1. Lab 1: Introduction to embedded C programming
    1. 1.1 - What is a program?
    2. 1.2 - Introduction to the IAR Workbench IDE
    3. 1.3 - Introduction to Programming the ez430
    4. 1.4 - Setting Breakpoints in Workbench
    5. 1.5 - Lab 1: C and Macros with Texas Instruments' ez430
2. Lab 2: Introduction to embedded Assembly programming
    1. 2.1 - Introduction to Assembly Language
    2. 2.2 - Structure of an Assembly Program
    3. 2.3 - Lab 2: Introduction to Assembly Language
3. Lab 3: Clocking
    1. 3.1 - What is a digital clock?
    2. 3.2 - Clock System on the ez430
    3. 3.3 - Lab 3: Clocking on MSP430
4. Lab 4: Interrupts and Timers
    1. 4.1 - Interrupts
    2. 4.2 - Timers
    3. 4.3 - Watchdog Timer
    4. 4.4 - Lab 4: Timers and Interrupts
5. Lab 5: Optimization and Low Power Modes
    1. 5.1 - Memory Conservation
    2. 5.2 - Improving Speed and Performance
    3. 5.3 - Reducing Power Consumption
    4. 5.4 - Lab 5: Optimization and Low Power Modes
6. Lab 7: Analog to Digital Conversion
    1. 7.1 - Introduction to Sampling
    2. 7.2 - Analog-to-Digital Converter on the MSP430
    3. 7.3 - Lab 7: The ADC

1.1 - What is a program?

A **program** is a set of instructions that are grouped together to accomplish a task or tasks. The instructions, called **machine code** or **assembly code** consist of things like reading and writing to memory, arithmetic operations, and comparisons. While these instructions sound simple, it is actually possible to solve a huge group problems with them. The difficulty in doing so is that you must specify in exact detail precisely how. Good programming is both an art and a science, and what you will learn today is a beginning of the craft.

As mentioned above, the individual instructions that the machine actually quite simple or **low-level** in computer parlance. Writing complex programs in assembly code took such a long time that eventually better **programming languages** were invented. A programming language, like C, is a formal set of grammar and syntax like assembly code; but the instructions in **high-level** languages encompass hundreds of assembly instructions. Programs called **compilers** translate a program written in a higher level language into assembly so that the computer can actually execute the instructions. Compilers let the programmer write programs so that humans can read them relatively easily while the computer can still execute the instructions.

Generally programming code is organized into text files with suffixes that indicate the programming language. In the case of C these files are appended with ".c", and a C program is made up of at least one of these files. Many C programs also use **header** files that contain frequently used segments of code so that it does not need to be written multiple times. A ".h" is appended to the end of these files.

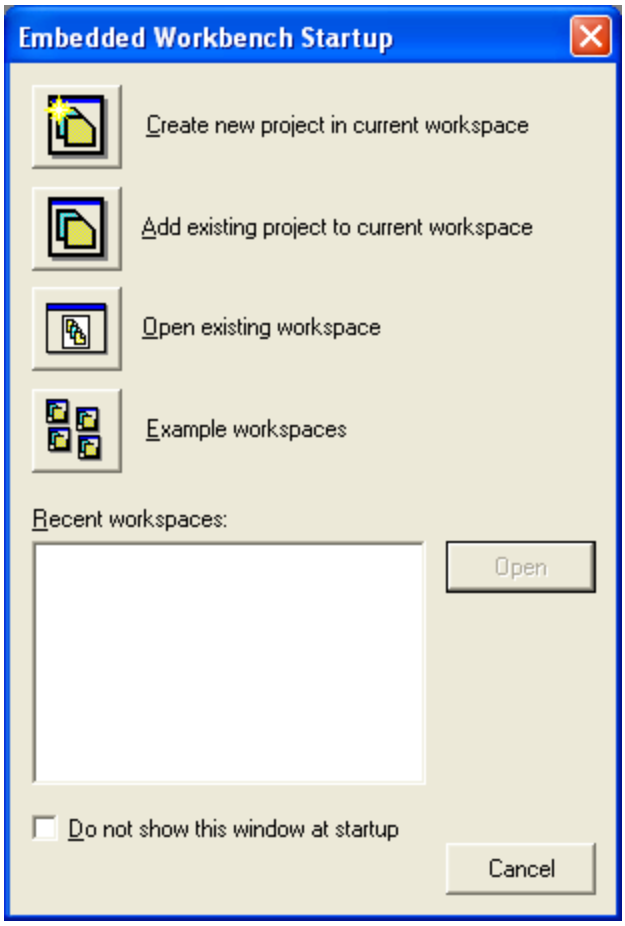
1.2 - Introduction to the IAR Workbench IDE

Goals

To develop applications to run on the ez430 chip, we use the IAR Embedded Workbench IDE (integrated development environment). Not only does this application provide a powerful code editor, but it also allows a simple one-click deployment of the source code onto the MSP chip using USB as well as hardware debugging capabilities that allow you to trace through actual stack calls. This module is intended to give you an introduction to the IAR Workbench application so that you may create and develop your own ez430 applications.

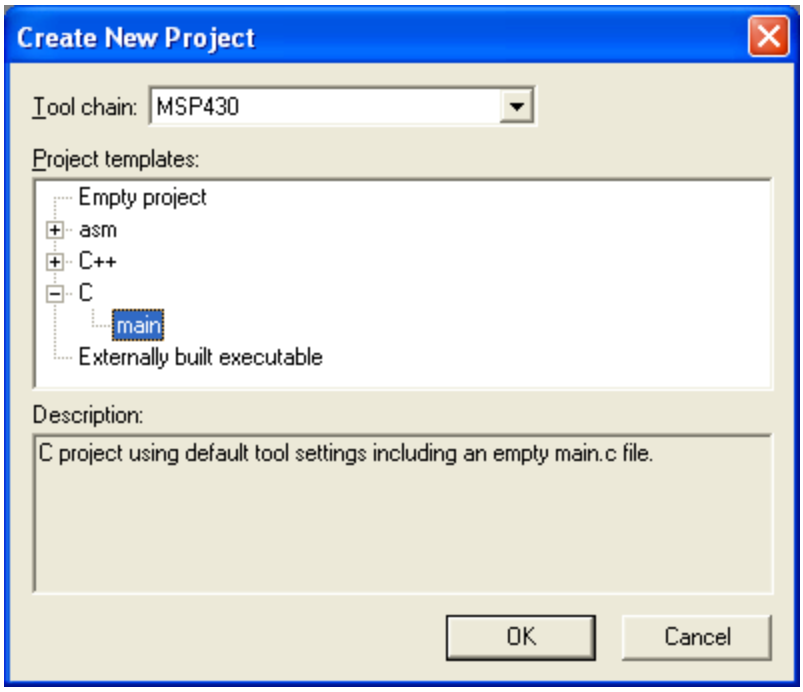
Create a Project

The very first thing you must do before you can start downloading any code onto the MSP, is to create a project in Workbench that will contain all of the relevant files for your application. When you open Workbench you should see the following window (This window is equivalent to selecting **File->New->Workspace** and then **Project->Create New Project**. This window may also be reached through Help->Startup Screen):
Startup Screen



Select "Create new project in current workspace" to begin.

Next, the following window will appear:
Project Template Selection



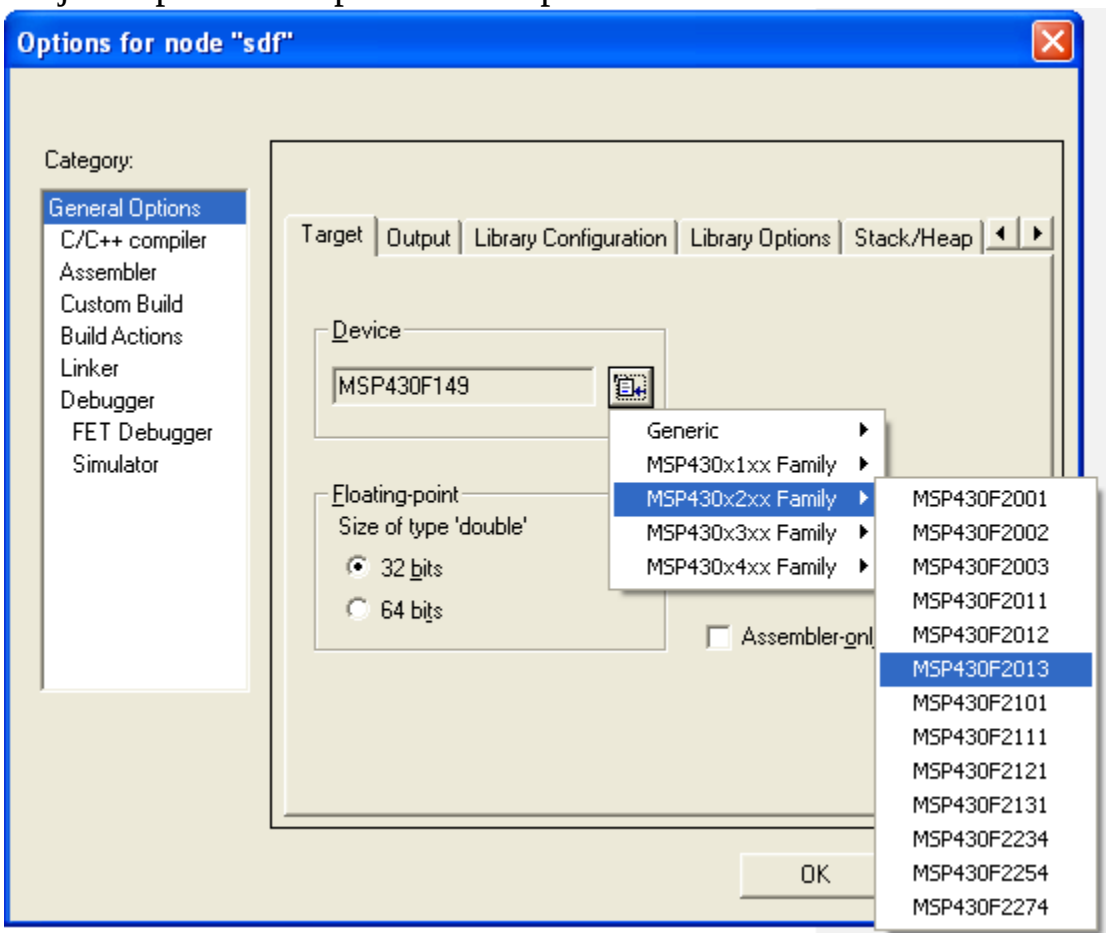
Here we select the language in which we shall write our code. Select the 'C' file and then the 'main' option. In one lab, we will program in assembly; for that lab, select the "asm" option.

Now a Windows Save dialog should appear. Give your project a name and save it in its own **seperate folder**. A project is not only a C file that contains your code but also several other files (a project file with the ".eww" extention, a project settings folder, a debug folder which contains compiled code outputted by Workbench etc). Keeping the files for your current project in their own folder will help you stay organized. If you save and close your project, opening it again is as simple as opening this folder, and doubleclicking on the Workbench project file, which has a ".eww" extention.

Configure Project Options

Now, we must configure the project options. Make sure the project is selected in the "Files" pane to the left of the screen (the project is the top of the workspace's file tree). Then select **Project->Options** to get the following screen and make the indicated selections:

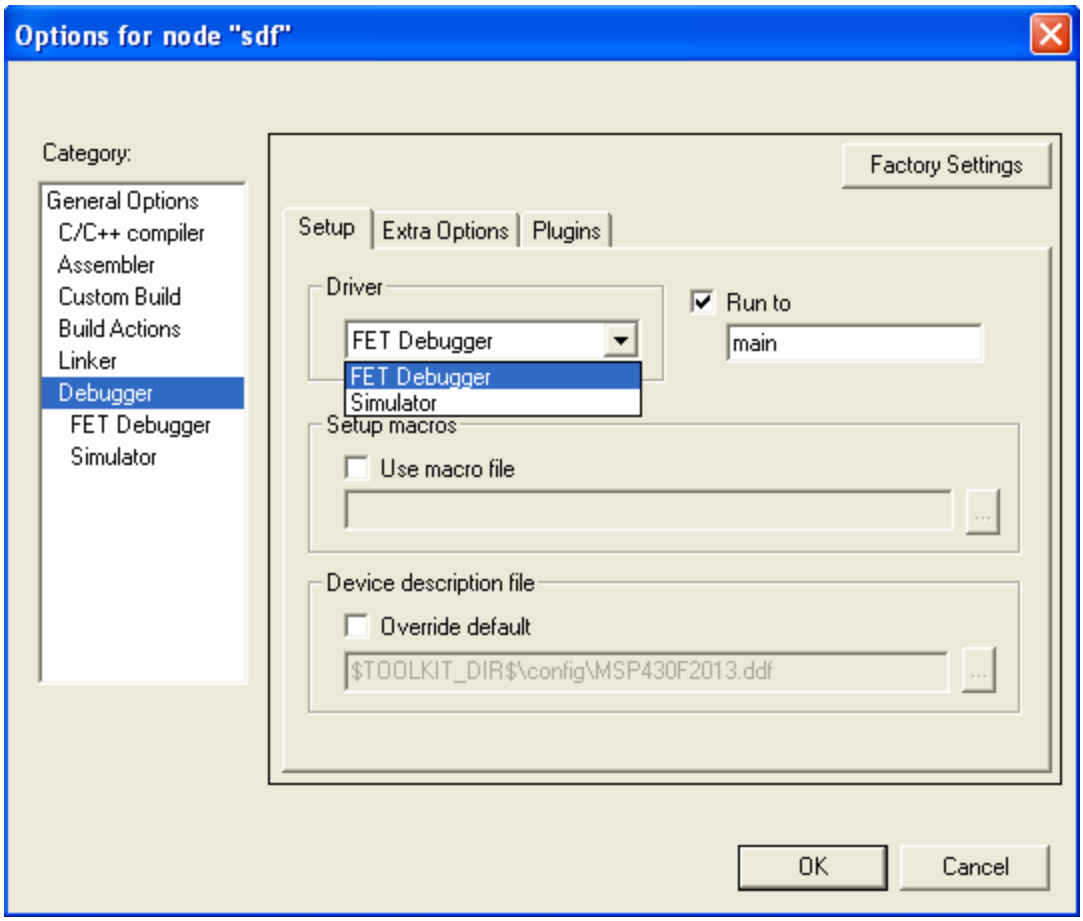
Project Options Setup: General Options



Make sure you choose the right processor for your application! Here, I've selected the MSP4302013 (a member of the ez430 series) since that is the chip I plan to use.

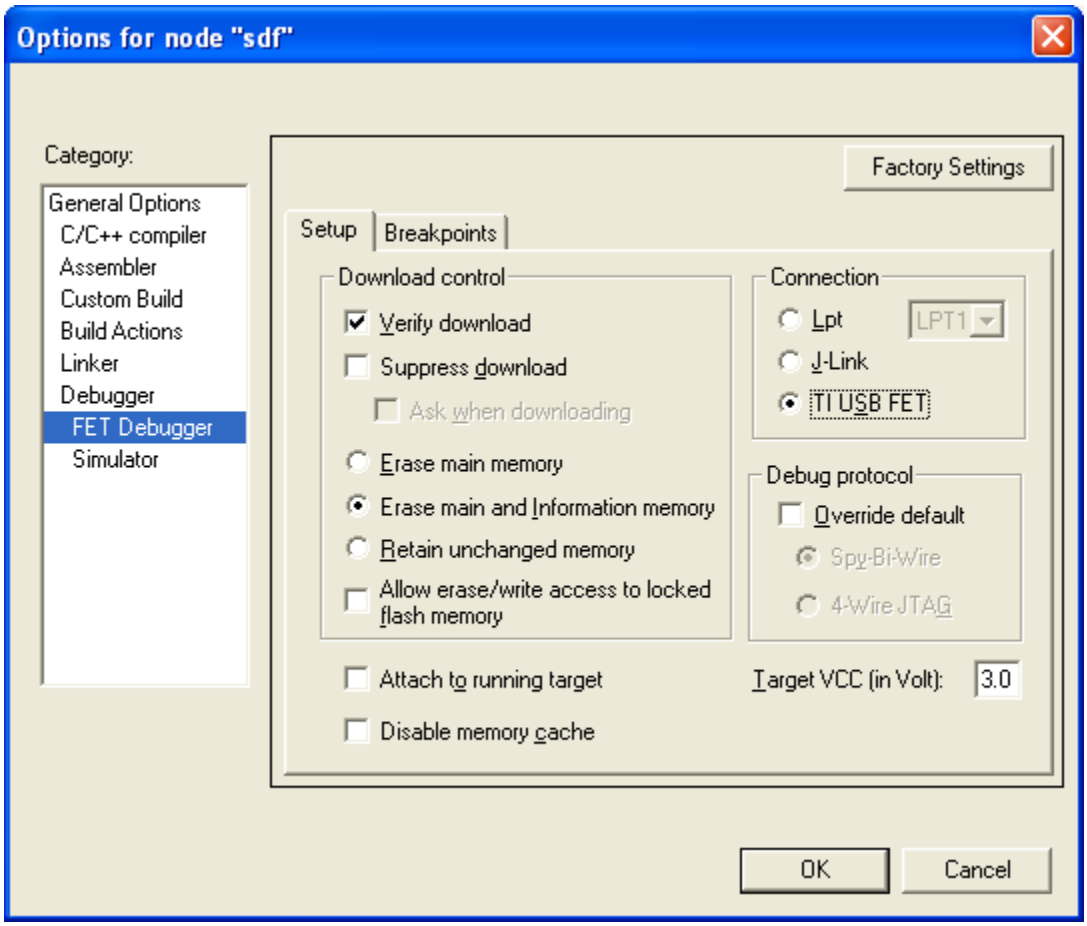
Now select the "Debugger" option and make the following selections:

Project Options Setup: Debugger



Here we must select to use the FET (Flash Emulation Tool) Debugger to run our code.

Finally select the "FET Debugger" option and make the following selections:
Project Options Setup: FET Debugger



Now we select the "Verify Download" check box and the "TI USB FET" radio button (while leaving other options as they are) so that our program is properly transferred to the chip.

Your project is created, saved, and now you may write your code. Only one project may be open at a time.

Adding Files to the Project

If your project consists of more than one file, you will need to add it to the current project. To add a file to your project, choose **Project->Add Files**.

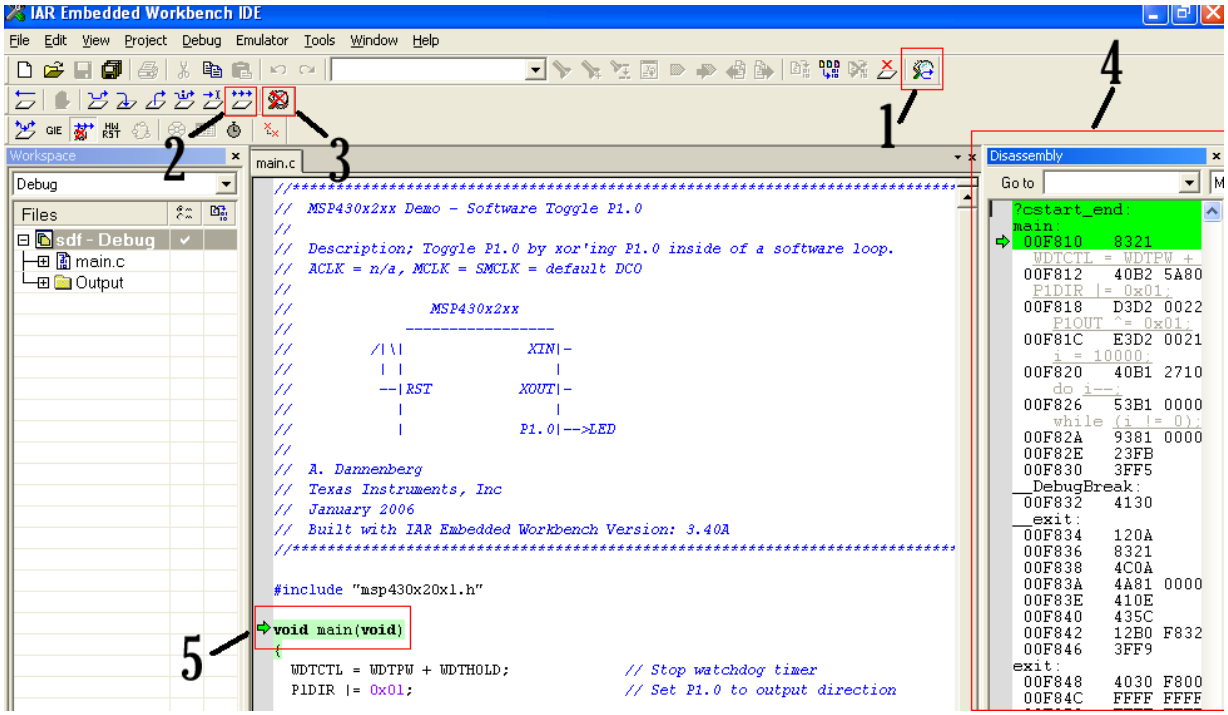
Run Your Program!

This is the part you've been waiting for: actually testing your program! Use the **Project->Debug** menu item to load the currently active project and set your program running. You can also set any breakpoints beforehand; by default, the very first line of executed code is set as a breakpoint. Now that you have started the debug process, the layout of the Workbench program changes slightly. A Debug menu and the "Disassembly", the assembly/machine code instructions at their specific memory addresses on the ez430, appear. To continue the execution of the program, select **Debug->Go**. If you would like to set breakpoints (in order to view the "path" of program execution or the values of specific variables or registers at a certain point during the program execution) click to the left of a line of code such that a red 'X' appears.

You can pause the debugger when the target is running if you would like to look at the value of certain variables. Select **Debug->Break** and open a watch window to examine the value of certain variables. It is suggested that you copy the variables you are interested in to temporary global variables. Because local variables go out of scope, it is uncertain if their correct value is maintained when the debugger is paused.

Debugging can be stopped using **Debug->Stop**. At this point, the watch window will not display your variable values. At this point, you may make any modifications to your program as necessary and restart the debugger from the beginning.

All of the necessary commands have shortcut icons as the following diagram shows:
Workbench: Debug View



The program layout during debugging

Legend:

1. **Make/Debug**- This button will compile, load, and run the code onto the chip. This process happens rather rapidly so if the execution of this seems to hang, then something is wrong. Usually disconnecting the tool and/or restarting Workbench solves this problem. Don't forget to save first!
2. **Go**- This button is for the Go command in the debug menu; it tells the program to continue execution until the next breakpoint. The buttons to its left are other ways of traversing through breakpoints that you may experiment with.
3. **Stop Debug**- Selecting this will end the debugging process. If you want to modify your code then you must stop debugging first.
4. **Dissassembly**- This is a map of the memory space of the ez430 and where your code has been placed (in its compiled form).
5. **Current Location**- The Green Arrow/Green Highlighting indicates the current instruction being executed.

Good luck!

You're all set to start using the CrossStudio compiler to write embedded microcontroller code. Save often and remember that disconnecting and reconnecting or restarting the program will solve almost all interface problems that you encounter. Don't worry if breakpoints seem a little confusing now. We will go into more depth shortly and a good bit of practice is necessary to use them effectively.

1.3 - Introduction to Programming the ez430

Configuring Digital I/O: Introduction

Digital I/O, such as the LED, is configured by modifying several registers pertaining to the port that they are attached to. Check the datasheet to find the respective port number of the peripheral you which to control. For more detailed information about Digital I/O on the MSP430 check **Chapter 6: Digital I/O** of the ez430's user guide. The basic logic behind seemingly trivial process of turning the LED on and off is the same behind the operation of every peripherial on the tool.

First, we must assign the direction of the corresponding I/O pins. This is done by setting the direction register, **PxDIR**, with the appropriate bit. By default, all I/O pins are assigned to be inputs.

- Bit = 0: The port pin is switched to input direction.
- Bit = 1: The port pin is switched to output direction.

On the ez430 the LED is located on port 1. The port number will correspond the to x value in registers such as **PxIN**, **PxOUT**, or **PxDIR**. Therefore, if we wanted to define the direction of a pin on port 1 we would write to **P1DIR**.

A Useful Tangent: Common Bitwise C Operations

In order to modify values of registers you must first understand the following commonly used C logic operations:`~A` `A | B` `A & B` `A ^ B` `A |= B` `A &= ~B` `A ^=B`

In C, the NOT: `~`, OR: `|`, AND: `&` and XOR: `^` operators are all bitwise. This means that the operations are done bit by bit to the binary values of the two operands.

Note: The prefix "0b" to a number means that that number is binary. Similarly an "0x" prefix means that the number is hexadecimal.

Here are examples of the bitwise operations: `int A = 0b1101; int B = 0b0101; int C; /* The results of these operations are very straightforward. If we are dealing with the "|" operator then we OR together each bit to make the result. This means 0b1101 | 0b0101 is computed as [1 OR 0, 1 OR 1, 0 OR 0, 1 OR 1] which is [1,1,1,1] thus the result is 0b1101; */ C = ~A; // The value of C is 0b0010 C = A | B; // The value of C is 0b1101 C = A & B; // The value of C is 0b0101 C = A ^ B; // The value of C is 0b1000`

Operators such as `+=` or `-=`, implemented as `A += B`, are short forms for `A = A + B`. This is translated to "take the current value of A, add it to B, and finally store that result back into A." For `-=` it similarly translates to "subtract B from A and store the result back to A." If "@" is some operator, than `A @= B` would be "@ together A and B and store the result back into variable A."

If we combine the use of these two aspects of C then the aformentioned logic operations that are listed at the top of this section begin to make sense. The following examples will help you figure out why they are useful to us when we attempt to configure any register on our microcontroller. It is imperative that you understand how these function. You will use them in every program you write because these operations allow you to modify single bits of a register to modify a specific register and change a peripheral's settings.

Configuring Digital I/O: Examples

Exercise:

Problem:

How do we switch the pin (P1.0) corresponding to the LED to be an output?

Solution:

`P1DIR |= 0x01;` 0x01 in hex is equivalent to 0b00000001 in binary. You may refer to the module about Binary and Hexadecimal Notation to learn how to do this conversion or you could use the Windows Calculator (**Start->Run...->'Calc'**) to do it quickly. We can now easily see that P1.0 is set to 1 which makes it an output.

Note: Remember to use `|=` instead of just `=` because it won't overwrite bits that are not set to 1 in the mask.

Output pins may be toggled using the **PxOUT** register. The LED is turned on by setting its corresponding register bit **high**.

Exercise:

Problem:

What code would turn on the LED without modifying any other bits in the register? What would turn it back off?

Solution:

```
P1OUT |= 0x01; //Turns LED On P1OUT &= ~0x01; //Turns LED Off
```

This will turn on and turn off the LED assuming it had already been set to be an output.

Note:~0x01 = 0xFE = 0b11111110

Exercise:

Problem:

What would be the proper syntax for toggling the value of P1.0 so that we can turn the light on if it is off or vice versa?

Solution:

`P1OUT ^= 0x01;` The XOR operation will invert the bit in question so that we may toggle it regardless of its current state. If this doesn't make sense, do the computation by hand to see how that single bit is flipped.

Exercise:

Problem:

Now, write the full C program to turn on the LED. Do the following:

1. Create a new project in Workbench as previosuly described
2. Do **not** delete the default code in the main.c file except for the `return 0;`, which is not necessary. All of your programs will use this as a skeleton
3. Include the correct header file by adding the following line at the top of the main.c file: `#include "msp430x20x3.h";`
4. Consider defining macros for commonly used register values. For example, if we add `#define led_on ~0x01` to the top of the file (after the `#include`) we may call `led_on` every time we wanted the value ~0x01. Similarly we may add `#define led_off 0x01` to use when we wanted 0x01.
5. Complete the program. It is as simple as it seems; no more than two lines of code in the main function are necessary

Solution:

```
#include "msp430x20x3.h" void main(void) {
P1DIR |= 0x01; // Set P1.0 as output P1OUT |=
0x01; // Set P1.0's output value as high (turns
the LED on) }
```

1.4 - Setting Breakpoints in Workbench

C with an embedded controller does not have as many input-output (IO) features as a regular computer. To help you debug, it will sometimes be necessary to stop the processor while it is running and examine the state of the system. To accomplish this we will use **breakpoints**. A breakpoint is a specific command to the development environment to stop execution of the processor when a certain condition happens. These conditions range from when a certain instruction is reached to when certain data is written or read. The advanced options are broad.

To set a basic breakpoint, one which will stop execution when a certain line is reached, just click on the left margin of the C file on the line you want to trigger. A red 'X' should appear to indicate you have set a break point. Click once more to make it go away. Workbench keeps track of all of the breakpoints for you. To see this information go to **Window->Breakpoints**. This will pop up a list of all of the breakpoints you currently have enabled. Right clicking in the window will allow us to create, delete, and modify breakpoints. We will use the same terminology for breakpoints as the IAR environment does, but the usage is not standardized.

The most useful and important breakpoint is the the **code** breakpoint. It triggers on arrival at a certain instruction in the source code. This kind of breakpoint can be created simply by left clicking on the left margin of the line in question. The breakpoint should then appear in the breakpoint window. Right clicking on the entry for a breakpoint allows you to edit it. For all breakpoints, you may select that they only trigger on a certain iteration by editing the **skip count** field. If left as 0, the breakpoint will trigger each time it occurs. Entering a number into the skip count field will trigger the breakpoint on that numbered time the event occurs.

In tandem with breakpoints, we can use the **Locals** and **Watch** windows. To access either of these windows select **View->Locals** or **View->Watch**. Whenever the program pauses at a breakpoint, these windows will be updated with the proper current values. The locals window displays the values for all local variables, variables within the function being processed, that are currently being used. The watch window displays values for variables that you add to it. Right click any variable or macro directly in

your code and select "Add to Watch." Now, whenever your code pauses at a breakpoint, you can see the current value of that variable.

Alternate explanations of breakpoints can be found in the help contents of the Workbench IDE.

1.5 - Lab 1: C and Macros with Texas Instruments' ez430

The primary difference between "normal" and embedded C programming is that you will need to write directly to registers to control the operation of the processor. Fortunately, the groundwork has already been laid for you to make this easier. All of the registers in the ez430 have been mapped to macros by Texas Instruments. Additionally, the important bit combinations for each of these registers have macros that use the same naming convention as the user's guide. Other differences from the C used on most platforms include:

- Most registers in the ez430 are 16 bits long, so an `int` value is 2 bytes (16 bits) long.
- Writing to registers is not always like writing to a variable because the register may change without your specific orders. It is always important to read the register description to see what the register does.
- The watchdog timer will automatically reset the ez430 unless you set the register not to.
- There is only a limited "standard out" system. Standard out will typically print results to your computer screen.
- Floating-point operations cannot be efficiently performed. In general, you should avoid floating point decimal calculations on the ez430 because it does not have special hardware to support the complicated algorithms used.

Exercise:

Problem:
Code Review

In this exercise, you may want to use some of the debugging tools (Breakpoints, Watch Window, Locals Window) help you understand what the code is doing. Start a new project. Cut and paste the following code into main.c:

```
#include "msp430x20x3.h" void main(void){ int i,j,tmp; int a[20]=
{0x000C,0x0C62,0x0180,0x0D4A,0x00F0,0x0CCF,0x0C35,0x096E,0x02E4,
0x0BDB,0x0788,0x0AD7,0x0AC9,0x0D06,0x00EB,0x05CC,0x0AE3,0x05B7,0x001D,0x0000
for (i=0; i<19; i++){ for (j=0; j<9-i; j++){ if (a[j+1] < a[j]) { tmp = a[j]
a[j] = a[j+1]; a[j+1] = tmp; } } } while(1); }
```

1. Explain what this program is doing. Why is the `while(1)` statement at the end of the code necessary for all programs we write at this point?
2. Use any of the methods listed above to show the updated array. What is the final result?
3. Modify the code so that it prints the final version of the array to standard out (you will have to use a loop of your choice to cycle through each element of the array). What are the drawbacks and benefits of using `printf` over setting a breakpoint?

Note: To use the standard out, add the following line to the top of your code: `#include "stdio.h";` Then, select **Project->Options**, the **Library Options** tab in **General Options**, and finally select **Tiny** in the **Printf formatter** drop down menu. The `printf()` function will print to standard out (when the debugger is running, select **View->Terminal I/O**). For example, `printf("x equals %d\n", x);` will print out the value of x to the window. The `%d` means that x is a number, and `\n` will produce a line break.

Exercise:

Problem:
Functions

Multiplications and division are very complex operations to do on any microprocessor. The operations should be avoided if possible or should be replaced with simpler, equivalent operations.

1. What do the operators << and >> do?
2. How could you use these operators to perform multiplication and division?
3. Write the function `multiply(int x, int y)` that takes parameter `x` and multiplies it by `y` by using a bit shift. It must return an `int`. For symplicity, it is OK to assume that `y` is a power of 2.
4. Next, write the function `divide(int x, int y)` that takes parameter `x` and divides it by `y` by using a bit shift. It must also return an `int`.

Exercise:

Problem:
Digital I/O Registers

Open the header file **msp430x20x3.h** by right clicking the name in your code and selecting **Open "msp430x20x3.h"** This file contains the macros and register definitions for the ez430 we are using. Using the ez430 schematic, this header file, and ez430's User Guide please answer the following questions.

1. The Watchdog Timer will automatically reset the hardware if it isn't periodically reset or disabled entirely. Usually, we will simply disable it. It can be disabled by writing to the `WDTPW` (watchdog timer password) and `WDTHOLD` (watchdog timer hold) section of the Watchdog Timer Control Register (`WDTCTL`). Refer to **Section 7.3** of the User's Guide for more information. Find the macros for this register in the header file. How are they different from their description in the User's Guide? Finally, write the C code required to disable it.
2. What are the differences among P1DIR, P1SEL, P1OUT, P1IN?
3. Some port pins have multiple functions to output and it is up to the user to select the appropriate signal. Write some code that would select the alternate function of P2.2 (pin 2 of port 2). What will the result be on our hardware?

Exercise:

Problem:
Programming Digital I/O

Write a program to blink SOS in Morse code repeatedly. In Morse code SOS is S:"..." O:"---" S:"..." where each '.' is a shorter blink and a '-' is a longer blink. According to The International Morse Code Standard (on Wikipedia) the relative lengths of times between dots and dashes are as follows:

- The amount of time of a dash is equivalent in length to 3 dots.
- The amount of time between parts of a letter is equivalent in length to one dot.
- The amount of time between letters is equivalent in length to 3 dots.
- The amount of time between words (assume each SOS is a word) is equivalent to 5 dots.

Make the delays (time length of a dot) by using a for-loop that counts to a large enough value.

Note: Make sure you disable the watchdog timer at the very beginning of the program.

2.1 - Introduction to Assembly Language

Assembly Language

Assembly language, commonly referred to as assembly, is a more human readable form of machine language. Every computer architecture uses its own assembly language thus processors using an architecture based on the x86, PowerPC, or TI DSP will each use their own language. **Machine language** is the pattern of bits encoding a processor's operations. Assembly will replace those raw bits with a more readable symbols call **mnemonics**.

For example, the following code is a single operation in machine language. `0001110010000110` For practical reasons, a programmer would rather use the equivalent assembly representation for the previous operation. `ADD R6,R2,R6 ; Add $R2 to $R6 store in $R6` This is a typical line of assembly. The **op code** `ADD` instructs the processor to add the **operands** `R2` and `R6`, which are the contents of register R2 to register R6, and store the results in register R6. The "`;`" indicates that everything after that point is a comment, and is not used by the system.

Assembly has a one-to-one mapping to machine language. Therefore, each line of assembly corresponds to an operation that can be completed by the processor. This is not the case with high-level languages. The **assembler** is responsible for the translation from assembly to machine language. The reverse operation is completed by the **dissasembler**.

Assembly instructions are very simple, unlike high-level languages. Often they only accomplish a single operation. Functions that are more complex must be built up out of smaller ones.

The following are common types of instructions:

- Moves:
    - Set a register to a fixed constant value
    - Move data from a memory location to a register (a load) or move data from a register to a memory location (a store). All data must

be fetched from memory before a computation may be performed. Similarly, results must be stored in memory after results have been calculated.
    - Read and write data from hardware devices and peripherals

- Computation:

    - Add, subtract, multiply, or divide. Typically, the values of two registers are used as parameters and results are placed in a register
    - Perform bitwise operations, taking the conjunction/disjunction (and/or) of corresponding bits in a pair of registers, or the negation (not) of each bit in a register
    - Compare two values in registers (>, < , >=, or <=)

- Control Flow:

    - Jump to another location in the program and execute instructions there
    - Jump (branch) to another location if a certain condition holds
    - Jump to another location, but save the location of the next instruction as a point to return to (a call)

Advantages of Assembly

The greatest advantage of assembly programming is raw speed. A diligent programmer should be able to optimize a piece of code to the minimum number of operations required. Less waste will be produced by extraneous instructions. However, in most cases, it takes an in-depth knowledge of the processor's instruction set in order to produce better code than the compiler writer does. Compilers are written in order to optimized your code as much as possible, and in general, it is hard to write more efficient code than it.

Low-level programming is simply easier to do with assembly. Some system-dependent tasks performed by operating systems simply cannot be expressed in high-level languages. Assembly is often used in writing **device drivers**, the low level code that is responsible for the interaction between the operating system and the hardware.

Processors in embedded space, such as the ez430, have the potential for the greatest gain in using assembly. These systems have very limited computational resources and assembly allows the maximum functionality from these processors. However, as technology advances, even the lowest power microcontroller is able to become more powerful for the same low cost.

2.2 - Structure of an Assembly Program

The assembly program begins execution at the reset interrupt. The reset interrupt is the first thing that occurs when power is given to the processor. By default in the Workbench files, the reset interrupt is loaded to send the execution of the program to the start of the written code. Until a branch is reached, the processor will execute each instruction in turn. If the program does not loop back to an earlier point to keep going, eventually the execution will reach the end of the valid instructions in memory and attempt to execute the "instructions" in the following memory addresses (which are invalid and possibly gibberish). You should never let this happen.

The control of a programs execution is called **control flow**, and it is accomplished through branching, jumping, function calls, and interrupts. Interrupts are the subject of future labs. Branching and jumping refer to changing the next instruction from the next one sequentially to an instruction elsewhere in the program. By branching to an instruction above the branch itself you can cause the program to repeat itself. This is a basic loop in assembly. Branches can also be conditional. In the MSP architecture conditional branches are generally dependent on the status register (SR) bits to decide whether to execute the next instruction after the branch or the instruction the branch specifies. Many arithmetic and logical operations can set the relevant bits in the status register; check the ez430's User's Guide for which ones you will need. A full description of each of the Assembly instructions for the ez430 can be found in **Section 3.4**.

To store values to perform operations, you must use the ez430's registers to store values. The ez430 has 16 CPU registers. Of these 16, the upper 12 are general purpose 16 bit registers (R4-R15). The lower four are:

- R0 Program Counter(PC) – This register controls the next instruction to be executed by the MSP core. In general, this register is incremented automatically during execution. It can be used as a source in operations normally.
- R1 Stack pointer (SP) – The stack pointer is used to keep track of previous execution modes and to return from interrupts. Can be read as a normal register.

- R2 Status Register (SR) – The status register can be written to change the operating mode of the MSP as specified in the User's Guide. When read it can act as a constant generator. Depending on the instruction code options this register will be read as: a normal register, 0x0000, 0x0004, or 0x0008 depending on the As bits.
- R3 Constant Generator II – This register cannot be written to, and when read produces: 0x0000, 0x0001, 0x0002, or 0xffff depending on the As bits.

The rest of the registers on the ez430 behave as if they were memory. In most cases, these special purpose registers can be read and written to normally but they affect the behavior of their respective systems.

Once you understand the basics of assembly you should be able to write some simple routines. Once you create a new Assembly project in Workbench, replace the default code with the following. You'll see that the Watchdog timer has already been deactivated. Put your assembly code in the place indicated. Also, notice the difference in location of instructions and **labels**. Labels, which mark the begining of certain blocks of code, are left aligned such as the "RESET" seen below. Instructions, however, are tabbed over. Remember to follow this convention because the compiler will assume anything left aligned is a label.

```
#include "msp430x20x1.h"
;-------------------------------------------------------------------------- ORG 0xF800 ;
Begining PsuedoOP ;-------------------------------------------------------------------------------------------------
RESET mov.w #0x280,SP ; Set stackpointer mov.w #WDTPW+WDTHOLD,&WDTCTL ; Stop watchdog timer ;;
**YOUR CODE GOES HERE** ;-------------------------------------------------------------------------------------------------
---- ; Interrupt Vectors ;-------------------------------------------------------------------------------------------------
----- ORG 0xFFFE ; MSP430 RESET Vector DW RESET ;
END
```

2.3 - Lab 2: Introduction to Assembly Language

In this lab you will be asked to write simple assembly statements, follow already written assembly statements, and finally to reproduce the SOS light blinking program from Lab 1 in assembly.

Exercise:

Problem: Formulate instructions to do the following things:

1. Set bit 3 to 1 at the memory address 0xd640 while leave bits 0-2 and 4-16 unaffected.
2. Jump to the instruction labeled POINT if the carry bit is set.
3. Shift register R6 right one place while preserving the sign.

Exercise:

Problem:

Examine this loop: `... more instructions... Mov.w &I, R4 Cmp.w #0x0000, R4 JZ After_loop Start_loop: Dec.w #0x0001, R4 JZ After_loop BR #Start_loop After_loop: ...more instructions...`

1. How many times will this loop execute?
2. Why do we use the BR instruction with a #Start_loop, but with the JZ we use a plain After_loop?
3. What does the first JZ instruction do? If we did not have this initial Cmp and JZ, what (possibly) inadvertent effect might occur?

Exercise:

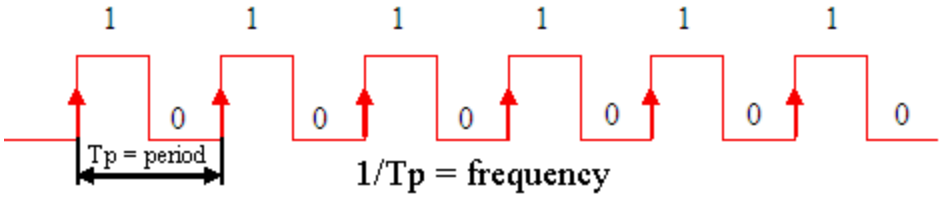
Problem:

Re-write the blinking light program from Lab 1 using assembly code instead. As you may recall, the program must blink "SOS" (which is "... --- ...") and conform to the following Morse Code standards:

- The ammount of time of a dash is equivalent in length to 3 dots.
- The ammount of time between parts of a letter is equivalent in length to one dot.
- The ammount of time between letters is equivalent in length to 3 dots.
- The ammount of time between words (assume each SOS is a word) is equivalent to 5 dots.

3.1 - What is a digital clock?

The clock of a digital system is a periodic signal, usually a square wave, used to trigger memory latches simultaneously throughout the system. While no part of this definition is strictly true, it does convey the basic idea (think flipping a light switch on and off at a certain speed). Square waves are used because the quick transitions between high and low voltages minimize the time spent at uncertain digital levels. The clock ideally reaches all parts of the system at the same time in order to prevent sections from getting out of sync. Clock signals are generally periodic because the user wants to run the system as fast as possible, but this is often not a necessary attribute.



Clock signals are used to synchronize digital transmitters and receivers during data transfer. For example, a transmitter can use each rising edge of the clock signal of Figure 1 to send a chunk of data.

A faster clock rate all means that you can process more instructions in a given amount of time at the cost of an increased power consumption.

3.2 - Clock System on the ez430

The clock system on the ez430 is designed to be flexible and low powered. The implementation of these goals is largely based on the ability to select different clock speeds for different parts of the chip. By choosing the minimum clock speed necessary for a given module, power consumption is reduced and the particular synchronization needs of the module can be met.

Our ez430 technically has two main clock sources for the clocking system, and three clock lines that chip systems can choose between. The clock sources are used as the basis for the clock lines, and this allows for a mix of slow and fast clocks to be used around the system. Currently, without modification, our tool only supports one.

Clock Sources

1. **Low Frequency Crystal Clock (LFXTCLK)** – This clock is sourced from an external crystal (with an intended oscillation of ~32kHz) that does not exist by default on the ez430. Theoretically, we could attach the crystal to the XIN/XOUT pins and be able to use this as a clock source although such a procedure is not necessary for this course. This crystal would be the source of the **Auxiliary Clock (ACLK)**, one of the three clock lines discussed below
2. **Digitally Controlled Oscillator Clock (DCOCLK)** – this is the only internally generated clock input, and it is the default clock source for the master clock upon reset. By default this clock runs at about 1MHz, but the RSELx, MODx, and DCOx bits allow this to be divided down or even blended to achieve a lower clock frequency on average. **Chapter 4** of the User's Guide outlines specific ways to configure the DCOCLK to operate at the desired frequency.

Clock Lines

1. **Master Clock (MCLK)** – This clock is the source for the MSP CPU core; this clock must be working properly for the processor to execute instructions. This clock has the most selection for its source. The source is selected with the SELMx bits of the Basic Clock System Control Register 2 (BCSCTL2). The divider is controlled with the DIVMx of the BCSCTL2. Finally, the CPU can be turned off with the

CPUOFF bit of the Status Register (SR), but to recover from this state an interrupt must occur.

2. **Submaster Clock (SMCLK) -** This clock is the source for most peripherals, and its source can either be the DCO or Crystal 2. The source clock is controlled with the SELS and SCG bits of the BCSCTL2 and SR. The divider is controlled by the DIVSx bits of the BCSCTL2.
3. **Auxiliary Clock (ACLK) -** this clock line's source is always LFXTCLK. It is an option for slower subsystems to use in order to conserve power. This clock can be divided as controlled by the DIVAx bits of the Basic Clock System Control Register 1 (BCSCTL1).

The MSP clock system has dividers at the beginning of its clocks, and at most peripheral destinations. This allows for each module to keep a separate timing scheme from other modules by dividing the input frequency and then outputting it. The simplest dividers are multiples of two, thus the output might be a square wave of one half, quarter, or eighth the input square wave's frequency. This is often necessary for off chip buses because these systems have to meet a variety of speed requirements from the outside. For educational purposes the fastest clocks are usually the most useful, but remember that power consumption is the primary cost of high speed clocks.

3.3 - Lab 3: Clocking on MSP430

The following exercise will show you how to manipulate the clocking system on the ez430. You will need to refer to the ez430's Schematic and User's Guide to correctly configure the clock as specified.

Exercise:

Problem:
Clock Setup

In order to easily check the state of the clock, output MCLK/SMCLK (they are both defaulted to the same source) from a pin header (HINT: Output the SMCLK from P1.4 on pin 6). Use the oscilloscope to observe the frequency of the clock, and to see the impact of the changes you will make.

1. Without modifying the clock registers any further, at what clock rate is the processor running at? How is the clock currently configured in order to produce this built-in clock signal? In other words, what is the clock's source and how is the source configured?
2. Write code that sets the DCOCLK to operate at 8MHz. You should be able to do it in just two lines of code. We will use this clock setup for all future programs unless otherwise noted.
3. Using your new (8MHz) timer settings and your SOS light system from Lab 1, modify the code so that each "dot" and "dash" delay is for approximately the same ammount of time. Basically, generate the same behavior as the original version but using the new clock settings. What were the original (much slower) settings for the clock? What was changed in order to keep the original blink rate?

Exercise:

Problem:
VLO Clock Setup

The following is code to set the MSP430 to operate off of the VLO clock. The only problem is that it doesn't work. Macro definitions and logic operations are incorrectly used (although the comments are correct). Without adding or removing any lines of code, properly source the MSP430 off the VLO clock. `WDTCTL & ~WDTPW + WDTHOLD; // Stop watchdog timer BCSCTL1 |= LFXT1S_2; // LFXT1 = VLO IFG1 = OFIFG; // Clear OSCFault flag __bis_SR_register(SCG1 + SCG0); // Stop DCO` Now using these timer settings, repeat number 3 from this lab's problem 1.

4.1 - Interrupts

An **interrupt** is an event in hardware that triggers the processor to jump from its current program counter to a specific point in the code. Interrupts are designed to be special events whose occurrence cannot be predicted precisely (or at all). The MSP has many different kinds of events that can trigger interrupts, and for each one the processor will send the execution to a unique, specific point in memory. Each interrupt is assigned a word long segment at the upper end of memory. This is enough memory for a jump to the location in memory where the interrupt will actually be handled. Interrupts in general can be divided into two kinds- maskable and non-maskable. A **maskable** interrupt is an interrupt whose trigger event is not always important, so the programmer can decide that the event should not cause the program to jump. A **non-maskable** interrupt (like the reset button) is so important that it should never be ignored. The processor will always jump to this interrupt when it happens. Often, maskable interrupts are turned off by default to simplify the default behavior of the device. Special control registers allow non-maskable and specific non-maskable interrupts to be turned on. Interrupts generally have a "priority;" when two interrupts happen at the same time, the higher priority interrupt will take precedence over the lower priority one. Thus if a peripheral timer goes off at the same time as the reset button is pushed, the processor will ignore the peripheral timer because the reset is more important (higher priority).

The function that is called or the particular assembly code that is executed when the interrupt happens is called the Interrupt Service Routine (ISR). Other terms of note are: An interrupt flag (IFG) this is the bit that is set that triggers the interrupt, leaving the interrupt resets this flag to the normal state. An interrupt enable (IE) is the control bit that tells the processor that a particular maskable interrupt should or should not be ignored. There is usually one such bit per interrupt, and they are often found together in a register with other interrupt enable bits. The most important interrupt on MSP430 is the reset interrupt. When the processor detects a reset or powers up for the first time, it jumps to the beginning of memory and executes the instructions there. The highest priority interrupt vector begins at the address 0xFFFE. The lowest priority interrupt begins at 0xFFE0. The complete set of interrupts is ranked by priority:

- 8 - non-maskable : External Reset, Power-up, Watchdog Timer Reset, Flash Key Violation, NMI
- 7 - non-maskable : Oscillator Fault, Flash Memory Access Violation
- 6 - maskable : Watchdog Timer
- 5 - maskable : Timer A Capture Compare Register 0 (CCR0) Interrupt
- 4 - maskable : Timer A Capture Compare Register 1 (CCR1) Interrupt
- 3 - maskable : Sigma/Delta 16 bit (SD16) Converter Interrupt
- 2 - maskable : Universal Serial Interface (USI) Interrupts
- 1 - maskable : Port 2 I/O Interrupts
- 0 - maskable : Port 1 I/O Interrupts

When the interrupt first occurs on the MSP there is a precise order of events that will occur. This process takes 6 instruction cycles AFTER the current instruction completes.

1. The program counter as it is after the above instruction is pushed onto the stack. The stack is memory whose contents are kept in last in first out order. The stack pointer is always updated to point to the most recent element added to the stack. This allows the processor to call functions and track interrupts. When something is pushed onto the stack, the stack pointer is incremented and the pushed data is written to that location. When you copy out of the stack and decrement the stack pointer, this is called popping something off the stack.
2. The status register is pushed onto the stack.
3. The highest priority interrupt waiting to occur is selected.
4. Single source interrupts have their interrupt request flags reset automatically. Multiple source interrupt flags do not do this so that the interrupt service routine can determine what the precise cause was.
5. The status register with the exception of the SCG0 bit is cleared. This will bring the processor out of any low-power modes. This also disables interrupts (the GIE bit) during the interrupt.
6. The content of the interrupt vector is loaded into the program counter. Specifically the processor executes the instruction at the particular memory location (the interrupt vector) for that interrupt. This should always be a jump to the interrupt service routine.

The interrupt service routine is the code that the programmer writes to take care of the work that needs to be done when a particular interrupt happens. This can be anything you need it to be. Because entering the interrupt turned off the GIE bit, you will not receive any interrupts that happen while you are still in the interrupt service routine. You can turn the interrupts back on if you need to receive interrupts during your interrupt, but usually it is a better idea to make interrupt service routines shorter instead. In C interrupts are simply functions with special declarations. You never call these functions; the compiler simply sets up the interrupt vector table to call your function when the particular interrupt occurs.

This example interrupt is pulled from the msp430x20x3_wdt_01.c example file by Mark Buccini. The complete file can be found on the Ti-ez430 homepage. `// Watchdog Timer interrupt service routine #pragma vector=WDT_VECTOR __interrupt void watchdog_timer(void) { P1OUT ^= 0x01; // Toggle P1.0 using exclusive-OR }`

Interrupt functions should always be void and accept no arguments. This particular interrupt service routine (ISR) is called `watchdog_timer`, but the name does not matter. The way the compiler knows that this function should handle the watchdog timer interrupt is what follows the function name. the `#pragma vector =` indicates that this is an interrupt and `WDT_VECTOR` is a macro from the MSP header file that indicates the interrupt vector being used. Every interrupt vector in the processor has a macro defined for it (which can be found in the header file). To attach this interrupt service routine to a different interrupt, all you need to do is change the `WDT_VECTOR` to one of the other macros defined in the ez430 header file.

When the end of the ISR is reached the MSP executes a precise set of steps to pick up the execution of the program where it left off before the interrupt occurred. This process takes 5 cycles.

1. The status register and all previous settings pops from the stack. Any alterations to the status register made during the interrupt are wiped away.

2. The program counter pops from the stack and execution continues from where it left off.

Interrupt Enable Registers

Using interrupts successfully is not as simple as just writing an interrupt service routine and waiting for the event to occur. Because you do not necessarily want to activate every interrupt in the processor at once, the MSP allows you to mask out certain interrupts. When the triggering event first occurs, the processor checks whether the interrupt is enabled before jumping to the interrupt service routine. For most interrupts, the MSP checks the general interrupt enable bit in the status register and the particular interrupt's enable in the interrupt enable registers. If both of these have been configured to allow the interrupt, then the processor enters the interrupt service routine if the event in question has occurred.

By default most interrupts are turned off upon reset. To use most peripheral modules you will need to set the enable bits in the interrupt enable registers and turn on the general interrupt enable. Enabling sometimes causes the interrupt flag to be set, so you should consult the User's guide on the best order to handle the enabling. Usually to properly configure the interrupt, you will also need to have set up the peripheral module in question before enabling the interrupt.

There are three categories of interrupts for the purpose of masking in the ez430. Reset interrupts, non-maskable non-reset interrupts, and maskable interrupts.

Maskable interrupts are the lowest priority interrupts and can be turned off individually with the various interrupt enable registers or turned off as a group by setting the general interrupt enable bit (GIE) in the status register (SR).

Non-maskable interrupts are not subject to the general interrupt enable (GIE). However each non-maskable interrupt source can be controlled by a bit to specifically turn it on or off. These are the flash access violation interrupt enable (ADDVIE), external NMI interrupt enable (NMIIE), and

the oscillator fault interrupt enable (OFIE). All three of these bits are located in the interrupt enable register 1 (IE1).

Reset interrupts have the highest priority and will always reset the execution of the device. The external reset can be configured to trigger either the reset interrupt or an NMI interrupt.

The interrupt enable registers (IE1 and IE2) are used to individually enable the interrupts. Refer to the ez430's User’s Guide and Data sheet for the specifics of each peripheral. The example code on Ti's website are also very helpful when learning how to use interrupts. The procedures followed are drawn from the instructions and notes in the documentation. Often the relevant information may not be in one chapter or section of the guides. This is part of the reason working examples are essential to developing a working knowledge of the processor.

More detailed information on the operation of interrupts can be found in the MSP User’s Guide. Unfortunately the material is generally found in the chapter for each subsystem. The general interrupt information is found in chapter 2.

4.2 - Timers

The Timer systems on the ez430 are a versatile means to measure time intervals. The timer (called Timer A) can measure the timing on incoming signals or control the timing on outgoing signals. This function is necessary to meet arbitrary timing requirements from outside components, and the ability is useful in phase locking scenarios etc. The Timer system is one system (and the most common system) to make use of interrupts Interrupts and Timers are bing introduced in tandem to illustrate this operation although they are seperate entities.

The most basic operation of the timer systems is that the counter in the module increments for each clock cycle. The timer clocks can be sourced from any of the clock lines discussed in Lab 3. The incoming source can be divided down by multiples of two before entering the counter. The user's guide for the MSP has good diagrams on the architecture of the system in the respective sections for the timer. Below the features and uses of Timer A are outlined.

Timer A Modes

Timer A is a flexible system; the main counter can vary its counting pattern among several options with the MCx bits of the Timer A Control Register TACTL. These modes are:

1. **Stop:** the counter is not running
2. **Up:** the counter counts upward from zero to the value in Timer A Compare Latch 0 (TACL0). When it gets to this value, it resets to zero. If the TACL0 value is larger than the maximum value of the Timer A counter, the counter behaves as if it were in Continuous mode.
3. **Continuous:** the counter counts from zero to the maximum value of the Timer (0xFFFF). When the counter reaches this value, it resets to zero.
4. **Up/down mode:** the counter counts up to the value in TACL0 then counts back down to zero (as opposed to resetting directly to zero). If the TACL0 value is larger than the maximum value of the Timer A counter, the counter behaves as if it were in Continuous mode.

Please note that the word count is used above- if the timer is set to a certain number it will not trigger anything. The timer must count from the number below to the target number to trigger effects.

Timer A Capture Compare Register 0

There are 2 capture compare registers in Timer A. While there is only one counter for both modules, they can each interpret the count independently. The most important module is module 0 because it controls the timer with its TACL0 register. Primarily it controls rollovers, but it also has its own dedicated interrupt. Setting this module up correctly is essential for desired operation of Timer A.

What is Capture/Compare?

A capture is a record of the timer count when a specific event occurs. The capture modules of the timers are tied to external pins of the ez430. When the control registers of Timer A and the specific capture compare module have been properly configured, then the capture will record the count in the timer when the pin in question makes a specific transition (either from low to high or any transition). This capturing event can be used to trigger an interrupt so that the data can be processed before the next event. In combination with the rollover interrupt on Capture module 0, you can measure intervals longer than 1 cycle.

A compare operation is less intuitive than the capture, but it is basically the inverse of a capture. While capture mode is used to measure the time of an incoming pulse width modulation signal (a signal whose information is encoded by the time variation between signal edges), compare mode is used to generate a pulse width modulation (PWM) signal. When the timer reaches the value in a compare register, the module will give an interrupt and change the state of an output according to the other mode bits. By updating the compare register numbers, you change the timing of the signal level transitions.

This may sound somewhat complicated, but the basic concept of measuring (input) or controlling (output) the time interval between high to low and

low to high transitions is all you need to know to start with. The MSP capture/compare modules have many different ways to perform each operation. This can be somewhat overwhelming, but it allows the microprocessor to handle inputs from a greater variety of other components. Capturing and comparing are done with the same modules, and each one can be configured individually. They can also be grouped using the TACTL to trigger the capture compare registers to load simultaneously (useful for compare mode). The ez430 User's Guide fully details the behavior of the modules and the registers that control them.

Timer Interrupts

There are two interrupts related to timer A. One interrupt is dedicated to capture compare module 0; and, depending on configuration, it fires when the counter rolls back to zero. The second interrupt handles all of the modules except for the first one, and fires to indicate that the module has captured or compared as explained above. Here this simply means the interrupt handles the second module. However, in other devices that have more capture/control registers, this interrupt would handle this entire set of modules. Each module can be individually masked or enabled, and a special register stores which module caused the interrupt. As with all maskable interrupts, setting the general interrupt enable is necessary to receive them. The interrupts are important in being able to perform complex operations as new data arrives.

4.3 - Watchdog Timer

What is the Watchdog Timer?

Software stability is a major issue on any platform. Anyone who uses software has probably experienced problems that crash the computer or program in question. This is also true of embedded programs, and in most cases there is no user around to reset the computer when things go wrong. That job is occupied by the watchdog timer. The watchdog timer is a 16 bit counter that resets the processor when it rolls over to zero. The processor can reset the counter or turn it off, but, correctly used, it will reset the processor in case of a code crash. To avoid getting reset, the program must reset the timer every so often. A program which has crashed will not do so, and the system will reset. To improve its efficacy, the watchdog timer register also requires a password. In order to change the lower part of the watchdog control register, the upper part of the register must be written with a specific value. This value is specified by the alias WDTPW in the MSP header files. This password reduces the likelihood that a random crashed instruction could prevent the reset.

Other uses for the Watchdog Timer

In situations where a system crash is not a concern, the watchdog timer can also act as an additional timer. The watchdog timer can be configured to give an interrupt when it rolls over; this interrupt could also be used to handle system crashes. While the watchdog timer is not as versatile as the other MSP430 timers, the watchdog control register, `WDTCTL` still allows selection of the timer's divider and clock source. Often the watchdog timer is simply turned off by setting the hold bit in the control register. Any changes to this register require writing the password to the upper bits.

4.4 - Lab 4: Timers and Interrupts

In this lab, we will cover the timing options for the ez430. The first part explains the clocking system of the processor and the options it allows while the second part covers the timers and timer interrupts available on the ez430. Each of these sections is strongly related to real time programming, but that topic will be dealt with separately in another lab. In general, a real-time application is one which responds to its inputs as fast as they happen. The microprocessor is generally expected to take less time to handle an interrupt or stimulus than the time before the next event will happen. The timer system is broken into two primary parts on the ez430 Timer A and the Watchdog timer.

Exercise:

Problem:
Timer A

Refer to **Chapter 8: Timer_A** User's Guide to get a detailed description of all the options in Timer A. Basically, setting up the timers requires that you define a source for the timer and to specify a direction for the count. It may also be helpful to clear the timer register before you begin to guarantee and accurate count from the first instance. Set up Timer A in Continuous Mode and sourced from SMCLK. Set TACCR0 and TACCR1 to have two different values. Output TA0 and TA1 from Pins 3 and 4 (off of P1.1 and P1.2) of the board so that you may directly observe the output of Timer A.

Using two different channels of the Oscilloscope try to recreate parts of **Figure 8-13. Output Example- Timer in Continuous Mode**. On Channel 1 show **Output Mode 4: Toggle** and on Channel 2 show **Output Mode 6: Toggle/Set**. Vary the TACCTLx in order to get as close to the orginal figure as possible. Take a screenshot of the scope and include it in your lab report.

Exercise:

Problem:
Timer

Set up the timers to fire interrupts to calculate time intervals and redo the SOS problem from Lab 1 using the timer to simulate the "dot" and "dash" time intervals. There should be **NO COUNTING LOOPS** in your program, and your program should be entirely interrupt driven. It is possible to have each Capture Control Register to fire an interrupt once it reaches its max value. Explain how you setup Timer A to simulate each time interval.

Exercise:

Problem:
Duty Cycle

We have discussed earlier that the Duty Cycle is related to the width of a pulse. If we trigger an LED with a relatively high frequency square wave, it will appear to be lit constantly even though it is actually switching on and off quickly. As we vary the duty cycle of our trigger signal, the LED may appear to get dimmer or brighter depending on which way we vary it.

Set up the timer to toggle the LED. Without changing the frequency of your timing pulse, change the duty cycle so that the LED appears to fade in and out. The time it takes to go from completely off to max brightness shouldn't take more than a second, then it should repeat. Once again, there should be no counting loops in your program, and you can use Timer A in any way you wish.

Once you get a single light to fade in and out, create another program with a function to set the LED at a certian brightness level when given atleast a 12 bit integer. For example, if I were to call `LED_level(0x111)`, the LED should appear very dim; if I were to call `LED_level(0xFA0)`, the LED should appear very bright. It may be helpful to have an extra function that initializes the Timer settings so that the use of this application is self contained. In the future, we will use this function to provide visual feedback when using other components.

5.1 - Memory Conservation

In the early days of computers, the instruction memories of main frames were incredibly small by today's standards, only in the hundreds or thousands of bytes. This small capacity emphasized making each instruction count and each value saved necessary. Fortunately, just as processors have exponentially increased in speed, program memory has similarly increased in capacity. Even though it seeme like there is an abundance, there are still general practices that must be kept in mind when using program memory. Also, smaller platforms, like microcontrollers, are still limited to memory capacities in the kilobytes. This module explains the kinds of memory, common memory organizations, and the basics of conserving memory.

How memory is organized

In most memory architectures there is some categorization of memory into parts. The basic principle behind this is that seperate sections of memory can serve specific purposes while each section can be more efficiently accessed. Types of memory include the following:

- **Instruction Memory** is a region of memory reserved for the actual assembly code of the program. This memory may have restrictions on how it can be written to or accessed because frequent changes to an application's instruction are not expected. Because the size of instruction memory is known when the program compiles, this section of memory can be segmented by hardware, software, or a combination of the two.
- **Data Memory** is a region of memory where temporary variables, arrays, and information used by a program can be stored without using long term memory (such as a hard disk). This section of memory is allocated during the course of the program when more memory for data structures is needed.
- **Heap Memory** is an internal memory pool that tasks dynamically allocate as needed. As functions call other functions, it is necessary that the new (callee) function's data be loaded into the CPU. The previous (caller) function's data must be stored in the heap memory so

that it may be restored when the callee function is finished executing. The deeper function calls go, the larger the heap portion of memory needs to be.

Often, the heap memory and the data memory compete directly for space while the program is running. This is because both the depth of the function calls and the size of the data memory can fluctuate based upon the situation. This is why it is important to return the heap memory the task uses to the memory pool when the task finishes.

Memory Allocation in Languages

The organization of memory can vary among compilers and programming languages. In most cases, the goal of memory management systems is to make the limited resource of memory appear infinite (or at least more abundant) than it really is. The goal is to free the application programmer from having to worry about where his memory will come from. In the oldest days of mainframes, when each byte of memory was precious, a programmer might account each address in memory himself to ensure that there was enough room for the instructions, heap, and data. As programming languages and compilers were developed, algorithms to handle this task were developed so that the computer could handle its own memory issues.

Today, even assembly programmers do not have to worry about memory allocation because current assemblers can handle that task. Memory allocation algorithms are good enough at their job that it isn't worth a programmer's time to manually allocate memory. There are a few different ways that languages solve the problem of memory allocation. In general, it is simply a matter of providing the programmer with memory that is known to be required at compile time including space for global data values and the code itself. The more difficult problem is how to provide flexible data memory that may or may not be needed when the program actually executes.

The approach that C takes is to make available to the the programmer special functions that manage memory allocation. These methods are called

`malloc(int)` (memory allocate) and `free(void *)`. The basic idea is that whenever the programmer needs a specific amount of additional memory, he calls `malloc(int)` with the integer being the number of bytes of memory needed. The function returns a pointer to a block of memory of the requested size. When the programmer is done with a particular block of memory, he may call `free(void*)` to let the memory management library know that the particular block of memory isn't needed anymore by passing the pointer to that block to the function. If the programmer is diligent about returning (freeing) memory that isn't needed anymore, then he will enjoy an abundant supply of memory without having to count individual bytes. On the other hand, if a programmer repeatedly requests memory but does not free the memory back to the system, the memory allocator will eventually run out of memory and program will then crash. Thus, it is essential for passages of code that frequently request memory allocations to free these allocations as quickly as they can. Un-freed memory blocks are not fatal in very infrequently executed parts of code; however, the longer a program runs, the more potential there is for a problem. In general, a program that allocates but does not free memory, is said to have a **memory leak**.

Other languages handle the problem of memory allocation automatically. Java allocates the memory for new data on the fly using the keyword `new` instead of the function `malloc`, but the more important difference is that freeing takes place automatically. Part of the Java system called the **garbage collector** detects memory that can be safely freed and does so. In this manner, Java programs do not suffer memory leaks in the way a C program might.

Memory and the MSP

In the ez430 there is no inherent difference between instruction memory, data memory, and heap memory. The only subdivisions in memory are the blocks of flash and the sections of RAM. Any of these sections can hold any type of memory; however, because it is problematic to erase and rewrite flash in the middle of program execution, the flash memory is best saved for instructions and constants. The remaining RAM must be shared then between the heap, the dynamically allocated memory, and the global

variables. On the ez430, there is only 2KB of RAM, so no memory leaks are tolerable.

How memory is wasted or conserved

The most notable way to waste memory, memory leaks, have already been discussed, but there are several others. While memory leaks abuse the dynamically allocated portion of data memory, many layers of function calls abuse the heap. Above, it was explained that each time a function calls another function, the caller's registers and data are moved onto the heap. If each called function calls another function in turn, then the heap portion of the memory will grow significantly. For high power computing systems, this is not usually a great threat to the overall supply of memory compared to other memory leaks. Embedded systems, however, must avoid deep layers of function calling or risk exhausting the overall supply of memory.

There is a programming technique called recursion, which uses deep layers of function calling, where a function calls itself repeatedly on progressively smaller or simpler versions of the data until the answer is trivial (a base case). While this technique leads to some very clever solutions to some complex problems, it is uses large amounts of memory to achieve this end. Therefore, recursion is generally a poor choice when dealing with microcontrollers.

Another way to waste memory is to create too many global variables. Specifically, variables whose scope could be local to a function or that could be allocated dynamically waste memory because they take up space even when not in use.

5.2 - Improving Speed and Performance

So far in this course, programming assignments have focused on functionality. In most applications of embedded programming, speed and power performance are equally important. Long battery life is won through judicious hardware and software design. Skillful programming will allow the same job to be done with cheaper parts to improve the bottom line. This lab will introduce the basic concepts behind power and speed performance improvement.

Speed Performance

It is well known from the consumer PC market that the speed of computers can be measured in hertz. It is less well known that the frequency of the computer's processor does not adequately indicate a computer's performance or even the performance of the processor itself. By using the ez430, the choice of processor and maximum speed has been made, but the question of speed is a different one in embedded programming. While the dominant paradigm in consumer personal computing is to increase the performance of the computer to allow the system to do more with each generation, embedded processors are chosen to be able to perform specific tasks. The cheapest processor that can meet the specifications for the design will be chosen. While the issue and business conditions make the situation much more complicated than just price, the pressure is still toward choosing a part with **less** performance, not more.

In order to improve the performance of a software application, it is necessary to understand the way performance is measured. Measuring performance between platforms and software packages is a problematic endeavor; improving the performance of a single program on a single platform is much simpler. Although a detailed explanation of the nuances of performance measurement in computing is beyond the scope of this lab, a simple way to gauge the amount of time a program will take to perform a task is to count the number of processor cycles that the code will take. On the MSP430, each CPU instruction, jump, and interrupt takes a fixed number of cycles as explained in the MSP430 User’s Guide. Taking into

account branching, function calls, and interrupts the assembly code of a program can be used to calculate the time needed for a section of code.

Performance Tips

As mentioned above, embedded programming has different priorities from personal computing. Because the embedded programmer is usually trying to accomplish a specific task within a certain amount of time, the most important test of performance is whether the program is performing calculations on the inputs as fast as the inputs can enter the system. The goal is to make applications "real time." When the first draft of a program is unable to keep up with the required sampling, it is necessary to reduce execution time. Often, changing the hardware configuration is not easily doable; and software speed gains are usually more cost effective.

There are many approaches to improving speed performance. Incredible amounts of research go into new algorithms for common problems to improve the way that problem is solved. However, simply eliminating unnecessary instructions is a good start to improving performance. Test code left in a final version, any unnecessary instructions in a loop, and can all significantly increase the time in a section of code.

- In C, unnecessary code takes the form of too many method calls inside of a loop (because each method call adds a layer to the heap). While this is only a slight efficiency loss for code that is only executed once per sample, the loss can be very damaging if multiplied by a large loop. When trying to reduce execution time, it is best to start with the regions of the code where the processor spends the most time. Parts of the program that are only executed rarely have only a small effect on the speed compared to a loop that might run 100 times per sample. For example, if something can be done once, outside of the loop, do not do it many times inside of the loop.
- Remember to make judicious use of timers and other instruction saving interrupts. The timer interrupts allow the processor to periodically check on the status of the program without the use of slow `while(1)` or `for(;;)` loops (polling). However, for correct program behavior, it is important to do the minimum possible work in

an interrupt. This is most important with interrupts that happen frequently because the control flow of the program can be thrown off when interrupts happen faster than the system can handle. If the same interrupt occurs a second time before the first occurrence of the interrupt has completed, program behavior is much more difficult to control (essentially we have a recursive interrupt call). It is much easier to simply ensure that the interrupt is short enough to avoid the danger all together.

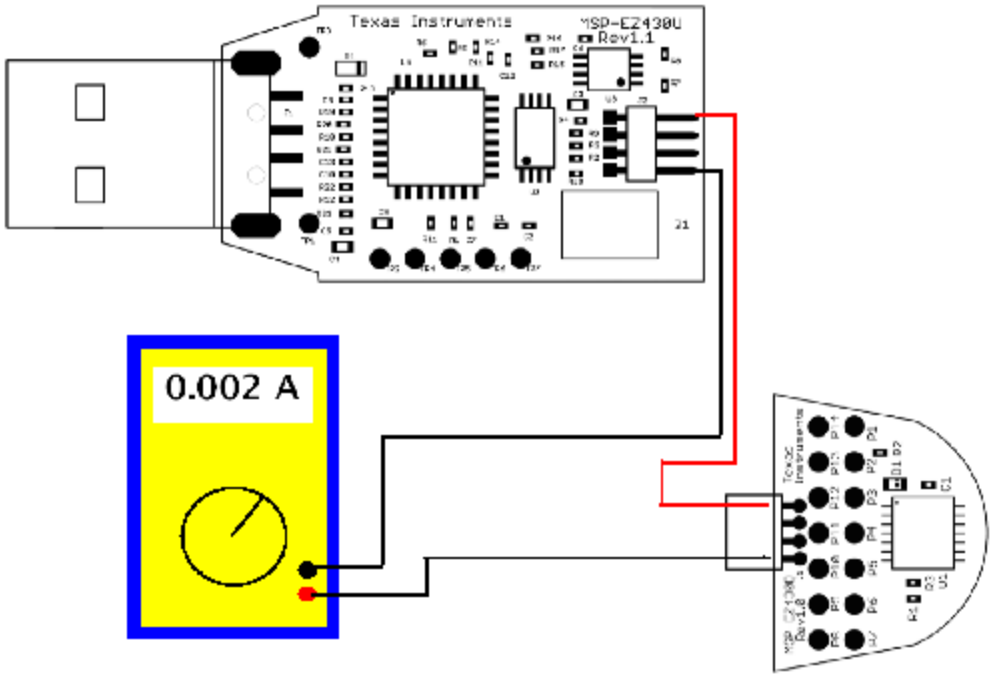
- Also, avoid recalculating values. If a piece of information is reusable, save it rather than recalculating it to save time. Sometimes memory is so scarce that this may not be possible.
- Don’t output to the console while debugging unless you absolutely must. Program flow is hampered immensley and program behavior might not reflect the behavior without the debug statement. Use breakpoints instead because the execution of the program is paused and the processor does not have to use any extra resources.
- Don’t leave legacy code from previous revisions. If you believe you may no longer need a part of the program, comment it out and note what you did in the comments. Even seemingly innocuous statements here and there in your code can slow overall performance.

5.3 - Reducing Power Consumption

One of the most important quality standards for battery powered devices is battery life. Handheld medical tools, electricity meters, personal digital assistants, and a goal of the designer and programmer is to lower the power use of the embedded system to negligible levels. This portion of the lab will give an overview of how power can be conserved using hardware and software. In designing battery powered devices, savings can be gained from the choice of electronic components, the arrangement of components, and the software on the design. The exercises will integrate the low power modes of the MSP into existing labs, so that examples of software power savings can be shown.

Measuring power on the ez430

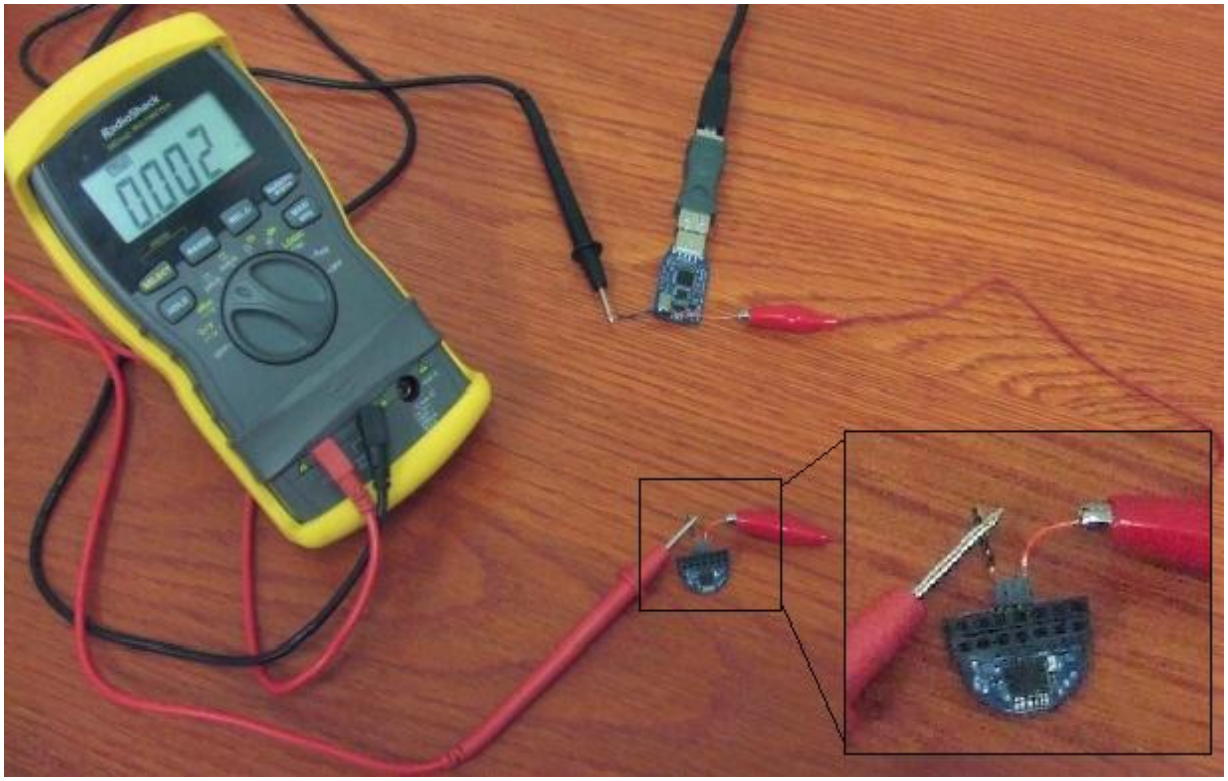
Measuring the current consumption of the ez430 is a slightly tricky task. To do so we must insert an ammeter in serial with the ground pin between the Spy-Bi-Wire programming interface and the target board. Since the program is loaded into flash on the tool, we can run our code, cease debugging, and the tool will run our program whenever it is powered. Thus, the following diagram shows how we can insert an ammeter in serial with the ez430 while the program runs even though we cannot debug it at this point (the two middle data pins aren't connected).
Connection Diagram



Connecting the Ammeter in serial as shown will allow for you to measure current consumption.

The following is what it would look like if we actually connected everything together. (Notice that I have added sockets (DigiKey Product ID: **********) to the "fully accessible" pins to make them more fully accessible.

Connection Picture



Here, I have created the setup using parts from around the lab.

Shutting off parts in general

Most parts have a shutdown or sleep mode available that will reduce the current consumption of the component considerably. In general, digital parts consume significant current when their transistors switch because of the charging and discharging of the internal capacitances of the transistors. Analog integrated circuits also support shutdown modes to reduce power consumption. Datasheets will specify the current consumption in both on and shutdown modes of the component. It is important to note that when a device is in shutdown mode, power and ground voltages are still powered and connected to the device.

In order to shutdown most integrated circuits, all that is required is a shutdown or sleep pin to be asserted properly. Other devices require a shutdown command to be issued over the bus. The primary disadvantages

of shutdown modes, apart from the fact that the device is inoperative is that recovering back into normal operating modes can impose a significant delay. A useful property of the MSP is that its recovery time from some low-power modes is fast enough to meet the response times of interrupts.

Parts without built-in shutdown modes must be shutdown by having its current supply controlled through a transistor or other switching device.

Using the Low Power Modes

The MSP430 was designed with the low power modes in mind from its beginnings. In lower power mode, the processor can achieve current in the microamps while still monitoring its inputs. The principles of utilizing the MSP power modes are described in detail in the second chapter of the MSP User's Guide. The modes vary the degree to which the processor is aware of its surroundings and the clocks that the processor keeps running. The processor lowers power consumption partly by shutting off external and internal oscillators.

There are four low power modes in addition to regular operating mode on the MSP430:

- Active Mode is the fully powered mode when the processor executes code and all clocks and peripherals are active. The chip consumes about 340 µA with 1 MHz clock at 3.3V in this mode.
- Low Power Mode 1 (LPM1) disables the CPU and MCLK while leaving the ACLK and SMCLK enabled. This allows timers, peripherals, and analog systems to continue operation while dropping current consumption to about 70 µA with 1MHz clock at 3.3V. Because the timers and other internal interrupt systems still operate, the processor will be able to wake itself.
- Low Power Mode 2 (LPM2) disables the CPU, MCLK, and the DCO are disabled but the SMCLK and ACLK are active. The DC is disabled if the DCO is not used for MCLK or SMCLK in active mode. Internal interrupts can still operate. Current consumption drops to about 17 µA.
- Low Power Mode 3 (LPM3) disables the CPU, MCLK, SMCLK, and DCO. The DC and ACLK remain active. This allows some peripherals

and internal interrupts to continue. Current consumption drops to about 2 µA.
- Low Power Mode 4 (LPM4) Current consumption drops to about .1 µA, but all clocks and the CPU are disabled. This prevents any of the on-chip modules from operating, and only off-chip interrupts can wake the device.

To enter a low power mode the status register in the CPU must be set to indicate the desired mode. Specifically the bits SCG1, SCG0, OSCOFF, and CPUOFF. The User's Guide details the specific bits needed. Also provided in the chapter is some example code on changing power modes. To exit low power mode, an interrupt is needed. In the interrupt, the previous status register state can be altered so that exiting the interrupt will leave the processor awake. The User's Guide explains in detail the specifics of entering and leaving low power mode. Example code with the compiler also demonstrates the low power modes.

Principles of low power operation on the MSP

The User's Guide for the MSP also explains the principles needed to lower the power consumption of a design. Be sure to minimize wasteful code execution. This is the same idea as improving speed performance because every unnecessary instruction wastes a little bit of battery power. All of the techniques that improve code efficiency will improve power efficiency. Increasing clock speed will not yield similar power savings because faster execution increases power consumption. Similarly, unused peripheral modules on the processor should be de-activated to save power. Use interrupts to handle events to allow the processor to stay in Low Power Mode 3 as much as possible. By reducing the awake time of the processor, the average current consumption of the MSP can be reduced to levels approximately as low as LPM3 while maintaining the same functionality.

5.4 - Lab 5: Optimization and Low Power Modes

Low Power Modes and Code Optimization

Exercise:

Problem:
Fibonacci Optimization

The **Reducing Power Consumption** module discusses why it is important to keep power in mind when programming embedded devices. We have yet to consider this while programming the previous labs. Writing efficient code is the first step in improving power consumption, next we can disable all parts of the board that aren't currently being used.

Take the following piece of code: `long fibo(int n) { if (n < 2) return n; else return fibo(n-1) + fibo(n-2); }` It recursively calculate the nth number in a Fibonacci sequence recursively. Recursion makes this piece of code easier to read, however, it is very inefficient and consumes far more memory than it has to. If you try to compute a large number, say `fibo(50)`, then it will take much longer and will consume more power than it should.

The original program is very inefficient and wastes memory in several of the ways described in the inefficient Memory Conservation module. Modify the code to eliminate the memory waste and improve the speed of the program. Note that there is a tradeoff between speed and memory (though at first the program is simply gratuitously wasteful). What is the nature of the tradeoff? Assuming the one addition takes one cycle to complete, how long would it take the original code to complete `fibo(50)`? How long would it take your new, improved version? Assume that you are only considering the addition operations.

Exercise:

Problem:
Low Power Modes

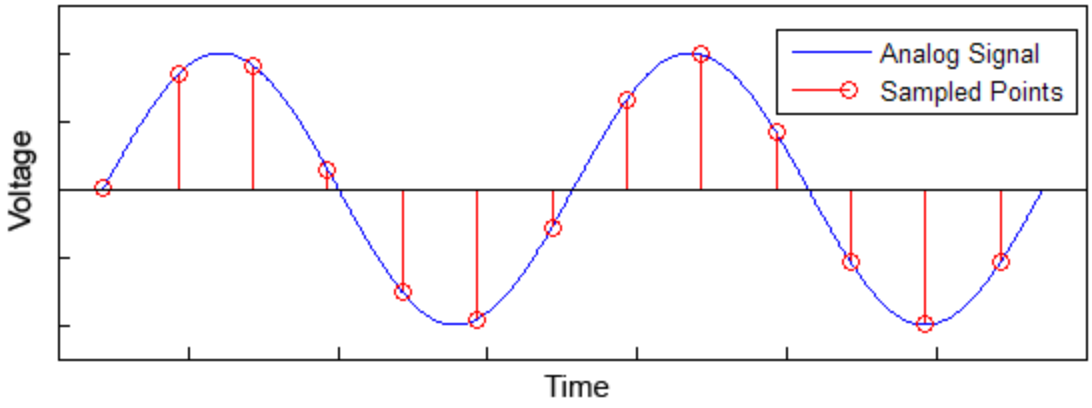
Modify your project so that the processor remains in one of the low power modes whenever it is not doing any calculations. Wake up from the low power mode using a timer interrupt (change the timer settings so that there is a substantial time period between interrupts) and have your program compute `fibo(50)`. You may want to make the program compute fibo(50) more than once so that the MSP is in the ISR performing an intensive task for longer and thus make it easier to read the current consumption value. Output the result to the standard out display. What is the result? (Hint: 12,586,269,025) Make sure the result is correct number. As soon as the calculation is done, return to low power mode. Perform the same process to calculate the current consumption during the use of your Fibonacci function. You should have measured three current consumption values: 1) in the low power mode, 2) while processing `fibo(50)`, and 3) while processing the 50th Fibonacci number using your function.

Note: A number must be small enough to fit in its given type. If it is too large, you may get unpredictable results. Try using a `long long` for extra huge numbers. Because of our `printf` settings, we cannot output such large data types. You must use bit-wise operations to separate the number into smaller chunks suitable for printing.

From now on, instead of `while(1)` at the end of programs, simply place the tool into a low power mode. This will allow future applications to more closely resemble and operate as real-world embedded programming.

7.1 - Introduction to Sampling

Sampling refers to the process of converting a continuous, analog signal to discrete digital numbers. Typically, an **Analog to Digital Converter (ADC)** would be used to convert voltages to a digital number corresponding to a certain voltage level.



This shows the way that a given analog signal might be sampled. The frequency at which the signal is sampled is known as the **sampling rate**.

Resolution

The number of bits used to represent a sampled, analog signal is known as the resolution of the converter. This number is also related to the total number of unique digital values that can be used to represent a signal.

For example, if a given ADC has a resolution of 12 bits, then it can represent 4,096 discrete values, since 2^12 = 4,096; if the resolution is 16 bits, it can represent 65,536 discrete values.

We may also think about resolution from an electrical standpoint, which is expressed in volts. In that case, the resolution the ADC is equal to the entire range of possible voltage measurements divided by the number of quantization levels. Voltage levels that fall outside the ADC's possible

measurement range will saturate the ADC. They will be sampled at the highest or lowest possible level the ADC can represent.

For example, ADC specifications could be as follows:

- Full scale measurement range: -5 to 5 volts
- ADC resolution 12 bits: $2^{12} = 4{,}096$ quantization levels
- ADC voltage resolution is: $\frac{5\text{V} - -5\text{V}}{4096} = 0.0024\ \text{V} = 2.4\ \text{mV}$

Large ranges of voltages will fall into in a single quantization level, so it is beneficial to increase the resolution of the ADC in order to make the levels smaller. The accuracy of an ADC is strongly correlated with its resolution however; it is ultimately determined by the **Signal to Noise Ratio** (SNR) of the signal. If the noise is much greater relative to the strength in the signal, then it doesn't really matter how good or bad the ADC is. In general, adding 1 more bit of resolution is equal to a 6 dB gain in SNR.

Sampling Rate

Analog signals are continuous in time. In order to convert them into their digital representation we must sampled them at discrete intervals in time. The interval at which the signal is captured is known as the **sampling rate** of the converter.

If the sampling rate is fast enough, then the stored, sampled data points may be used to reconstruct the original signal exactly from the discrete data by interpolating the data points. Ultimately, the accuracy of the reconstructed signal is limited by the quantization error, and is only possible if the sampling rate is higher than twice the highest frequency of the signal. This is the basis for the **Shannon-Nyquist Sampling Theorem**. If the signal is not sampled at baseband then it must be sampled at greater than twice the bandwidth.

Aliasing will occur if an input signal has a higher frequency than the sampling rate. The frequency of an aliased signal is the difference between the signal's frequency and the sampling rate. For example, a 5 kHz signal sampled at 2 kHz will result in a 3 kHz. This can be easily avoided by

adding a low pass filter that removes all frequency higher than the sampling rate.

7.2 - Analog-to-Digital Converter on the MSP430

The analog to digital converter (ADC) on the ez430 is a type called a **Sigma-Delta (SD)** Converter. The way it operates is slightly different from what was described in the previous section (although the end result is the same) but those specifics are out of the scope of this course. The SD converter on the ez430 has 8 channels and a 16 bit resolution. The module is highly configurable and can run largely free of program involvement. In this portion of the lab, we will broadly explain the features of the module, but the particular effects of each register are listed, as usual, in **Chapter 12** of the **User's Guide**.

Range of Measurement

The result of each conversion will be 16 bits long in the form of an unsigned integer whose value is:
Equation:

$$\text{SD16\_MEMx} = (65{,}536)\frac{\text{Vin} - \text{Vrneg}}{\text{Vrpos} - \text{Vrneg}}$$

Where **Vin** is the input voltage to be measured, **Vrneg** is the lower reference voltage, and **Vrpos** is the higher reference voltage. The reference voltages are set to power and ground by default, but they can be changed to an internal reference generator or an externally supplied reference using the `SD16CTL` register.

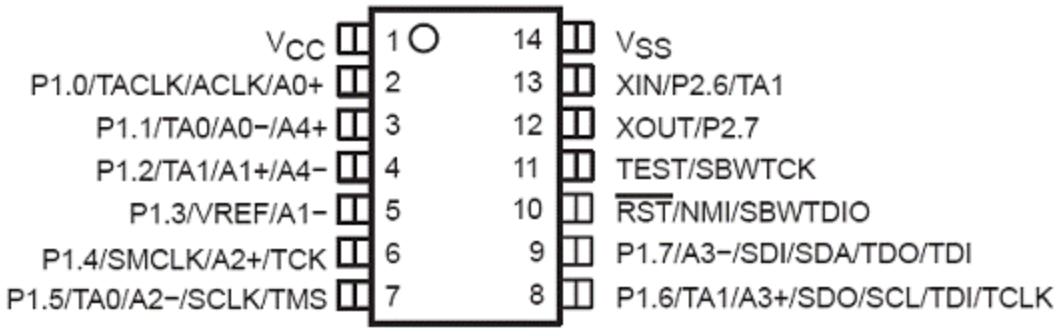
Input Channels for the ADC

The following is a table of the 8 input channels on the ez430. "Analog signal input" channels will take any analog source (we will get to the pin mapping's shortly). Other channels perform specific tool integrated tasks. For example, channel A6 is an integrated temperature sensor. For detailed descriptions of the others, see the **User's Guide**.
SD16 Input Channels

1. **A0**: Analog signal input
2. **A1**: Analog signal input
3. **A2**: Analog signal input
4. **A3**: Analog signal input
5. **A4**: Analog signal input
6. **A5**: $\frac{Vcc-Vss}{11}$
7. **A6**: Integrated temperature sensor
8. **A7**: Short for PGA offset measurement

Here is a pin diagram corresponding to the 14 accessible pins showing which pins correspond to which input channels. Notice that each channel has a "+" and a "-" input. Although with other ADC's, we would simply have one input pin and connect the other to the ground, the SD converter requires us to connect the positive and negative inputs to the corresponding input channels. Note that the numbers from 1-14 on the chip correspond to pins 1-14 on the target board.

Pin Map



Shows which pins correspond to specific SD converter inputs. Also shows which pins correspond to specific ports (i.e pin 4 is connected to P1.2).

Operation Reminders for the ADC

Remember the following when attempting to use the ADC:

- Be sure to enable SD16 interrupts (on `SD16CCTL0`) and to select the specific channel which you are using (on `SD16INCTL`)
- After configuring the ADC, you must enable the `SD16SC` bit to start conversion.
- All ADC values will be stored in the `SD16MEMx` variable where "x" is the number of the channel

The **User's Guide** will be very useful for this lab because of the complexity of this part of the MSP430. Be sure to go over the chapter atleast briefly before jumping into programming.

7.3 - Lab 7: The ADC

Exercise:

Problem:

Background Questions

1. The ez430 has a 16 bit SD Converter. The number of bits used by the ADC is known as its resolution. What are the number of possible values that the 16 bit SD Converter can represent?
2. Extreme voltages, either too high or too low, cannot be measured correctly. What is the range of analog voltages that can be accurately represented on the ez430? You may want to check the **User's Guide** or experiment with the hardware.
3. In the real world, signals are polluted by "noise" that alters the quality of the original signal. Signal to Noise Ratio, SNR, is often used as a measure of the quality of a signal. Before a signal is sampled through the ADC, it is helpful to condition the signal in order to improve its SNR. What can be done to condition the signal? Where would it be ideal to condition it and why? (i.e. at the ADC, near the source, at the processor?)
4. The Nyquist Theorem states that a signal must be sampled at least twice as fast as the highest frequency in order to be decoded without error. The minimum sampling frequency is therefore known as the Nyquist Sampling Rate. Typical, audio CDs are sampled at a rate of 44.1 kHz, and can be decoded to 65,536 unique voltage levels. What is the highest frequency that can be represented without error? How many bits per sample are used? What is the total data rate for a stereo CD?

Exercise:

Problem:

ADC Setup

1. Figure out what the following codes is doing. Set up the hardware so that it functions correctly, and comment each line of code. What is the code's function? `#include "msp430x20x3.h"`
   `void main(void) { WDTCTL |= WDTPW + WDTHOLD;`

```
DCOCTL = CALDCO_8MHZ; BCSCTL1 = CALBC1_8MHZ;
P1DIR |= 0x01; SD16CTL |= SD16REFON +
SD16SSEL_1; SD16INCTL0 |= SD16INCH_6;
SD16CCTL0 |= SD16SC; while (1) { while
((SD16IV & SD16IFG)==0); if (SD16MEM6 >=
0xD6D8) P1OUT |= 0x01; else P1OUT &= ~0x01;
_NOP(); } }
```

Note: Most modern compilers intended for use with embedded processors, such as the **IAR Workbench**, allow the user to check the status of the registers while the program is halted. This is extremely helpful in debugging code. For example, if the program is halted with a `_NOP()` after a sample is taken from the ADC, the user may check the `SD16MEMx` register (by setting a breakpoint at the _NOP() command) to see the new value that has been stored. If a value has changed since the last time the processor was halted, it will turn red in the watch window.

2. Create a version of the program that is completely interrupt driven. The original program uses a while-loop to poll for the interrupt flag. What is the sampling rate? Remember to put the processor into a low power mode and enable interrupts after configuring the SD Converter as discussed in **Lab 5**

Exercise:

Problem:
LED Dimmer

Add the file containing the LED dimmer function you wrote in **Lab 4** to a new project in which you configure the SD Converter. Attach a Photo Diode (DigiKey# ********), a diode that outputs a voltage depending on the ammount of light that hits it, to any one of the SD16 inputs. Input the SD16MEMx value into the LED dimmer function in such a way that the LED's brigtness is indirectly propotional to the ammount of light that hits the Photo Diode. In other words, if we block the light to the diode, the LED should appear brighter; if we shine light on the diode, the LED should appear dimmer. If you do not have access to a Photo Diode, simply set the input to the integrated temperature sensor and perform the same exercise.

Exercise:

Problem:
Flash Storage

Now, chose any analog source (temperature sensor, Photo Diode, function generator etc.) on any input channel and store the values in flash using what you learned in **Lab 6**. After a specific number (that you select) of samples are stored, have your program output the values using the `printf` with the help of a `for` or `while` loop. Then take the values and plot them on a graph. You may want to store every 10th (or 20th or 50th etc.) sample in flash to be able to see the analog signal over a longer period of time.