermost bit (bit 7 if counting from 0) indicates whether the number is negative or not. 10000000 is -128. Had you selected Word (16-bit representation), the decimal representation would have been 128 because calculator would have treated bit 15 as the sign bit. Bit 7 would have been treated as a positive weight.

Sign Extension

The issue above with the sign brings up an interesting problem. Let's say we have a sensor that is 12-bit, meaning that the 11th bit (the uppermost bit counting from 11) is the sign bit. When interpreted by most software that works with 8, 16 and 32 bit numbers, any 12-bit number will be interpreted as positive, when in fact it might not be. To solve this, a number needs to be sign-extended. Sign Extension preserves the value of the number, but moves the interpretation of the sign bit correctly. For example, we can sign extend a 12-bit number into 16-bits. Some processors such as the x86 come with a dedicated instruction to perform sign extension, but

the MSP430 doesn't have that capability. Performing a sign extension is quite simple. If the sign bit for the 12-bit number is set, all bits to the left of that bit (up to the uppermost bit 15) should be set to 1.

Let's assume that we received a 10-bit value from a sensor 1111110001. Interpreting it as a 10-bit number shows this is the number -15. By setting all bits on the left of the leftmost one to one, we maintain the value. 1111111111110001 is the same number, just with all ones up to the last bit in 16-bit representation. Viewed in the Windows calculator in WORD mode you can see this number is still -15.

Data Representation

The C compiler requires a programmer to specify the data type to be used for every variable declared. These data types are processors specific, so it's important to understand the size of data types available. Signed types refer to data types where the uppermost bit is treated as a sign bit and indicates whether the number is negative or not. Unsigned types treat that bit as any other bit, so the number is always positive.

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

Choosing the right data type is critical to ensure operations such a addition, subtraction operate correctly since otherwise overflow will occur and results will be incorrect.

Digital and Bitwise Operations

Representing numbers is useless with doing something on them. Along with arithmetic operations such as addition and subtraction, AND, OR, NOT and XOR are operations you will be frequently performing while controlling microcontrollers because bitwise operations are common. If you are not familiar with these operations, they are covered next. We will be using them in the next section.

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

A	**B**	**A AND B**
0	0	0
0	1	0
1	0	0
1	1	1
A	**B**	**A XOR B**
0	0	0
0	1	1
1	0	1
1	1	0

A	A NOT
0	1
1	0
1	0
1	1

In the tables, A and B are one digit binary numbers with a value of either 0 or 1. The third column represents the value after performing the operation. OR, AND and XOR are binary operators because they require two operands (such as A and B). NOT is a unary operator because it requires only one operand (such as A).

You can think of OR as an operator of if either one is a '1', the result is '1'. AND on the other hand requires both operands to be '1' in order for the result to be '1'. XOR can be thought of as indicating difference. If A and B and different, then the result is '1'. Not is simply the opposite of the operand.

Bitwise Operations on Numbers, Variables, and Registers

We now go to actually make use of the operations above. We will be performing bit operations in the C programming language on variables. Both variables A and B below will be considered unsigned char with 8-bits. We will explore some common operations that depend on bits.

Lets assume that the microcontroller has detected some event and we want to store it. One easy way of doing this is to create a variable and set it to 0 or 1.

```
char flag = 0;
flag = 1;
```

But, this can be wasteful. We are only really using 1 bit out of 8. To improve memory usage (which is always a concern in microcontrollers), we can use bit operations to set, clear and check individual bits, so we are using all 8 bits for various purposes. Lets look at the same operator bit wise:

```
flags = 0; // A flag for each bit
.
.
.
```

```
flag |= 0x01; // Some event happened
.
.
.
if(flag & 0x01)
{
   // Do something
   flag &= (~0x01); // Clear flag
}
```

The code above is something embedded programmers do all the time. Inside an interrupt routine or function, a bit is set. Notice the operator |= performs a bitwise assignment to the variable flags, in this case it ORs flags with 0x01. This is a convenient shorthand to writing:

```
flag = flag | 0x01;
```

You can visualize this with bits as follows:

0 0 0 0 0 0 0 0

OR

0 0 0 0 0 0 0 1

= ————————

0 0 0 0 0 0 0 1

The operations above are performed bit by bit on each position. The result is that the variable flag has bit 0 set with a '1'.

At a later time when we are processing events, we might want to know whether the event happened, so we use the AND operator, indicated by &. When we perform the AND operation with the value 0x01, the result will be 1 only if bit 1 is set (the event happened). Otherwise the AND operation will return 0 and the if statement will not execute the code in the curly braces.

0 0 0 0 0 0 0 1

AND

0 0 0 0 0 0 0 1

= ————————

0 0 0 0 0 0 0 1

And if the event never happened so flags is 0x00:

0 0 0 0 0 0 0 0

AND

0 0 0 0 0 0 0 1

= ————————

0 0 0 0 0 0 0 0

The if statement will run the code only if the result of the AND is 1 or greater (if any of the bits are set).

If the event happened, we must clear the flag so that we can know when a new event occurs. Again we use the AND for self-assignment by ANDing flag with the inverse of 0x01.

X X X X X X X 1

AND

1 1 1 1 1 1 1 0

= ———————

X X X X X X X 0

Notice that we used X for the uppermost bits. We did this because it's important to note that the bit wise operation only affects in this case one bit. All the other bits are ANDed with '1' and remain the same. This operation is also referred to as bit masking. Using AND and OR on bits allows us to set, unset, and check for the presence of any bit. This is important when configuring the MSP430 registers, which we will cover soon.

```
0x02 & 0x02 // Result is 0x02
0x02 & 0x01 // Result is 0 because bit 0 is not set
0x02 & 0x03 // Result is 0x02 because bit 1 is the only one set
0x02 & 0xFF // Result is 0x02 because ANDing with 0xFF has no effect
0x02 & 0xFF // Result is 0x02 because ANDing with 0xFF has no effect
0x02 | 0x02 // Result is 0x02 because bit 1 already set
0x02 | 0x01 // Result is 0x03
```

Toggling with XOR

The XOR operator is also quite useful. As the different operator, it enables us to toggle bits.

0 0 0 0 0 0 0 1

XOR

0 0 0 0 0 0 0 1

= ————————

0 0 0 0 0 0 0 0

Notice that we performed an XOR of 0x01 with 0x01 and obtained 0x00. Bit 0 flipped. If we perform another XOR using 0x01:

0 0 0 0 0 0 0 0

XOR

0 0 0 0 0 0 0 1

= ————————

0 0 0 0 0 0 0 1

The bit has flipped again, despite the fact that we performed the exact same operation. XOR toggling comes in handy when we're toggling a blinking LED and we want to change the state from ON to OFF and vice-versa. If we want to perform this operation in C, we can do as follows:

```
char flag = 0;
flag ^= 0x01; // Results in setting flag to 0x01
flag ^= 0x01; // Results in setting flag to 0x00
```

Note that again we are using the shorthand notation to set flag to the result of the XOR.

Using Defines to improve code readability

The C language allows us to define a preprocessor replacement using the #define command. This allows us to write more human readable values. The values below are defined in msp430.h, which is a header file included in Code Composer Studio:

```
#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)
```

With these definitions we can perform the same bit operations without having to remember the hexadecimal value for the position:

```
char flag = 0;
flag ^= BIT0; // Results in setting flag to 0x01
flag ^= BIT0; // Results in setting flag to 0x00
```

Before compilation, the preprocessor goes through all files and performs the preprocessor commands such as #include, #define, etc.

ASCII

ASCII is a character encoding scheme. Glyphs such as letters and numbers are represented by an equivalent hex scheme, making it possible to encode strings in our application and communicate in a way that a human being can understand. The characters encoded include English characters, both uppercase and lowercase, as well as numbers, punctuation symbols and some control codes. Let's see an example:

```
ch = 0x41;
```

The code above sets the variable ch to the hex value of 0x41. Code Composer Studio and terminal applications can interpret this in ASCII and will see the uppercase letter 'A'. CCS also allows us to use ASCII directly, since it handles all the underlying representation values:

```
ch = 'A';
```

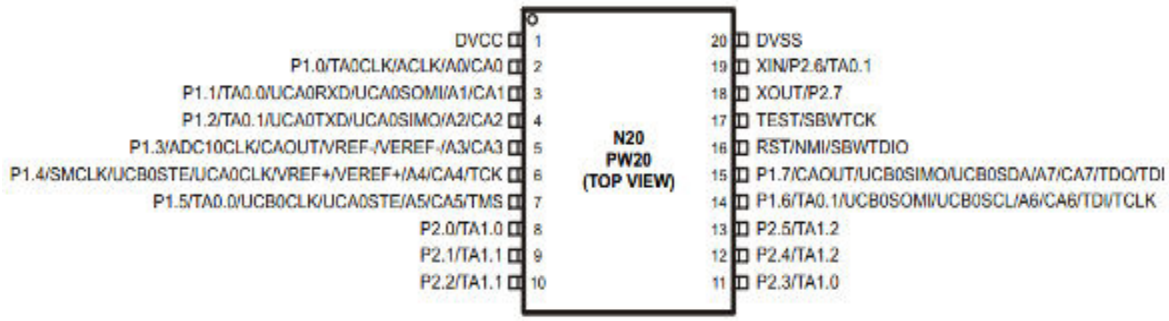
As you can see above, we can use single quotes to delineate single character value. When viewed in a memory editor, you will see 0x41 when seen as hex.

It's very important to differentiate the single character 'A' from the NULL terminated string "A" which cannot be stored in a single 8-bit character.

Chapter 4

General Purpose Input Output (GPIO)

GPIO stands for General Purpose Input Output and refers to the fact that the pins can support both output and input functionalities. Looking physically at any microcontroller you can readily see rows of pins that allow the microcontroller to control and communicate with outside devices. Lets take a look at the pinout diagram of the G2553 device.



Looking at the pins in the diagram we notice that each pin has a long name, with several designations separated by a slash. Because microcontrollers have a limited number of pins and

at the same time a large number of peripherals, the manufacturer has to multiplex the pins among the internal modules. This means that each pin has a number of functions that it can perform, but only one of them at a time. Most pins of the MSP430 operate as GPIO pins, with a possibility of functioning as a specialized pin in the right configuration. As GPIO pins, each pin is independently controlled and can be made an input and an output, high or low.

You might not realize it, but you have just stumbled upon one of the first thing every embedded designer must do. Because of the limited number of I/O pins and application requirements, you must carefully consider the use of each pin of the design. Pins are not born equal, and in some instances you will save yourself a lot of headache if you consider carefully what you need and what is available. You might even end up changing the microcontroller if the one you currently use doesn't provide the right mix of GPIO and peripheral pins.

Pins capable of GPIO functionality and control are indicated with the naming PX.Y, where X represents the port number to which the pin belongs and Y represents Pins belonging to port one are denoted by P1.Y, while pins forming Port 2 are named P2.Y where Y is a specific pin number. Notice that some pins are GPIO only, with no specialized function. The bit controlling the pin. The number of Ports and pins is device specific and can be obtained in the device datasheet. The MSP430F2553 has a total of 2 ports, Port 1 and Port 2.

Each port is assigned several 8-bit registers that control the function of the pins and provides information on their current status. The following is a list of registers always available for ports:

- PxSEL and PxSEL2 – These registers selects whether the pin operates in GPIO mode or is used for a specialized function as described in the pinout. PxSEL2 is not always available and is used to augment the number of multiplex options if the pin has various specialized functions. Setting PxSEL to 0 selects GPIO mode.
- PxDIR – If the pin is set to operate as GPIO, the bits in this register select whether a pin is a high impedance input (0) or an output(1).
- PxOUT – If the pin is set to operate as GPIO Output, this pin selects High (1) or Low (0) output.
- PxIN – If the pin is set to operate as GPIO Input, this pin indicates whether the voltage at the pin is High(0) or Low(0)

Note that each register bit controls only one pin in the port. For example, PxSEL looks as follows:

SEL P1.7	SEL P1.6	SEL P1.5	SEL P1.4	SEL P1.3	SEL P1.2	SEL P1.1	SEL P1.0

The second number in the port number (Y in PX.Y) tells you which bit controls it. P1.1 is controlled bit 1 in several registers including P1SEL, P1DIR, P1OUT and P1IN working all together to define the functionality of the pin.

Placing a pin in peripheral mode does not set the direction of the pin, which might be required depending on the peripheral. Make sure to consult the peripheral documentation in the Datasheet as to whether setting the direction is required and what is the proper direction.

Pull-Ups and Pull-Downs

Saving board space and reducing the number of components is the goal of every engineer. In many cases, GPIO lines need a pull-up or pull-down and integrating it in the MSP430 enables easy configuration and space saving. Some MSP430 support enabling a pull-up or pull-down resistors via software on certain ports. Note that this can only be enabled when the GPIO is in input mode. In order to control this, a few special registers are used:

- PxREN – Each bit enables (1) or disables (0) pull-up or pull-down resistors for the particular pin controlled by the bit
- PxOUT – When the pin is in input mode (as selected by PxDIR) and REN is enabled, this register selects whether the resistors is a pull-up (1) or pull-down (0)

PxREN only controls whether the functionality is enabled. PxOUT in this mode is the one that controls whether the resistor is pull-up or pull-down. A word of caution is in place. The pull-up and pull-down capability of the MSP430 is limited, with a resistance of about 20k to 50k ohm as shown in the datasheet. Using internal pull-ups in some cases such as I2C is not advisable, but it works well for switches and other cases.

Interrupt Capability

Some GPIOs in the MSP430 have the capability to generate an interrupt and inform the CPU when a transition has occurred. The MSP430 allows flexibility in configuring which GPIO will generate the interrupt, and on what edge (rising or falling). The registers controlling these options are as follows:

- PxIE – Each bit enables (1) or disables (0) the interrupt for that particular pin
- PxIES – Selects whether a pin will generate an interrupt on the rising-edge (0) or the falling-edge (1)
- PxIFG – Interrupt flag register indicating whether an interrupt has occurred on a particular pin (if it experienced the transition)

PxIES might be a little confusing. One easy way to remember the option is to think of the bit as the initial state. For example, if the bit is 0, the initial state is at 0, and the pin will generate an interrupt going from 0 to 1, a rising edge.

MSP430 devices typically have interrupt capability on Ports 1 and 2. Other ports may not have any capability for interrupts. You must therefore be careful when designing the MSP430 on a board to ensure that the pins are properly allocated so you have interrupt capability.

Electrical Specifications

In the discussion above we always talked about '1' and '0'. The reality is that a microcontroller generates a finite voltage representing '1' and '0'. Except for a few MSP430 with dual voltage rail capability, the HIGH (1) generated by the GPIO will be close to VCC. This assumes that the load on the GPIO is within reason. The MSP430 like most devices has a limit on the amount of current that can be drawn from the pins, both individually and as a whole. The datasheet of the MSP430 specifies that with 6mA of current draw at a pin the following should hold:

- VOH – Output Voltage High Level VCC-0.3V
- VOL – Output Voltage Low Level VCC+0.3V

The ramifications are significant. Running the MSP430 at 3.3V will make it difficult to interface to a 1.8V part because the high voltage will likely violate the specifications of the other device.

You could run the MSP430 at 1.8V, but there are other considerations such as the frequency dependency of the CPU on voltage. The main point is that you should consider the issues of GPIO loading and voltage in your design because they might be significant.

The MSP430 also requires that applied voltages to pins in input mode be within specification. The MSP430 uses a Schmitt-Trigger at the input of each pin to determine whether the voltage represents a logic High(1) or Low(0). The datasheet specifies what to expect as far as logic thresholds. In general, voltage on the pins should never exceed VCC that is operating the MSP430, and never 3.6V. Although it is possible to operate pins at VCC+0.3V, this triggers the ESD diode structure and may damage the part.

Power up Defaults

Upon powerup of the MSP430, before any code is executed, the registers controlling the MSP430 are cleared to defaults. This includes PxSEL set to 0, along with PxDIR, meaning that pins are generally configured as inputs with high impedance. In this mode, the pins draw almost no current and thus avoid disturbing the circuit. Although example code sometimes avoid explicitly setting all the configuration registers, we suggest that you be always explicit and never rely on any defaults. One reason is that a custom bootloader may be used that takes

control of the GPIO for some purpose and cause problems for the application which fails to initialize all the registers.

Initializing GPIO

We can't discuss GPIO without showing how to control it. The following examples show how to configure the GPIO as well as use the pin muxing:

Configuring P1.0 as a GPIO Output set to High(1)

```
#include <msp430.h>


void main()
{
   WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer
   P1SEL &= (~BIT0); // Set P1.0 SEL for GPIO
   P1DIR |= BIT0; // Set P1.0 as Output
   P1OUT |= BIT0; // Set P1.0 HIGH
}
```

Configuring P1.0 TA0CLK

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

```
void main()
{
   WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer
   P1SEL |= (BIT0); // Set P1.0 SEL as TA0CLK
}
```

Configuring P2.1 for Interrupts

```
#include &<msp430.h>

void main()
{
   WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer
   P2SEL &= (~BIT1); // Set P2.1 SEL as GPIO
   P2DIR &= (~BIT1); // Set P2.1 SEL as Input
   P2IES |= (BIT1); // Falling Edge 1 -> 0
   P2IFG &= (~BIT1); // Clear interrupt flag for P2.1
   while(1)
   {
```


```
    if(P2IFG & BIT1)
    {
    // Do Something
    }
  }
}
```

The code above shows how it is possible to poll the Interrupt Flag register to check whether an interrupt has occurred, as opposed to having an interrupt routine called. Polling should be generally avoided because of the performance penalty of having the CPU constantly check for the state.

```
#include <msp430.h>
volatile int flag = 0;

void main()
{
  WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer
  P2SEL &= (~BIT1); // Set P2.1 SEL as GPIO
  P2DIR &= (~BIT1); // Set P2.1 SEL as Input
```

```
  P2IES |= (BIT1); // Falling Edge 1 -> 0
  P2IFG &= (~BIT1); // Clear interrupt flag for P2.1
  P2IE |= (BIT1); // Enable interrupt for P2.1
  __enable_interrupt(); // Enable Global Interrupts
  while(1)
  {
    if(flag == 1)
    {
      // Do Something
      flag = 0;
    }
  }

}

// Port 1 interrupt service routine
#pragma vector=PORT2_VECTOR
__interrupt void Port_2(void)
```


```
{
   flag = 1;
   P1IFG &= (~BIT1); // P2.1 IFG clear
}
```

The application above makes complete use of interrupts. P2.1 is configured as an input pin with a falling edge interrupt. General interrupts are enabled so that the pin will make the CPU call the Interrupt Service routine. This routine sets a flag that causes some processing to be performed in the main loop.

Switches

One of the most common uses for GPIOs is to connect switches, especially momentary buttons for user interfaces. Switches can be connected easily, but require some consideration to use properly. The switch is typically connected between a pin and ground, so that pressing on the switch will cause a falling edge. For this to happen, the pin must be biased at VCC. Typically this means using a pull-up, either internal or external. For external switches, a value of 100k is typical. When the switch is open, the voltage at the pin is basically VCC given the small amount of current flowing results in virtually no voltage drop across the pull-up resistor. When the

switch is closed, the pin is connected to ground and the pull-up resistor conducts as well. This current is wasted, which is why large values of pull-up resistors are used.

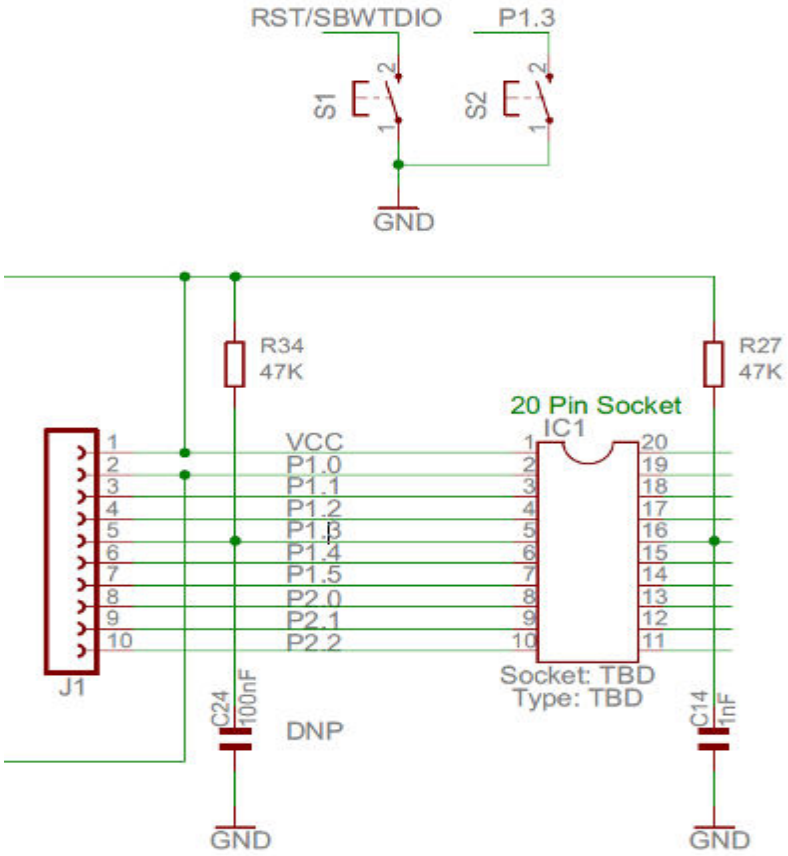
Up to now we have assumed that the switch is a perfect device with a well defined ON and OFF behavior. In reality, most switches are mechanical and are prone to bouncing. The contacts that are being connected together will take some time to settle. During this time, the electrical circuit will connect and disconnect, causing transients to appear at the pin. These transients are often in microseconds, which means that a microcontroller may see hundreds of transitions for just one press of the button. The first solution to this issue is to use a capacitor placed parallel to the switch to debounce it. When the switch is normally open, the capacitor is charged to VCC. When the user presses the switch, the capacitor begins discharging to ground, but does not do so immediately. During this time, the voltage drops off exponentially and only a single transient is seen by the pin. The speed of the transition depends on the RC constant of the pull-up and capacitor, but values of 10nF to 100nF and 100k typically work well.

Along with a hardware solution, it is possible to use a software timer to perform a software debounce. On the first interrupt you can trigger a timer and disable the GPIO interrupt. After a set amount of time, usually 20ms to 50ms, the timer triggers and its ISR checks whether the switch is still being held (the pin will show low on PxIN). If so, then a button press is handled.

Experiments have shown that different buttons behave differently, although 20ms to 50ms is typically the time required for them to completely settle down.

We recommend that you test the buttons in your hardware to see the behaviour of the bouncing. This can be done easily with an oscilloscope connected to the pin.

Lets look at how the MSP430 launchpad designs the switch:

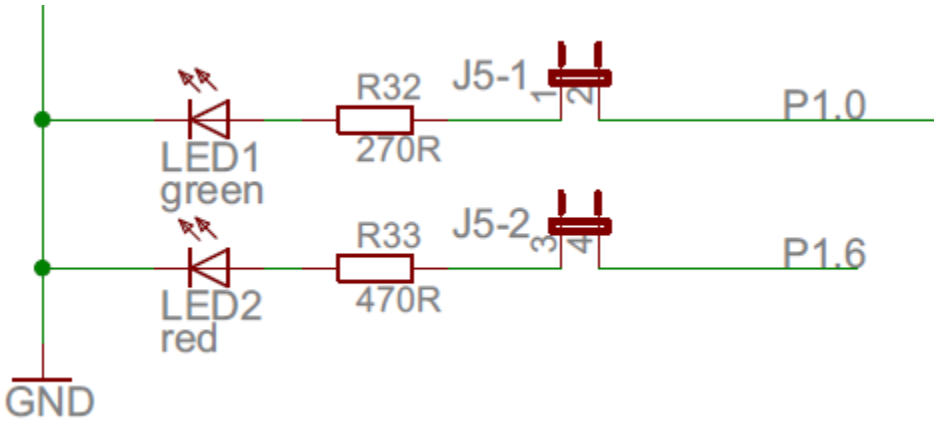


S2 is the switch of interest and it is connected to P1.3. P1.3 has both a 100nF capacitor and 47k ohm pull-up resistor, as was our recommendation above. This results in clean signals at the pin.

LEDs

Another easy use of GPIOs is for indication using LEDs. An LED can be connected directly to the MSP430 pin. One easy way is to connect the LED as follows:

When the pin is low, the voltage across the LED is almost zero, so it remains off. But, when the pin is high, the voltage causes the LED to turn on. Note that a series resistor is used to reduce the voltage given that most LEDs require around 2V (consult the datasheet for your LED). Values of 220 ohm to 1k ohm are typical. A microcontroller is typically better at sinking current than sourcing it. LEDs can be connected between VCC and the MSP430: The polarities are then reversed. When the MSP430 pin is high, the LED is off because the potential difference between the LED terminals is very small. When the MSP430 pin is set to '0' (Ground), the potential difference is large and the LED turns on.



You can see the Launchpad board has two LEDs at P1.0 and P1.6. Each LED has a series resistor with a different value to make the output luminosity more uniform when they are lit.

Chapter 5

MSP430 Clock

Clocks are at the heart of synchronous digital systems and processors. CPUs require clocks to run to ensure the orderly fetching and execution of instructions. 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 you to specify what clocks will be used and at what speeds. This means you have to understand your application and what clocks are needed to satisfy your requirements.

One of the most obvious effects of clocks is the speed at which the CPU is processing data. If the clock is slow, algorithms will take longer to complete and the application can violate timing requirements. Another good example is the timing required for UART. UART is covered in a later chapter, but being asynchronous means that the reliability of the data being transmitted depends on the input clock, as is the speed at which it transmits data. If the clock is unstable and the baudrate is inaccurate, errors will be detected at the receiver.

The MSP430 clock system is very flexible and is built both to enable a fast CPU while at the same time supporting slow clocks that allow low power. With low power being the cornerstone

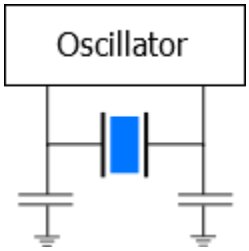
of the MSP430, reducing the frequency of the CPU is critical to reducing power consumption. The dynamic power consumption can be shown to be: 1/2 C*V^2*f where V is the voltage, C the switching capacitance and f is the frequency of the circuit. Reducing the voltage and frequency can provide tremendous reductions in power consumption which makes products last longer on batteries.

Clock Generation

As with any system, clock generation contains many tradeoffs between size, accuracy and cost. More expensive solutions typically tend to produce better and cleaner clock signals that have lower jitter and drift. In the MSP430, the two main clock generation mechanisms are internal RC type oscillators and internal oscillators using external crystals.

The MSP430 can contain several internal oscillators. Of particular note are the Digital Controlled Oscillator and the VLO low frequency oscillator. Both of these are based on an RC network. RC networks can be rather inaccurate in generating precise clocks, so TI has trimmed and calibrated them to reach close to 1% accuracy. These type of oscillators are heavily affected by temperature and cannot be always be used for interfaces requiring accurate timing such as UART without causing errors. The DCO is digitally controlled because its frequency be changed from several hundred kHz up to 25MHz.

The MSP430 can also use external crystals with internal oscillator circuitry to generate both low frequency (32kHz) and high frequency (Up to 25MHz) clocks that are as accurate as the crystal used. One issue with crystals is that they are not tunable, so the only possible option is to divide them, but they are far more accurate, reaching 50ppm accuracy or better. A downside of using crystals is that they occupy more space on the board and increase the BOM cost, but when doing UART or needing accurate timing measurement they are practically obligatory.



Crystals are accurate and they have low error due to aging. In general, every crystal is used in a parallel configuration, which is the case in the MSP430 and many other microcontrollers,. This requires capacitors to provide the load capacitance needed for the oscillator to oscillate at the right frequency. These capacitors are usually connected to the two crystal pins as shown in the figure above. 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). Any change in the effective loading changes the crystal's effective frequency, which is an error in

itself. However, it isn't only the capacitors that add capacitance. The pins of the microcontroller as well as traces between the microcontroller and the crystal and the internal circuitry of the MSP430 oscillator also act as capacitors. To have an accurate frequency, a designer needs to take all of these elements into account. We will go through a simple exercise to demonstrate how to calculate the capacitors we should use.

Let's assume we need a 4MHz crystal and have selected one with 7pF of loading capacitance. C1 and C2, the loading capacitors, are usually selected to be the same, so we have:

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

Because both capacitors are the same, C1 = C2, we get 14pF of capacitance. Therefore, two capacitors of 14pF will provide us with the required load. This is quite simplistic and doesn't take into account the other parasitics. The oscillator inside the microcontroller introduces two capacitances generally called CXIN and CXOUT. Each appears in parallel with one of the external loading capacitors. In addition to this, we also have the parasitics due to the PCB itself which are in parallel to the other capacitances.

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

Cpar is not always easy to measure. Because of this we typically assume a value of 2-5pF which works in most cases.

In some cases, external loading capacitors are actually not required. In the MSP430, several devices incorporate internal loading capacitors for the Low Frequency 32kHz watch crystals. The MSP430G2553 has 4 options for integrated effective load capacitance for low frequencies. You can choose between 1pF, 5.5pF, 8.5pF and 11pF, and these values also include the parasitic bond and package capacitance, which are around 2pF per pin. Note that you can still integrate external capacitors to try and compensate for parasitics, but this often causes more problems because the capacitors themselves have parasitics.

Whether you are using the integrated oscillators or external crystals, it is possible and advisable to measure the output frequency after a buffer. The MSP430 enables several clock sources to be output to a pin so the frequency can be measure using an oscilloscope or more accurately frequency counter.

We have discussed the clocks available, but typically they fall into the following:

- LFXT1CLK – Low frequency mode of the XT1 oscillator, typically used with watch crystals or clocks of 32.768kHz.

- HFXT1 – High frequency mode of the XT1 oscillator, used with High Frequency crystals. Not available in all MSP430 (not present in G2553 device)
- VLOCLK – Internal low frequency and low accuracy oscillator for low power
- DCOCLK – Internal Digitally Controlled Oscillator capable of generating a wide array of frequencies.
- XT2CLK – Optional high-frequency oscillator that uses standard crystals, resonators or external clocks in the 400kHz to 16MHz range or more.
- PxIN – If the pin is set to operate as GPIO Input, this pin indicates whether the voltage at the pin is High(0) or Low(0)

The exact clocks available on an MSP430 varies from device to device and between families. The datasheet and the User's Guide are the best sources to see what is available the frequency information as well as accuracy specifications.

Clock Tree Signals

The clock tree in the MSP430 is actually composed of several signals whose source is selectable. This flexibility helps power CPU and peripherals using different clocks to achieve either speed or low power.

The main signals in the MSP430 include

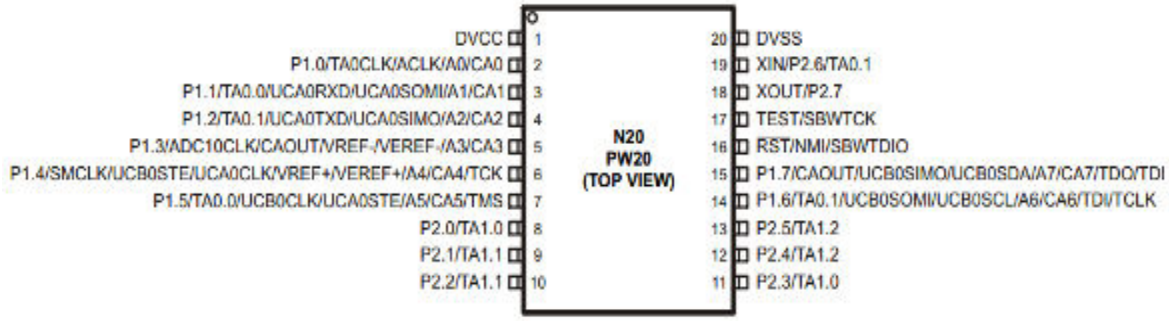
- ACLK: Auxiliary clock. ACLK is software selectable as LFXT1CLK or VLOCLK. ACLK is divided by 1, 2, 4, or 8. ACLK is software selectable for individual peripheral modules.
- MCLK: Master clock. MCLK is software selectable as LFXT1CLK, VLOCLK, XT2CLK (if available on-chip), or DCOCLK. MCLK is divided 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 of the MSP430 runs only from MCLK, while other peripheral modules can be sourced by any of the clock signals or in some cases from other clock sources brought in on specific pins.

Clocks on the MSP430 Launchpad

The Launchpad typically uses a MSP430G2553 device that contains a Basic Clock Module+. Only XT2 is unavailable on the device due to the lack of pins. XT1 supports both low frequency and high frequency crystals, and it integrates both DCOCLK and VLOCLK. After power-up, MCLK and SMCLK are sourced from DCOCLK at about 1.1MHz. ACLK is sourced from LFXT1CLK in LF mode.

The Launchpad comes with an option 32kHz crystal in the box that needs to be installed by the user.



Pin 18 and Pin 19 are XIN and XOUT and allow an external crystal to be connected, either 32kHz or a fast crystal. Given the small number of pins, this is all that's available on this device.

Internal DCO

Upon reset, the MSP430 does not know whether any external crystals are available, and what they are, so it defaults to using the internal DCO to power MCLK and other clocks. We mentioned before that the MSP430 will start up around 1.2MHz for the G2553. Let's see how we can change the frequency to speed up the MSP430. TI has taken each MSP430 and

performed a calibration of the DCO versus a known input frequency for several common frequencies. These calibration values provide DCO steps that result in a clock value as clock as possible to the frequency. The values themselves have been stored by TI on the Info flash segment, and care should be taken not to erase this segment of the flash.

Checking for DCO Calibration values in Flash

```
#include <msp430.h>
void main()
{
  WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer

  /* Check if 1MHz Calibration is present */
  if (CALBC1_1MHZ != 0xFF)
  {
    DCOCTL = 0; // Select lowest DCOx and MODx
    BCSCTL1 = CALBC1_1MHZ; // Set range
    DCOCTL = CALDCO_1MHZ; // Set DCO step + modulation
  }
```

```
/* Check if 8MHz Calibration is present */
if (CALBC1_8MHZ != 0xFF)
{
   DCOCTL = 0; // Select lowest DCOx and MODx
   BCSCTL1 = CALBC1_8MHZ; // Set range
   DCOCTL = CALDCO_8MHZ; // Set DCO step + modulation
}

/* Check if 12MHz Calibration is present */
if (CALBC1_12MHZ != 0xFF)
{
   DCOCTL = 0; // Select lowest DCOx and MODx
   BCSCTL1 = CALBC1_12MHZ; // Set range
   DCOCTL = CALDCO_12MHZ; // Set DCO step + modulation
}

/* Check if 16MHz Calibration is present */
```

```
  if (CALBC1_16MHZ != 0xFF)
  {
    DCOCTL = 0; // Select lowest DCOx and MODx
    BCSCTL1 = CALBC1_16MHZ; // Set range
    DCOCTL = CALDCO_16MHZ; // Set DCO step + modulation
  }
}
```

The code above is more of an example than real code. It goes through every DCO calibration setting and actually sets the DCO to that value. Typically you would just pick one value to run or have a more sophisticated approach. It's very important that you check whether the calibration value is valid. If the Info flash with the values was erased, its value would be 0xFF. Let's try out the speed of the DCO. The code below will blink the LED of the Launchpad using the default DCO speed, and then using the DCO at 16MHz to show that the speed is actually changing, and what the effect is.

Configuring MSP430 DCO to 8MHz

```
void main()
{
  WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer
  P1SEL &= (~BIT0);
  P1OUT &= (~BIT0);
  P1DIR |= (BIT0);

  for(blink_count = 0; blink_count < 10; blink_count++)
  {
    P1OUT ^= (BIT0); // Toggle LED
    for(i = 0; i < 20000; i++);
  }

  /* Check if 8MHz Calibration is present */
  if (CALBC1_8MHZ != 0xFF)
  {
    DCOCTL = 0; // Select lowest DCOx and MODx
```

```
    BCSCTL1 = CALBC1_8MHZ; // Set range
    DCOCTL = CALDCO_8MHZ; // Set DCO step + modulation
  }

  for(blink_count = 0; blink_count < 10; blink_count++)
  {
    P1OUT ^= (BIT0); // Toggle LED
    for(i = 0; i < 20000; i++);
  }
}
```

After blinking a few times slowly, the LED will blink rapidly and then turn off.

External 32kHz Clock

The following code makes P2.6 and P2.7 set for accepting a crystal at the input:

MSP430 Using External 32kHz crystal

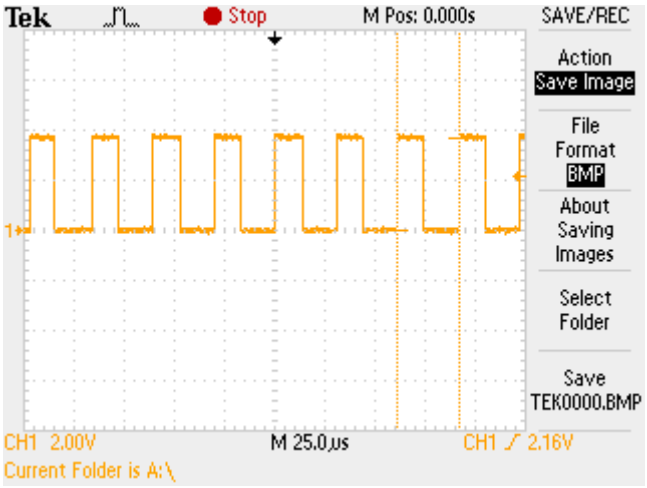
```
#include <msp430.h>
void main()
{
```

```
  WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer
  P2SEL |= (BIT6 | BIT7); // Set P2.6 and P2.6 SEL for XIN, XOUT
  P2SEL2 &= ~(BIT6|BIT7); // Set P2.6 and P2.7 SEL2 for XIN, XOUT

  /* Select 32kHz Crystal for ACLK */
  BCSCTL1 &= (~XTS); // ACLK = LFXT1CLK
  BCSCTL3 &= ~(BIT4|BIT5); // 32768Hz crystal on LFXT1

  /* Output buffered ACLK on P1.0 */
  P1SEL |= BIT0;
  P1SEL2 &= ~(BIT0);
  P1DIR |= BIT0;
}
```

If you have installed the 32kHz crystal that comes with the MSP430 Launchpad, you will see the following output as measured on P1.0. Make sure to remove the Jumpers for the LEDs because the presence of an LED on P1.0 will create problems with the output of ACLK.



Chapter 6

MSP430 Interrupts & Low Power

Using interrupts makes your MSP430 respond faster and allows the CPU to spend more time processing, as well as take advantage of low power

Basics of Interrupts

If you don't have experience programming microcontrollers, interrupts might seem like a thing of the past when IRQ switches needed to be set for cards installed in a computer. However, interrupts are still happening at the lower levels, managing DMA and other events. In microcontrollers such as the MSP430, interrupts play a key role in enabling fast response, scalability and detection of rare events. Using interrupts allows the MSP430 can detect a button press, or the arrival of a packet from a transceiver, oscillator faults and other exceptions.

Interrupts are generally any trigger that causes the CPU to deviate from executing instructions in the order set by the instructions. When an interrupt occurs, the CPU of the MSP430 saves its current state and goes to handle the interrupt handler if one exists. As we alluded to previously, interrupts can be both exception fault interrupts, which indicate an error has occurred in the

MSP430, or more event interrupts such as the GPIO interrupt triggering. Given the vast number of interrupts, the MSP430 prioritizes them. Exception interrupts are typically the highest priority because they require immediate attention. Moreover, interrupts are designed hierarchically. A peripheral module may generate only one interrupt, and the interrupt service routine will need to check the actual source of the interrupt. Even though interrupts are individually set in modules, there is a Global Interrupt Enable (GIE) bit that can disable the vast majority of them at once. We “mask” the interrupts when we prevent them from triggering their handler in the CPU. The interrupts may occur, but they are masked (hidden) from the CPU and will not cause their handler to run. One way of categorizing interrupts is by the effect they have on the system. The MSP430 generally categorizes interrupts as follows:

- System Reset Interrupts – When triggered, these interrupts cause a reset of the system
- Non Maskable Interrupts – These interrupts cannot be masked. Typically these are fault handlers such as oscillator faults and flash access violation which indicate a critical condition
- Maskable Interrupts – Most interrupts on the MSP430 fall into this category. GPIO, timers and peripherals all generate interrupts that can be masked by the GIE bit.

System Reset interrupts have no interrupt service routine. If you remember the JTAG pinout you might remember the RST/NMI pin. This pin is by default the MSP430's reset pin and it generates a complete reset of the MSP430.

All Non-Maskable Interrupts share the same NMI interrupt service routine. When configured in NMI mode, the RST/NMI pin will trigger the NMI interrupt handlers. Flash access violation is another non-maskable interrupt source that must be specifically enabled by the ACCVIE bit. Oscillator fault is similar in that it also needs to be enabled specifically to detect when an oscillator has failed. In the NMI ISR you must specifically check for the flags that caused the condition, reset them, and handle them accordingly.

As we said, most interrupts generated by peripherals such as GPIO and timers fall into the maskable interrupt category in that they can be masked from the CPU by disabling the General Interrupt Enable (GIE) bit. When using the MSP430, disabling all interrupts can often result in missing important event such as receiving characters. It's usually preferable to write the application in such a way as to make interrupts enabled as much as possible. Enabling and disabling the GIE can be done via several calls, the most common shown below:

Functions to control global MSP430 interrupts

```
__enable_interrupt(); // Enable Global Interrupts by GIE = 1
__disable_interrupt(); // Disable Global Interrupts by GIE = 0
```

In order for an ISR to trigger, all that's required is that the interrupt for the module be enabled, GIE is set and the condition for the interrupt is met. Note that this can even happen inside ISRs if the GIE is enabled, causing a possibly dangerous condition called nested interrupts which we will discuss later.

Interrupt Priorities

It is possible for multiple interrupts to occur at the same time. Which interrupt will the MSP430 handle first? The interrupts in the MSP430 and all microcontrollers have priorities. In the case of the MSP430 these are fixed and you must take care that your application does not depend on an order that is contrary to the priorities. For example, Timer_A and Timer_B have different priorities, and it might be necessary to choose one or the other when doing the hardware design or software implementation. The priorities table is quite large and we won't reproduce it here.The table itself is generic for many of the priorities. It requires the datasheet to specify the device-specific interrupt source. When multiple interrupts occur, the highest priority interrupt will be handled first. When the interrupt service routine is finished, if any other interrupts are pending, the highest of them will be handled, and so forth until all the interrupts are managed.

Anatomy of Handling an Interrupt

An interrupt is not very different than a conversation with two people. Let's say you're talking to a person and suddenly someone talks to you. If it's important, you might momentarily halt your conversation with the first person and turn your attention to the second person. Once you are done, you will go back to the first person and recall the last topic you were discussing. The MSP430 and all microcontrollers do the same thing, although it is done programmatically and behind the scenes.

We know that the MSP430 checks for the interrupt and selects the highest priority that is not masked. But how does the MSP430 remember the "conversation"? The current state of the MSP430 is composed of several things. All of these need to be stored before we go handle an interrupt so we can return successfully:

- The Program Counter (PC) holds the address of the next instruction to be executed
- The Status Register (SR) holds the status of CPU results, interrupts, and operation

So, saving these two items allows us to go back because we're saving the address of the instruction we would have executed had we not received an interrupt, as well as the status of the CPU and the system. Where do we save this? The stack on the MSP430 is where we will place the PC and SR (called pushing on the stack), both of which cause the Stack Pointer register

to increment accordingly. When we are finished, we will “pop” the Status Register and the Program Counter from the stack to their respective registers and we can continue from our last location.

Once the interrupt with the highest priority is selected, the interrupt flag for single-source flags are reset. Multi-source flags must be explicitly cleared. The current Status Register (not the one pushed to the stack) is cleared. Because the SR controls the low power modes including whether the CPU is currently turned on, any low power mode is therefore terminated (but since we stored the Status register in the stack and it will be restored when we’re done, we will return to the Low Power mode when the interrupt handling is complete). For the CPU to process the ISR, the content of the interrupt vector is loaded to the Program Counter.

This last statement is important to note. The interrupt table contains addresses to the actual interrupt handlers. When the ISR is handled, it is this address that is loaded to the program counter. The address can vary from compile to compile and is a used specific code. Once this is done, the CPU begins executing the instructions that form part of the interrupt routine.

Once the interrupt is completed with the RETI (Return from Interrupt) instruction, the SR and the PC are popped from the stack and we begin executing the instruction pointed by the stack register. Before exiting from an interrupt it is possible to modify the Status Register that will be

popped. This allows us, for example, to enter or exit low power modes, and even to disable the GIE using the following:

Functions to control MSP430 Low Power Modes

```
__bis_SR_register_on_exit(x) // Disable Global Interrupts by GIE = 0
__bic_SR_register_on_exit(x) // Enable Global Interrupts by GIE = 1
```

The functions above set and clear bits of the status register that was pushed to the stack in the stack itself and should only be used in interrupt routines. There are other non-ISR versions of the intrinsics above. These two functions require parameters for the bits to be changed. These bits are conveniently available as defines by Code Composer Studio:

MSP430 Low Power Modes Bits

```
LPM0_bits
LPM1_bits
LPM2_bits
LPM3_bits
LPM4_bits
GIE
```

We can call the intrinsics as follows:

```
__bic_SR_register_on_exit(LPM0_bits | GIE); // Exits LPM0 and disables GIE upon exit
```

Multiple Interrupt Sources

Some modules of the MSP430 provide only one interrupt source to the CPU, despite internally supporting multiple sources. An example of this are the GPIO Ports. A port supporting interrupts will have 8 interrupt sources. To avoid connecting such a large set of interrupts to the CPU, each port only provides one interrupt source and all interrupts for the port share the same Interrupt Service Routine. Once in the interrupt routine, however, we must check to see which bit actually caused the interrupt. For example, if the interrupt was caused by Port 1, we can use P1IFG and manually check for each bit.

Some MSP430 devices contain an Interrupt Vector (IV) register such as P1IV and RTCIV that make it easy to handle the interrupts. This registers contains a number representing the highest priority interrupt present on that port. Using this IV register and the intrinsic **__even_in_range()** we can handle interrupts as follows:

We can call the intrinsic as follows:

```
// Won't run on MSP430G2553
#include <msp430.h>
volatile int flag = 0;

void main()
{
   WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer
   P1SEL &= (~BIT3); // Set P1.3 SEL as GPIO
   P1DIR &= (~BIT3); // Set P1.3 SEL as Input
   P1IES |= (BIT3); // Falling Edge
   P1IFG &= (~BIT3); // Clear interrupt flag for P1.3
   P1IE |= (BIT3); // Enable interrupt for P1.3
   __enable_interrupt(); // Enable Global Interrupts
   while(1)
   {
      if(flag == 1)
      {
```


```
        // Do Something
        flag = 0;
      }
   }
}


// Port 1 interrupt service routine
#pragma vector=PORT1_VECTOR
__interrupt void Port_1(void)
{
   switch(__even_in_range(P1IV,16))
   {
   case 0: break; // No Interrupt
   case 2: break; // P1.0
   case 4: break; // P1.1
   case 6: break; // P1.2
   case 8: // P1.3
   flag = 1;
```

```
  break;
  case 10: break; // P1.4
  case 12: break; // P1.5
  case 14: break; // P1.6
  case 16: break; // P1.7
  }
}
```

Each time we access P2IV, we get the number of the highest priority interrupt. Using **__even_in_range()** isn't just convenient. This intrinsic is information to the compiler that the number provided by P2IV is always even and goes up to 16. This enables the compiler to make smart decisions to optimize the service routine. In this case, the compiler instead of checking bit will add the value of P2IV to the program counter, to naturally jump to the handler. This results in faster processing of some of the interrupts as it requires fewer instructions.

This code will not run directly on the G2553 device because it lacks P1IV and P2IV.

Performance Implications

Whether you have realized it by now, saving the current state, handling an interrupt and then restoring the state takes time. In systems with very tight deadlines and many interrupts, doing his frequently for multiple interrupts can take a significant amount of time and interfere with other interrupts, especially given that typically interrupts are ignored during interrupt handling proper. The CPU is basically wasting time just saving its state, a costly overhead when done often.

The mantra of keeping interrupts short has been drilled into embedded engineers for as long as they have existed, yet still there are many who use ISRs as a kitchen sink and even place loops inside interrupts. Never do this. A fast interrupt is the best kind. Using a flag set at an interrupt and processing it in the main loop is almost always the best choice.

MSP430 Low Power Modes

The MSP430 is designed from the ground up for low power. This includes both design and process implementation. Despite this, the bulk of power savings is realized by placing the MSP430 in various power saving modes. We first have to understand what consumes the most current in the MSP430. This breaks down as follows:

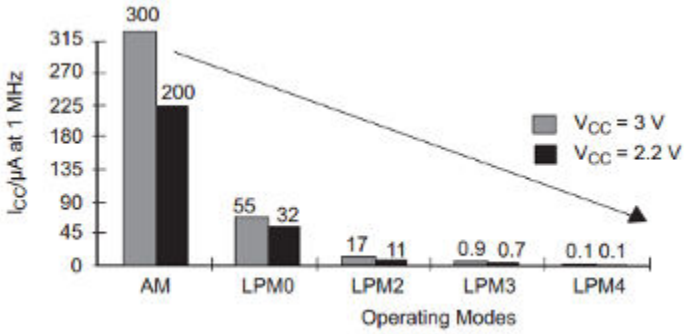
- MSP430 CPU draws the most, proportional to the frequency at which it is running.
- Clocks and Oscillators, especially high speed clocks.
- Modules and Peripherals

Saving power comes down to shutting off as many modules, peripherals, and clocks as we can, for as long as possible without violating the application time requirements. It is also important to reduce the speed of the CPU as it can significantly affect power consumption. The trade-off is that processing will take longer. MSP430 peripherals are also designed for low power. For example, the ADC will shut off automatically after the conversions are finished. The issue isn't always what to turn off but when, and when to wake them up. Interrupts play a central role because they enable the MSP430 to go to a deep sleep and wake up if events have occurred, either external as detected by GPIOs or internally generated by the peripherals. To enable low power the MSP430 supports several modes, each shutting off the MSP430 more and more:

- Active Mode (AM) – Not a low power mode but rather the mode in which everything is turned on, except perhaps for some peripherals
- LPM0 – CPU and MCLK are shutoff. SMCLK and ACLK remain active.
- LPM1 – CPU and MCLK are off, as in LPM1, but DCO and DC generator are disabled if the DCO is not used for SMCLK. ACLK is active.

- LPM2 – CPU, MCLK, SMCLK and DCO are disabled, while DC generator is still enabled. ACLK is active.
- LPM3 – CPU, MCLK, SMCLK, DCO and DC generator are disabled. ACLK is active.
- LPM4 – CPU and all clocks disabled

These LPM modes refer to the individual bits in the Status Register. By setting and clearing the bits in the SR, one can turn off CPU and clocks resulting in certain Low Power Modes. Let's look at the power consumption profile of the various LPMs:



At 1MHz, we go from 300uA down to less than 1uA by switching to LPM3. This is what makes the MSP430 such a good microcontroller for low power applications. It can survive for years on batteries. But, to take advantage of this, we have to build an application that takes advantage of these low power modes and turns on only what's necessary when it's necessary.

In general, using LPM modes comes down to the following:

- If your application can wait until a GPIO causes an interrupt and you need no clocks active, use LPM4
- If you need ACLK at 32kHz with a peripheral such as a timer, use LPM3
- Use LPM0 as much as possible – basically top the CPU from processing if not needed
- Avoid polling. Use interrupts for all low level drivers for UART, SPI and I2C as well as other modules as much as possible. Build your application to operate asynchronously.
- Avoid using the CPU to do anything that can be done with the peripherals, including timing and other operations.

Aside from the LPM modes above, some MSP430 devices include LPM3.5 and LPM4.5 which disable the Power Management Module. The MSP430G2553 does not include a PMM and therefore does not support these modes. When LPMx.5 mode is entered, all RAM and register contents are lost, although I/O states are locked and will not change.

Low power modes allow us to conserve energy, but we do pay a penalty for using them. That penalty is the time it takes to go from a low power mode to Active Mode. The deeper we go into low power modes, the longer it takes to go to Active Mode. The primary reason is that oscillators take time to come to a stable state. In some applications, the delay in getting to

active mode to handle an interrupt is too long and precludes the programmer from making the MSP430 go to sleep. Slower clocks such as 32kHz oscillator take much longer to stabilize (10ms to 100ms or more in some cases), while the fast clocks and crystal may take only a few microseconds. For this reason, it is preferable to wake up with a very fast clock, which is the default behavior.

Chapter 7

MSP430 UART Peripheral

Physical Interface

From a pinout perspective, UART signals only require one line for unidirectional communications, although two are typically used for bi-directional transmit and receive. Being asynchronous, UART signals don't require any other clock line because the two UART devices agree on a common baud rate, stop, start and data bits. This makes the receiver capable of decoding the data. UART is connected by crossing the TX and RX lines, as shown below:

UART transmits bits sequentially by pulling the UART line low for a fixed amount of time determined by the baud rate. The UART line when idle is high at the I/O level, 3.3V or whatever the VCC of the MSP430 is set. RS232, on the other hand, uses similar signaling to UART but also incompatible voltage levels (the voltages are high and some are negative) that will damage the

MSP430. Therefore, never connect an RS232 device directly to the MSP430. Use a MAX232 or similar device to convert between the two signal levels.

The smallest element of transmission is the UART frame or character, which consists of the Start bit/s, data bits, stop bits and optional parity bits, as shown below:

Bit number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
Start bit	Start bit	Data Bit	Data Bit	Data Bit	Data Bit	Data Bit	Data Bit	Data Bit	Data Bit	Stop Bit	Stop Bit

The start bits alerts the receiver that data is coming. Without it, if the first bit was a '1', it would be seen as an idle line since an idle UART line is also high. The number of data bits is typically 8, but it can be configured for 7 bits as well. Although some UART receivers can use a different

number of bits, only 8 or 8 bits are supported by the MSP430. After the data bits stop bits are sent along with an optional parity bit.

MSP430 families contain different peripherals capable of UART communications. Among these are USCI, USART and eUSCI modules. UART can also be generated using timers or even bit-banged. Some peripherals have sophisticated options that we will not cover since they are rarely used.

Configuring the MSP430 for UART operation

We previously covered the issue of signal multiplexing. In order to use UART, specific pins that are connected to the UART module must be used, and the pin muxing must choose the Primary Peripheral mode. On the MSP430G2553 that is on the MSP430 Launchpad, UCA0 pins are present on pins P1.1 and P1.2 as UCA0RXD and UCA0TXD, respectively. Setting the pin muxing is simple since TI provides the information for the bit settings in the datasheet of the device. In this case, UART requires both PxSEL and PxSEL2 to be set to '1'. PxDIR does not need to be configured.

Configuring MSP430 Pin Muxing for UART

```
P1SEL |= (BIT1 | BIT2);
P1SEL2 |= (BIT1 | BIT2);
```

With the pins configured, we must configure the clocks for UART. The baudrate generation requires a clock of a certain precision. Although baudrates vary, 9600 and 115200 baud are the most common and we will focus on them. In order to generate a baud rate, we must feed a large enough clock so that it can be divided by the baud rate generator for the actual baud rate. For 9600 baud we can use both the slow 32.768kHz crystal if one is present, or a faster source such as the internal DCO or an external crystal. 115200 baud requires a fast clock, so only the DCO and a fast external clock can be used. Below we have the code needed to configure USCI_A0 for UART operation at 9600 baud:

Configuring MSP430 Pin Muxing for UART

```
#include <msp430.h>


int main(void)
{
```

```
WDTCTL = WDTPW + WDTHOLD; // Stop WDT

/* Use Calibration values for 1MHz Clock DCO*/
DCOCTL = 0;
BCSCTL1 = CALBC1_1MHZ;
DCOCTL = CALDCO_1MHZ;

/* Configure Pin Muxing P1.1 RXD and P1.2 TXD */
P1SEL = BIT1 | BIT2 ;
P1SEL2 = BIT1 | BIT2;

/* Place UCA0 in Reset to be configured */
UCA0CTL1 = UCSWRST;

/* Configure */
UCA0CTL1 |= UCSSEL_2; // SMCLK
UCA0BR0 = 104; // 1MHz 9600
UCA0BR1 = 0; // 1MHz 9600
```

```
  UCA0MCTL = UCBRS0; // Modulation UCBRSx = 1

  /* Take UCA0 out of reset */
  UCA0CTL1 &= ~UCSWRST;

  /* Enable USCI_A0 RX interrupt */
  IE2 |= UCA0RXIE;

  __bis_SR_register(LPM0_bits + GIE); // Enter LPM0, interrupts enabled
}

/* Echo back RXed character, confirm TX buffer is ready first */
#pragma vector=USCIAB0RX_VECTOR
__interrupt void USCI0RX_ISR(void)
{
  while (!(IFG2&UCA0TXIFG)); // USCI_A0 TX buffer ready?
  UCA0TXBUF = UCA0RXBUF; // TX -&amp;gt; RXed character
}
```

The clock source for the UART baud generator is SMCLK sourcing the DCO running at 1MHz. Before we can configure the UART peripheral we need to place it in reset mode. Not all registers require this but it is best to do so when first configuring USCI, whether it's for UART or any other mode. Notice that we use the assignment operator, so all the other bits are set to zero. With the reset in place, we make SMCLK the clock source for the UART. Being flexible, there are other possible options for the UCSSELx:

- 00b = UCAxCLK (external USCI clock)
- 01b = ACLK
- 10b = SMCLK
- 11b = SMCLK

UART can actually use a clock coming on a pin instead of one of the internally generated clocks. This can be useful in reducing the number of clocks in the system and reducing system cost.

Configuring the Baudrate

Getting the baudrate from the clock and configuring the baudrate registers is often the most confused to those trying to use the UART capabilities of the MSP430. Obtaining the main divisor is often easy, but adjusting modulation and other bits can be tricky since they need careful

adjustment to reduce error. TI provides a list of register values for common clock frequencies that makes baud rate configuration easier.

Clock	Baudrate	UCBRx	UCBRSx	UCBRFx	UCOS16
1,048,576 Hz	9600	109	2	0	0
8MHz	115200	69	4	0	1
16MHz	115200	8	0	11	1

The two clocks selected represented some of the lowest error rates, as reported by TI. Looking at the table, there are many configuration which can result in large errors for both Transmission and Reception. This doesn't take into account drift and inaccuracy of the clocks, which can make the results worse or better. In general you should aim to keep the UART error below 5% and use higher layer error correction mechanisms such as checksums or CRC to ensure data is properly delivered if you have issues.

With all the configuration of the module complete, all that remains is to take it out of reset so it becomes active, which is done by clearing the UCSWRST bit. The UART is now running. However, instead of polling to see if a character was received, we will use interrupts to echo back a received character.

MSP430 UART Interrupts

UART includes two main interrupt sources: RX and TX. The RX interrupt fires when a character is received and has been placed into the buffer, whereas the TX interrupt is set when the TX buffer is available to be filled with a data.

In the RX ISR, the RXIFG receive flag is cleared automatically when RXBUF is read. You should always read RXBUF or clear the flag. Otherwise, the flag remains set after the ISR returns and the interrupt will immediately trigger and stay in a loop. If you don't need the data in the RXBUF it might also be useful to disable the RXIE interrupt enable. The most common approach with the RX ISR is to store the received character in an array and process all the received characters once all have been received.

MSP430 Launchpad UART

The MSP430G2553 or any MSP430 in the socket is connected to the computer via USB using a combination of MSP430 and a TUSB3410 or similar USB device. In many Launchpads, these devices limit the baud rate to 9600 baud. Note that this isn't a limitation of the MSP430 UART interface itself. Rather the UART to USB converter chain is limited and won't allow baudrates higher than 9600.

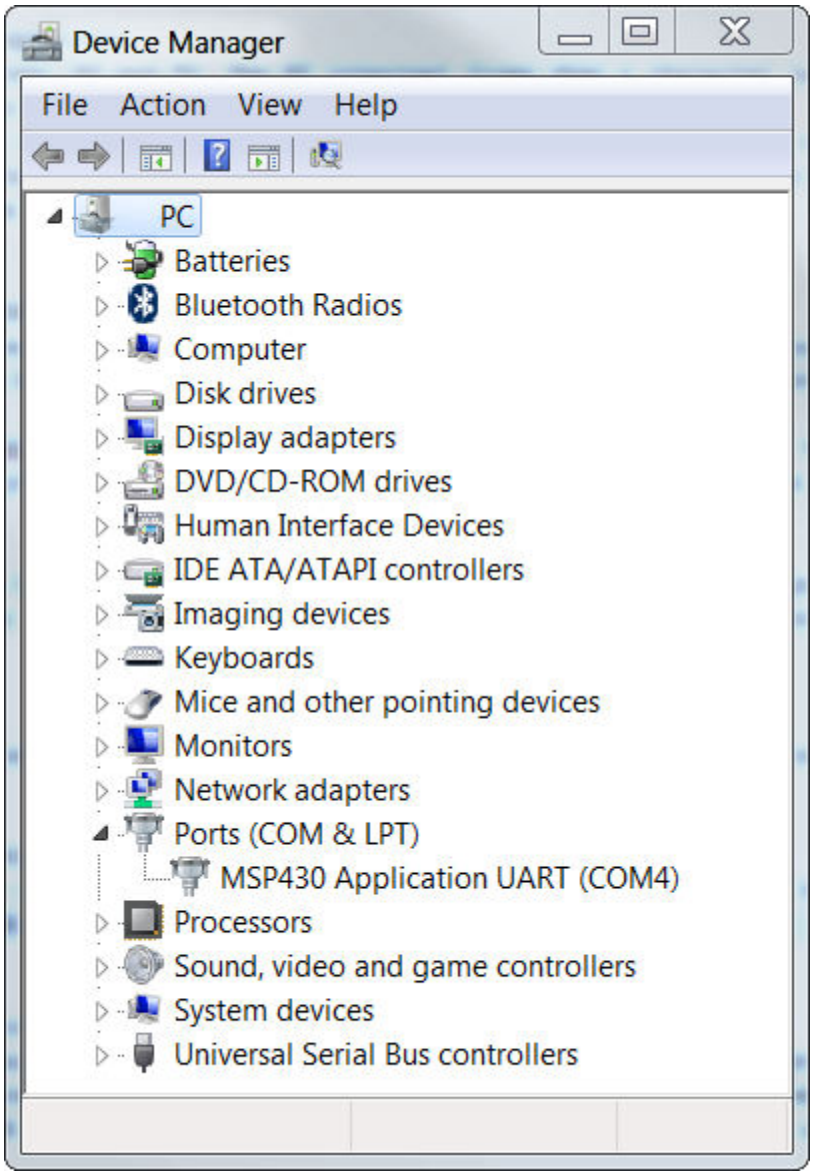
If you need to transmit at higher baudrates to transfer data faster, you will need to connect the MSP430 through a USB to serial (UART) converter. There are many converter solutions, and you can certainly find complete cables, but many of these are based on two devices which are very popular:

- FT232RL and similar devices from FTDI chip are a complete USB to UART solution that integrates the EEPROM and oscillators. Basically with one chip and a few external passives you can have reliable UART communications.
- CP210x from Silicon Labs (Silabs) is another popular device that provides a USB to UART bridge. A few of the devices are also highly integrated

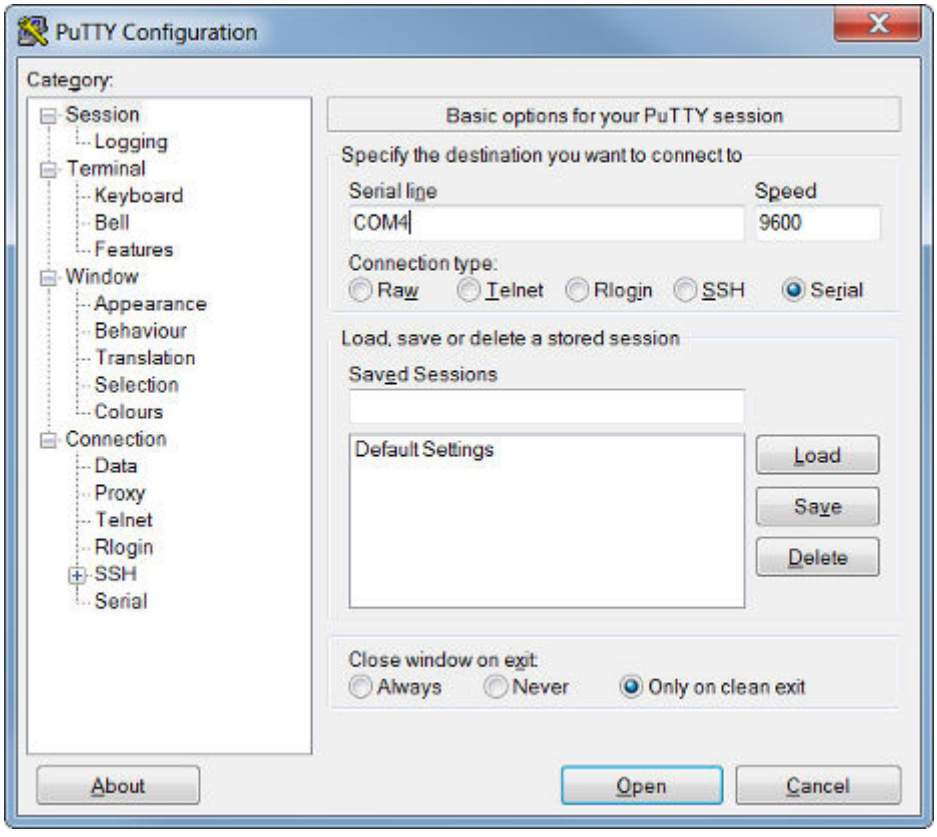
Both of these solutions are easy to use and their drivers are simple and widely available. They can easily support over 1Mbaud.

Talking to the MSP430

The first thing that you should do is to ensure the jumpers on the board are in HW UART mode. Without this, the code above will not work. Download Putty and run it. You need to know to which COM port your MSP430 Launchpad is connected.



We know that the MSP430 UART is connected to COM Port 4. Open Putty and select serial mode.



If putty opens up, you can begin typing and characters will be sent back to you.

Hardware Flow Control

Some systems attempt to transmit as much information as possible by using flow control. Two devices implementing flow control use two extra lines called Clear to Send(CTS) and Request to Send (RTS). The usage of these lines has varied, but at its core the MSP430 UART does not support any flow control. However, it is possible to implement it using GPIOs and interrupts. For example, one can begin transmitting will begin transmission when the GPIO ISR has triggered. It's important to make sure that the RTS and CTS lines are used with ports supporting interrupts, typically ports 1 and 2.

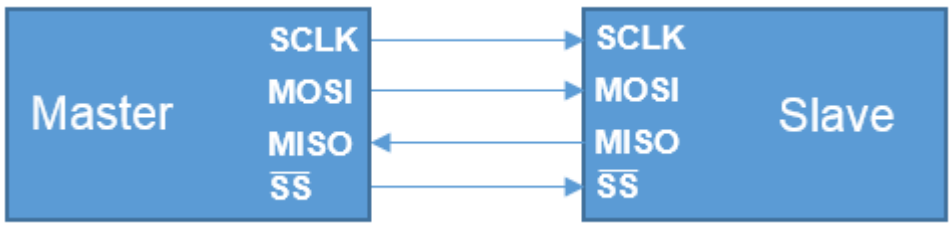
Chapter 8

MSP430 SPI Interface

SPI is one of the most common interfaces in Embedded Systems and it is one of the most utilized in the MSP430. Although it requires more pins than UART, its external clock often enables very fast transfers that are more reliable because both ends are synchronized. The MSP430 supports SPI in various modules.

Physical Interface

A SPI connection between a Master and a Slave device is shown below:



SPI uses the concept of a master device and multiple slave devices. Let's look at some of the signals:

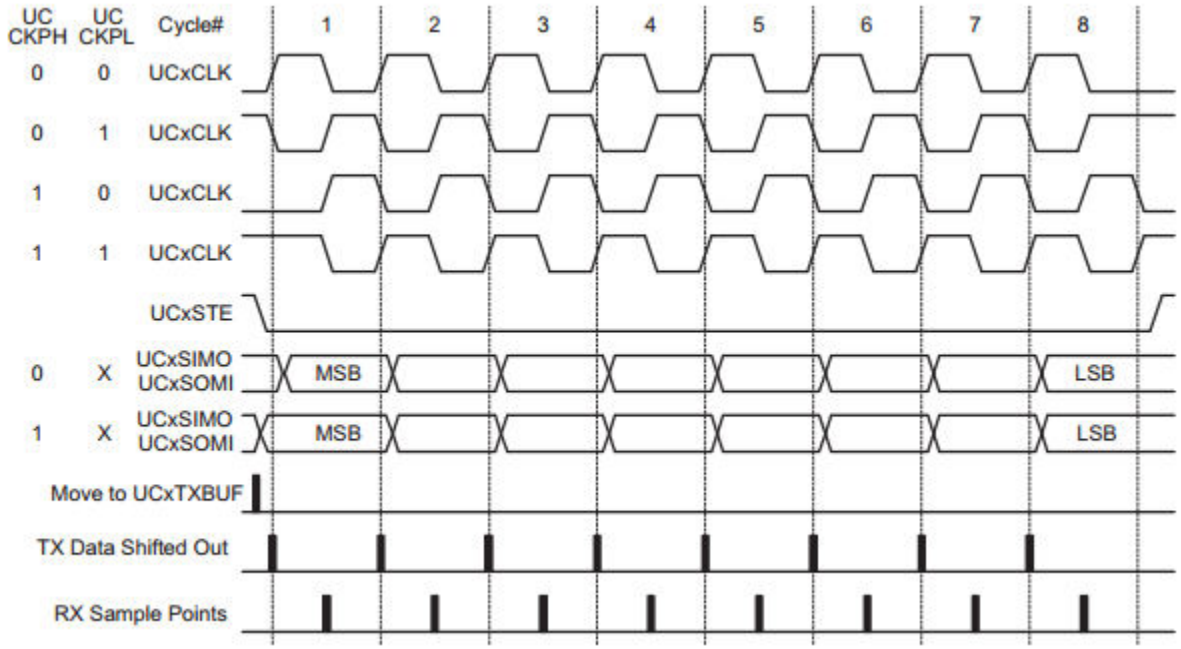
- SCLK – Main clock synchronizing SPI on both devices. Generated by the Master to all the slaves
- MOSI – Master Output and Slave Input in which data is sent from the master to the slave on each clock edge
- MISO – Master Input and Slave Output in which data is sent from the slave to the master on each clock edge
- SS – Slave select, often called CS (Chip Select) or CSn (Chip Select Active Low). This line selects the current active slave

SPI is a full duplex protocol in which the slave and the master exchange data at each transition of the clock by setting MOSI and MISO to a bit value. Note that MOSI and MISO are descriptive name for the connection. Often times they are called SIMO and SOMI or Slave Output (SO) and Slave Input (SI). It's very important to avoid mixing the two signals by crossing them. MOSI should connect to MOSI, MISO to MISO.

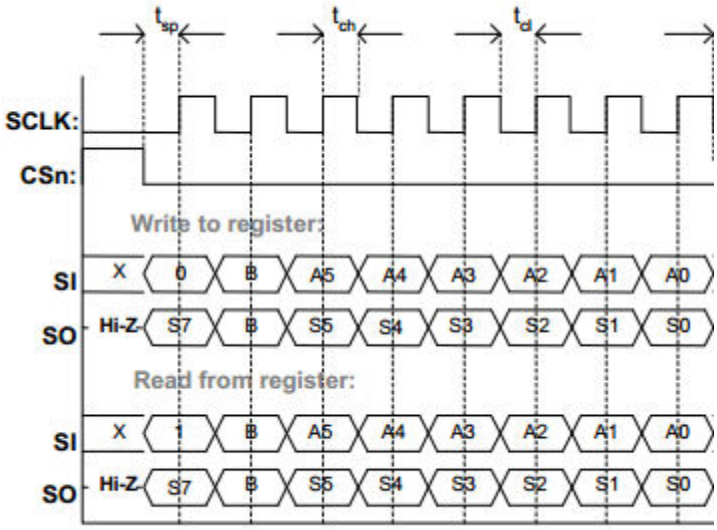
Because it uses a bus topology, the master can talk to multiple devices, although one at a time. The multiplexing of selecting the different devices is accomplished by using a separate Slave

Select line for each device. In some cases where only one slave is present, using a Slave Select might not be required by tying SS on the slave to ground. Some slaves, however, need a falling edge transition to operate properly. You should read the datasheet of the slave device carefully to ensure the waveform generated by the SPI master complies with the requirements of the slave. Slave Select line is typically active low which means it remains high when not sending data to the slave. The SS line is then pulled low for the duration of the transaction, framing it.

We mentioned above that the master and slave exchange data bits which are captured at a clock transition, but we didn't specify whether it was on the rising or falling edge of the clock. We also didn't mention whether the clock is active low or active high. This is because these are configurable and a slave can use either of the 4 SPI modes. The figure below shows the configuration of the Polarity (POL) and Phase (PH) bits in the MSP430 SPI modules.



Its important that the right phase and polarity are selected so that the slave and master both capture the bit at the correct clock transition. If this configuration is not correct, you will see garbage coming out, and the slave will not properly receive the data sent by the MSP430. For example, we can see the SPI interface specifications for the CC1101 transceiver:



The CC1101 needs to sample with a rising edge clock at the center of the bit time. Clock Phase = 1 and Phase = 0 are the correct MSP430 master settings in this case.

Although SPI is full-duplex, oftentimes the data sent by the slave is ignored when the master is writing and vice-versa.

Configuring the MSP430 for SPI

The MSP430 family has various peripherals that support SPI. For the MSP430G2553 that is used with the MSP430 Launchpad this includes the USCI peripherals. Other MSP430 devices in other

families include the Universal Serial Interface (USI) and the USART Peripheral Interface which also support SPI.

Let's look at an example of configuring the USCI in the MSP430G2553 for SPI:

Configuring MSP430G2553 Pins for SPI

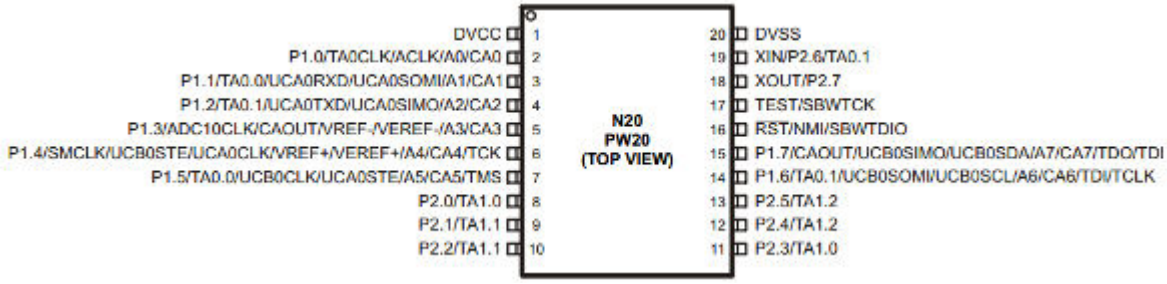
```
#include <msp430.h>


int main(void)
{
   WDTCTL = WDTPW + WDTHOLD; // Stop WDT


   P1OUT |= BIT5;
   P1DIR |= BIT5;
   P1SEL = BIT1 | BIT2 | BIT4;
   P1SEL2 = BIT1 | BIT2 | BIT4;


   UCA0CTL1 = UCSWRST;
   UCA0CTL0 |= UCCKPH + UCMSB + UCMST + UCSYNC; // 3-pin, 8-bit SPI master
```

```
  UCA0CTL1 |= UCSSEL_2; // SMCLK
  UCA0BR0 |= 0x02; // /2
  UCA0BR1 = 0; //
  UCA0MCTL = 0; // No modulation
  UCA0CTL1 &amp;amp;= ~UCSWRST; // **Initialize USCI state machine**
}
```



The MSP430G2553 has two SPI interfaces, mapped with the following pins:

UCA0

- P1.1 – UCA0SOMI
- P1.2 – UCA0SIMO
- P1.4 – UCA0CLK

UCB0

- P1.6 – UCB0SOMI
- P1.7 – UCB0SIMO
- P1.5 – UCB0CLK

The code above also uses P1.5 as CSn. Why haven't we used STE? Because STE is not slave select. The Slave Select operation on the MSP430 must be performed using GPIOs. The STE pin, on the other hand, is used when the MSP430 operates as a slave and allows master arbitration. It will not act as a slave select. To enable SPI we need to make the pins for MISO, MOSI and CLK in peripheral mode so USCI will control them. We then configure a custom pin such as P1.5 above, to be the SS or CSn. Looking at the port schematics in the G2553 datasheet we can see that to make the pins work with USCI we need to set Both P1SEL and P1SEL2 to '1'.

You might have noticed that we change P1OUT before PDIR, which might seem a bit unnecessary. After all, the output doesn't take effect unless the direction is Output. The reason for this is that if we were to first set the direction, the output register might have been set to '0'. In the time between the output register being 0 and being set to 1, we will cause an undesirable and momentary glitch on the line. The line might go From High Impedance (since by default the pins are in high impedance), to '0' and then to '1'. The change from '0' to '1' is mostly harmless, but can sometimes cause issues with devices.

Note that for the cases when we want to use both SPI interfaces, we would need to change CSn from P1.5 to another pin so that P1.5 can be used for UCB0CLK.

With the pinout configured, it's time to configure the USCI module for operation. As when we configured USCI for UART, we first place the module in reset. We then configure several bits of the UCA0CTL0 register. The code sets the SPI controller to be a Master sending MSB first, which is set by UCMSB and UCMST. UCSYNC selects SPI mode for the module since it supports various protocols and we want to select SPI specifically. UCCKPH enables the right SPI mode for the CC110x transceivers and we use it only as an example. You will need to adjust this for your own device.

As with UART, SPI mode also needs an input clock. The code above selects SMCLK as the source clock for the module with the UCSSEL_2 bit. Just after selecting the clock source, the clock divider is set to device the input clock (SMCLK in this case) by 2. Note that the USCI module must have an input clock divided by at least 2. So, if SMCLK is running at 16MHz, our SPI will run at 8MHz. With the clocks and module configured, we can take the module out of reset and it will be ready to run. Because feeds the master clock to the SPI slave, both master and slave are synchronized which means clock accuracy is not critical. Because of this we don't have to worry

about accurate clocks when using SPI. SPI can theoretically operate at any frequency, but the MSP430 and slave limit the maximum frequency that can be used.

Transmitting and Receiving

Once out of reset, the SPI peripheral is ready. Let's see how we transmit a byte. Note that this is a blocking implementation. In practice you will want to use interrupts which will enable the MSP430 to do other things while the SPI module is operating.

Configuring MSP430G2553 Pins for SPI

```
#include <msp430.h>


volatile char received_ch = 0;
int main(void)
{
   WDTCTL = WDTPW + WDTHOLD; // Stop WDT


   P1OUT |= BIT5;
   P1DIR |= BIT5;
   P1SEL = BIT1 | BIT2 | BIT4;
```

```
P1SEL2 = BIT1 | BIT2 | BIT4;


UCA0CTL1 = UCSWRST
UCA0CTL0 |= UCCKPH + UCMSB + UCMST + UCSYNC; // 3-pin, 8-bit SPI master
UCA0CTL1 |= UCSSEL_2; // SMCLK
UCA0BR0 |= 0x02; // /2
UCA0BR1 = 0; //
UCA0MCTL = 0; // No modulation
UCA0CTL1 &= ~UCSWRST; // **Initialize USCI state machine**


P1OUT &= (~BIT5); // Select Device


while (!(IFG2 & UCA0TXIFG)); // USCI_A0 TX buffer ready?
UCA0TXBUF = 0xAA; // Send 0xAA over SPI to Slave
while (!(IFG2 & UCA0RXIFG)); // USCI_A0 RX Received?
received_ch = UCA0RXBUF; // Store received data


P1OUT |= (BIT5); // Unselect Device
```


```
}
```

The first thing to do is to select the SPI slave by putting P1.5 low using the P1OUT register. After this we need to check whether the TXIFG flag is set. As with UART, this flag is set when the TX buffer is ready to be filled with data. In our case we know that no other operation has been performed on the SPI module, so there is no reason to check and this is a formality. But it is important to build code that works well in all cases. Note that the while loop waits while the flag is 0 and will exit immediately once the flag is set. As soon as it is, we can send a byte by filling the TXBUF. As soon as TXBUF is filled, the SPI will generate SCLK and transfer the bits. But, how do we know when the transmission is done? We should never interrupt a slave by raising the slave select during a transmission since most devices will interpret that as an aborted transmission and will discard the bits. If you remember, SPI is a duplex protocol which means that the slave also sends back data. In most cases this data is 0x00 or 0xFF (garbage), but we can use the RXIFG flag to wait until the transmission and the reception on the master side is over. After we receive the data, we store the byte received in a variable and then terminate communications by raising P1.5 High again.

With these elements it is possible to build a complete application to talk to one or multiple slaves.