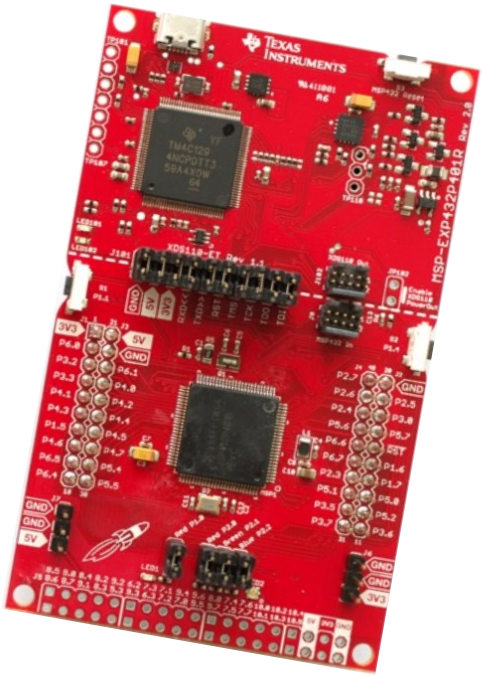
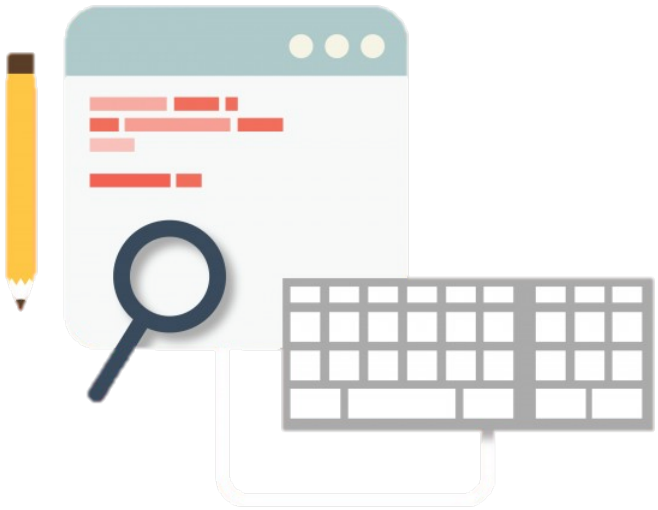
CPE 329
Laboratory Manual



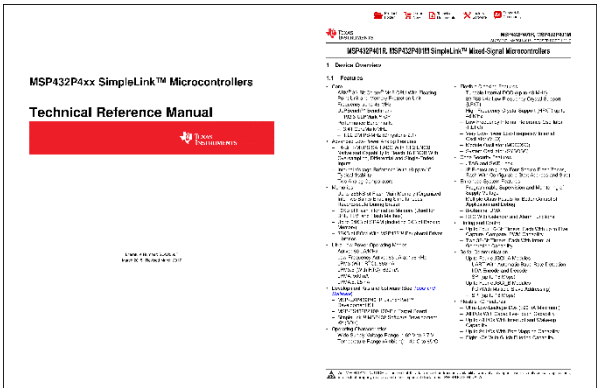
Jeff Gerfen and Paul Hummel

Cal Poly San Luis Obispo
Fall 2017

Assignments



A1 - Datasheet Fun



Background Information:

Please acquaint yourself with the MSP432 Quick Star Guide (QSD), MSP432 Datasheet (MDS), MSP432 Technical Reference Manual (TRM), and LaunchPad User's Guide (LUG) on PolyLearn. Although the User's Guide and Quick Start Guide contain a variety of handy information such as pinouts, schematics, board layout, etc., the Technical Reference Manual and Datasheet provide a wealth of detailed information regarding how to use the MSP 432 from a programming standpoint. *The MSP432 Technical Reference Manual will be your go-to document for much of the work in this course.*

Instructions:

1. Gather the MDS and TRM for the MSP432 from Texas Instruments. (While these are available on Polylearn, it could be worthwhile trying to find these documents directly from Texas Instruments to learn how to find datasheets in future)
2. Review the datasheets to get an idea of their organization and what data they each contain. Note that using a PDF reader that can show bookmarks or table of contents can make navigating these documents easier.
3. Answer the questions below.

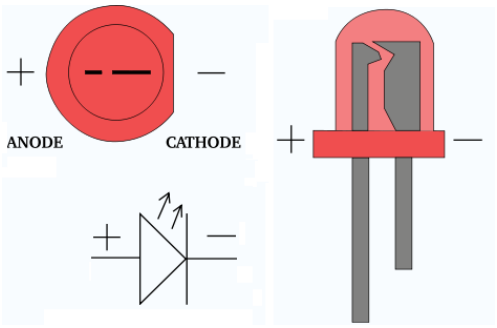
Questions:

1. Create a memory map for Code and Peripheral address spaces of the MSP432P401R.
2. How many internal oscillators does the MSP432 have?
3. How many timers does the MSP432P401R have? What size are the timers?
4. What is the maximum sampling rate of the analog to digital converter on the MSP432P401R?
5. What is the equation for determining the digital output of the analog to digital converter when operating in single-ended mode on the MSP432?
6. Which register is the primary mechanism for changing power modes on the MSP432?
7. When the temperature goes up, does the general I/O output current from the MSP432 go up or down?
8. The high drive I/O on the MSP432P401R produces more current by a factor of X. Estimate X according to the datasheet

Deliverables:

1. Single PDF listing each question and corresponding answer

A2 - Blinking LED, Clock Control, and Software Delay on MSP432



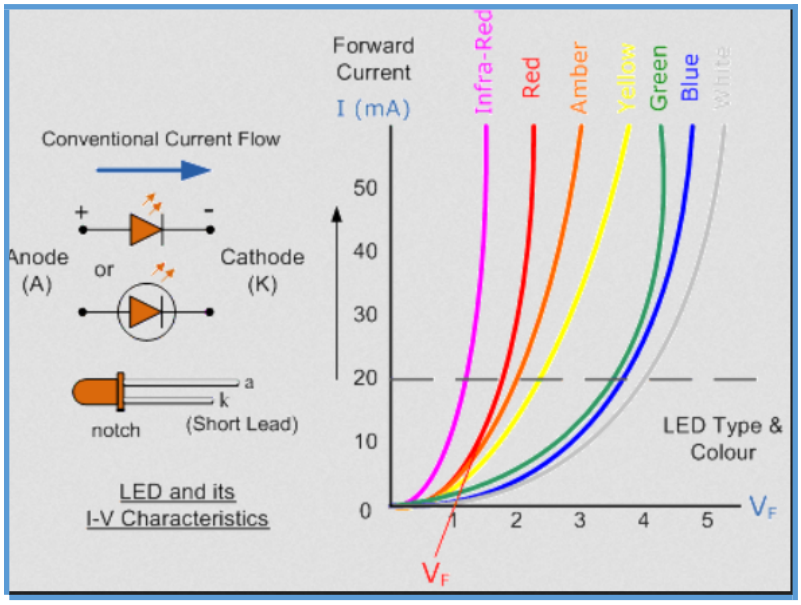
Background Information:

LaunchPad LEDs

The MSP432 has two built-in LEDs for development work – LED1 and LED2. LED1 is red and LED2 is a multicolor RGB LED of which any combination of the colors (red, green, blue) can be activated at a given time. Please find LED1 and LED2 in Figure 29 in the LaunchPad User's Guide and then trace them back to the MSP432 in Figure 28 of the same document.

Using External LEDs

External LEDs may also be wired to one of the output ports also. As with any LED, it is necessary to use a series current limiting resister to avoid excessive voltage across the LED and hence LED destruction. The series limiting resister will keep LED operation low on the curve, with resistor selection determining the exact location.



Setting MSP432 Digitally Controlled Oscillator (DCO) Frequency

The MSP432 contains a DCO which is software programmable to run at frequencies of 1.5 MHz, 3 MHz, 6, MHz, 12 MHz, 24 MHz, and 48 MHz. Details regarding the MSP432 clocking system and control of the DCO can be found in Chapter 5 of the MSP432 Technical Reference Manual (TRM). Briefly review the descriptions for the registers CSKEY and CSCTL0 in Chapter 5 of the MSP432 reference manual. Technical Note 3 in this lab

manual introduces the MSP432 clock system and provides some example code on how to change the clock frequency.

Instructions:

Create Delay Functions

1. Download program *A2_Blinking_LED_starter_code.c* from PolyLearn and get it running on your LaunchPad.

2. Modify the program to create two delay functions:

   `delay_ms()`

   `delay_us()`

   Each of these functions should take in the number of milliseconds or microseconds respectively as well as the system clock frequency as integers and provide the appropriate delay. Set your system up to utilize #defines as appropriate for representing things like system frequency. For example, your function calls should look something like this: `delay_ms (200, FREQ_12_MHz);` Note that your delay functions can return void or may optionally return -1 if the combination of the delay value and system frequency is unachievable.

Create a Function to Set MSP432 Frequency

3. Create a function to set the DCO frequency on the MSP432 such as `set_DCO().` The function `set_DCO()` should allow cause the MSP432 to run at any of the frequencies from 1.5 MHz to 48 MHz listed above. Again use the same defines so your function call should look something like `set_DCO(FREQ_12_MHz);`

4. Use an oscilloscope to confirm that that the functions written to delay and set the system clock operate properly. Ensure that you collect screen shots for a sampling delays and clock frequencies to include in your writeup. *Hint: You should probably start by using the default system speed of 3 MHz to create your delay functions. Once those are verified to work, you should be able to change the DCO clock without changing the delay functions to verify your system clock timing is changing properly.*

Validate Delay Accuracy

5. Demonstrate that you can use your functions to cause an LED to blink for one second on and one second off at a variety of clock frequencies. Confirm that you have correct results with an external timing device, e.g. a watch.

6. Utilize an external timing device, e.g. a stopwatch, to verify that your LED on and off times are accurate within 5%.

Generate and Observe Short Pulses

7. Use your above functioning code to generate two 100us pulses with a 100us space in between. These pulses should be non-repeating. Capture these pulses with an oscilloscope using normal triggering and confirm the accuracy of your delay functions.

8. Use an oscilloscope to determine the shortest pulse that can be accurately generated with your newly created clock control and delay functions.

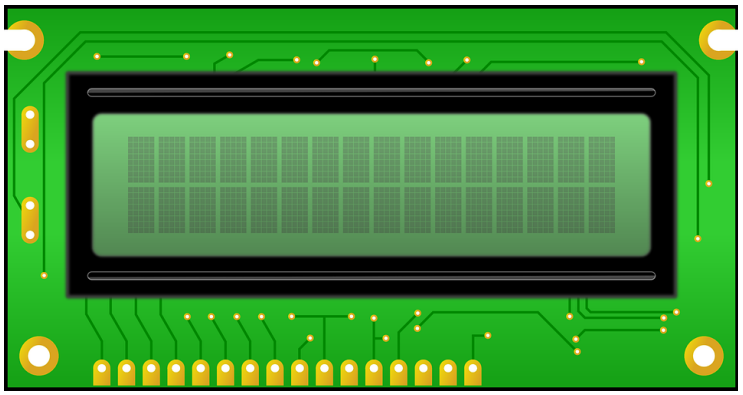
Building a Header File and Associated C file

9. Place the code that implements **`delay_us()`** and **`delay_ms()`** in a separate file such as **`delay.c`** with an accompanying **`delay.h`** , allowing easy inclusion in later projects in the course. Technical note 9, Header Files, provides details on how to accomplish this.

Deliverables

1. Create a simple pdf document (not a full lab report) containing the following:
    a. Demonstration of one-second pulse with a minimum of three clock frequencies via a YouTube video link.
    b. Documentation of accuracy of the one-second pulses with three different clock frequencies.
    c. Screen shot of the two 1us pulses using normal triggering. Confirm that the pulses are accurate to within 5% or better.
    d. Screen shot of the shortest pulse generated. Confirm accuracy of this pulse.
2. C-source code for your system. (*All submitted code should be well commented or self documented. Choose variable names for readability. Provide comment headers where appropriate)*

A3 - LCD Display



Instructions

1. Gather and review the data sheet for the LCD display.

Build the Circuit

2. Draw a wiring diagram for connecting the LCD to the MSP432 using 4-bit (nibble) mode. Make sure to draw and label the pins as organized on the MSP432 and LCD display. Note that most pins on the MSP432 are not organized sequentially in order. Be sure to include any necessary power connections.

3. Construct the circuit.

Code Design

4. Implement basic functionality required to write a character to a single location on the display, e.g. write a capital 'A' to the left-hand location on the first line.
5. Modularize your code, writing useful library functions to make later use of the LCD display easy to integrate. Minimum library functions to be written include:

```
Clear_LCD();          // clear the display
Home_LCD();           // move the cursor to the top left of the LCD
Write_char_LCD();     // write a character on the LCD
```

   You may utilize function parameters as required to implement these functions.

   You may consider writing functions such as `Write_string_LCD()`, which would write a string to a specified location on the LCD or any other function that seems handy.

Building a Header File and Associated C file

6. Place the code that implements these LCD drivers in a separate file such as **`LCD.c`** with an accompanying **`LCD.h`** , allowing easy inclusion in later projects in the course. Technical Note 9, Header Files, provides details on how to accomplish this.

Questions

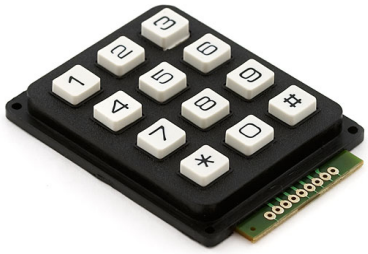
1. What is the minimum amount of time necessary for the LCD to start up? This is the amount of time from when the devices is powered up until a character can be written to it.

2. Draw a timing diagram for clearing the LCD and displaying the letter A in the home position

Deliverables:

1. Single PDF with YouTube demo, answers to questions, and C source code written.

A4 – Keypad Integration



Background Information:

Keypad Basics

A keypad consists of an array of switches connecting two sets of wires divided into rows and columns as shown in Figure 1 below. A button press causes a connection to be made between one of the rows (A,B,C,D) and one of the columns (one, two, three).

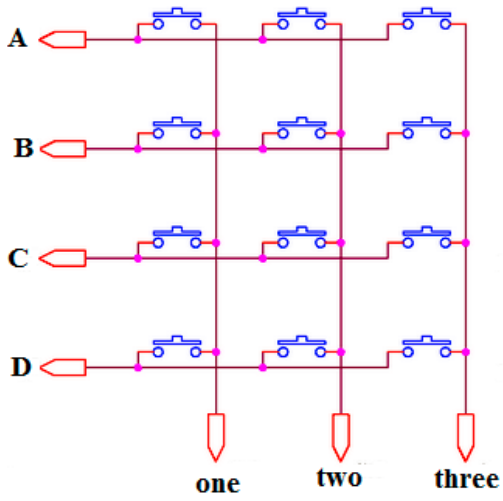


Figure 1: Keypad Schematic

Detecting Key Presses

Algorithms and approached to detecting key presses vary. One method for detecting key presses is to:

1. Place a pull-up resistor across each column wire. The pull-up resistor should tie to system supply of 3.3V for MSP432 applications.
2. Configure the rows as GPIO outputs and the columns and GPIO inputs.
3. Drive each row low (A, B, C, or D in the schematic above) one at a time, testing each column while a row is low to see if a low is read on that column.
4. The key pressed can be ascertained once it is known which row and column were low at the same time.

Instructions

1. Investigate the schematic diagram for the keypad and review the supplied code *A4_Keypad_starter_code.c* to ensure you understand how the keypad and code operate.
2. Modify the code as required so that it supports the 3x4 keypad in the lab kit rather than a 4x4 keypad it is designed for. Run this code and confirm that the LED2, the tri-color LED changes colors as you press buttons 1 through 7. Note that LED2 will not illuminate if other buttons are pressed.
3. Create a schematic to integrate the keypad and LCD on the MSP432 LaunchPad. Note that some pin assignments will need to be changed.
4. Modify software as required so that the LCD will show which button is pressed in the upper left position on the LCD.
5. Create a library that will allow the keypad to be used with other programs.

Deliverables

1. Create a simple pdf document (not a full lab report) containing the following:
    a. YouTube video demonstration of the keypad and LCD operating together.
    b. Schematic diagram of the system.
    c. C-source code for your system. (*All submitted code should be well commented or self documented. Choose variable names for readability. Provide comment headers where appropriate)*

A5 - Interrupts and Timers

Instructions

25 KHz Clock with 25% Duty Cycle

1. Load the Code Composer Studio demo project titled ***msp432p401x_ta0_01*** into your project workspace. Run it as-is on your LaunchPad development board and observe the LED blinking under default conditions.
2. Create a 25 KHz clock with a 25% duty cycle using DCO = 24 MHz. Observe this clock on an oscilloscope and take a screen capture of it. Please show the arithmetic used to calculate period, time high, time low, CCRO values, etc. The ISR should explicitly set the LED output bit high or low based on detecting the current output value rather than simply toggling the bit using **^=** C syntax. Simply toggling a bit creates an opportunity to lose track of the state of the bit and hence end up with a situation such as an inverted waveform. As always, ensure that your setting of port direction, output, etc. avoids affecting other bits than the desired bit to be changed. Take a scope capture of this clock.

ISR Processing Measurement

3. Change your code from step #2) above to have a simple ISR which generates a 50% duty cycle square wave. Add a second GPIO bit output to your system to measure ISR processing time – this bit will be driven high upon entering the ISR and driven low when leaving the ISR. Bring MCLK out of the MSP432 for viewing on the oscilloscope. You may bring MCLK out on P4.3 as shown below. Consult the data sheet for proper port selection settings.

Table 4-1. Pin Attributes (continued)

PIN NO.[1]			SIGNAL NAME[2] [3]	SIGNAL TYPE[4]	BUFFER TYPE[5]	POWER SOURCE[6]	RESET STATE AFTER POR[7]
PZ	ZXH	RGC					
56	H9	N/A	P4.0 (RD)	I/O	LVCMOS	DVCC	OFF
			A13	I	Analog	DVCC	N/A
57	H8	N/A	P4.1 (RD)	I/O	LVCMOS	DVCC	OFF
			A12	I	Analog	DVCC	N/A
58	G7	33	P4.2 (RD)	I/O	LVCMOS	DVCC	OFF
			ACLK	O	LVCMOS	DVCC	N/A
			TA2CLK	I	LVCMOS	DVCC	N/A
			A11	I	Analog	DVCC	N/A
59	G8	34	P4.3 (RD)	I/O	LVCMOS	DVCC	OFF
			MCLK	O	LVCMOS	DVCC	N/A
			RTCCLK	O	LVCMOS	DVCC	N/A
			A10	I	Analog	DVCC	N/A

4. Watch this bit on the oscilloscope while watching SMCLK on another trace to determine how many SMCLK cycles it takes to execute your ISR. How many clock cycles does it take to execute? Take a scope capture of your ISR execution measurement pulse on channel A and MCLK on channel B.

Shortest Pulse and Failing ISR

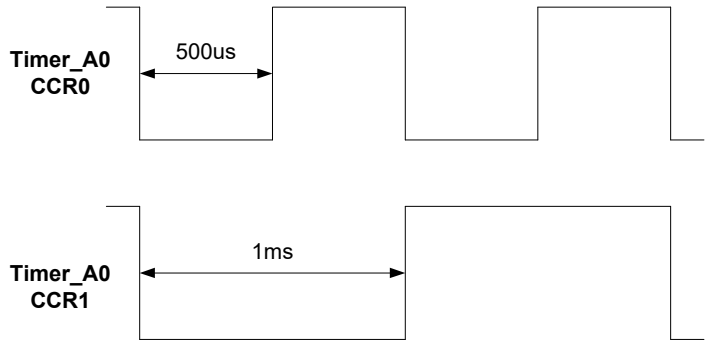
5. Keep DC0 at 24 MHz. Lower the amount you increment CCRO to a small number, e.g. less than 20, to determine where your timer fails to generate a pulse width proportional to the CCR0 value. Record the smallest value of CCR0 that works and take a scope capture of this. Does this value of CCR0 correlate to the amount of clocks that the ISR requires to execute? Describe your observations in your A4 writeup.

Clock with 50% Duty Cycle and 20-Second Period

6. Set your Launchpad DCO for MCLK = 1.5 MHz. Use a timer to create a clock with period of 20 seconds. Note that you will have to create a timer with a shorter period, e.g. 100 ms, and count these events to toggle an output port bit driving an LED after a predetermined number of clock events have occurred. Confirm your timer operation with a stopwatch and make a YouTube demo which shows the stopwatch display and a toggling LED.

2-Bit Counter

7. Set DCO to 1.5 MHz. Use Timer_A0 with CCR0 and CCR1 to establish timing to create 2-bit counter with the LSB having frequency of 1,000 Hz and the MSB having a frequency of 500 Hz. The resulting Counter should look something like this:



8. Take a scope capture which shows both bits to see a complete count from 00 to 11. Take a second scope capture which captures the transitions as both waveforms transition from 1 to zero. Ensure this screen shot is zoomed in sufficiently to see any delays between one waveform and the other. What do you observe?

Extra Credit Reflex Game (20%)

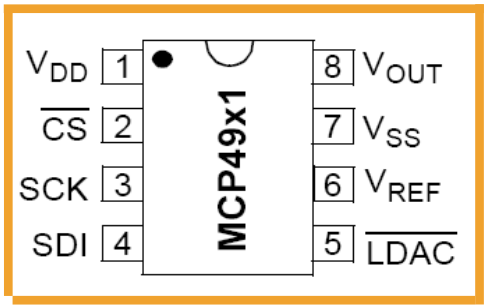
9. Create an electronic timing game called ***Reflex*** which measures the amount of time it takes between presses of two different buttons and displays the results in milliseconds on the LCD screen. The object of this game is for one person to press button #1 and then the other person to press button #2 as quickly as possible to test their reflexes. This game should automatically repeat after each play of the game. Demonstrate ***Reflex*** with a YouTube video.

Deliverables

1. A single pdf document with
    a. Links to your single YouTube demo video
    b. Clock calculations arithmetic
    c. Oscilloscope screen shots
        i. 25 KHz clock (step 2)
        ii. ISR execution and MCLK (step 4), annotated to demonstrate how many clock cycles for ISR execution to measure a bit
        iii. Shortest pulse (step 5)
        iv. Two screen shots of the 2-bit counter (step 8)
    d. YouTube video of 20-second clock
    e. Properly formatted code of:
        i. main.c with ISR for 25 KHz clock system
        ii. main.c with ISR for 10-second period clock system
        iii. main.c with 2-bit counter generation
    f. Your observations
2. Zipped project directory for Reflex Game (if doing extra credit)

**Properly formatted implies fixed-width font adjusted so lines don't wrap, excess white space removed for compact presentation, following accepted code formatting standards, header at top of each new file to assist reader quickly interpreting what they are looking at, etc.*

A6 - Assignment DAC Waveform Generation

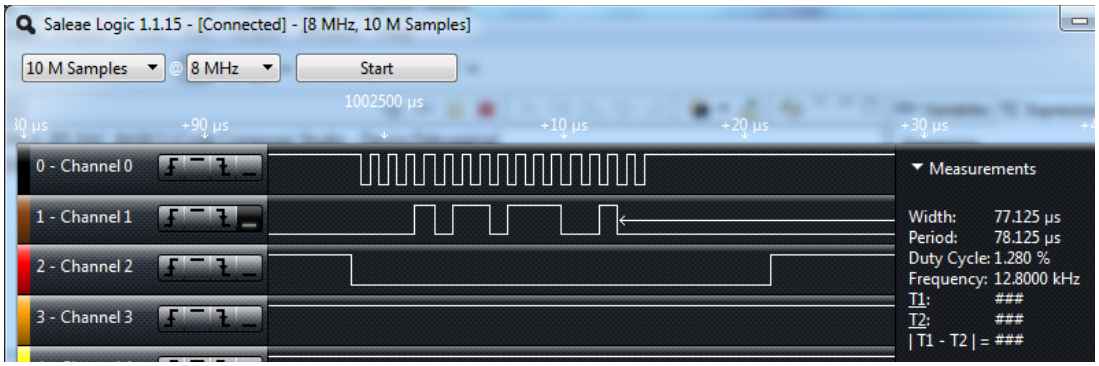


Instructions

Background Review

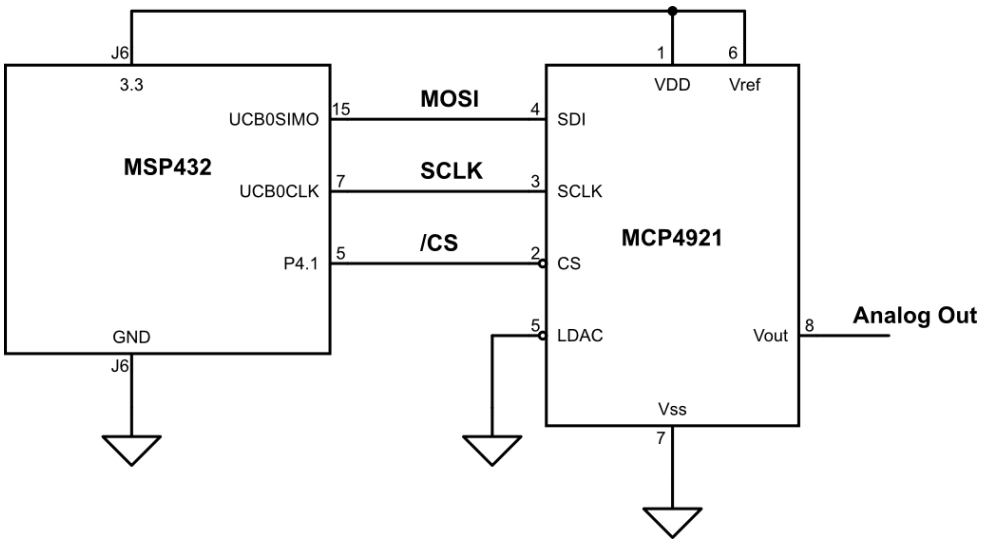
1. Review the MCP4921 DAC datasheet and relevant sections of the MSP 432 Technical Reference Manual.

   The following screen shot represents one command to the SPI DAC. Channel 0 is the clock, Channel 1 is data, and Channel 2 is chip enable.



Interface to the 4921 DAC

2. Connect the MCP4921 DAC to the MSP432P401R via SPI.



3. Practice getting the DAC to generate specified DC voltages. For example, control the DAC to generate 1.0V, 2.0V, etc. to ensure it is operating properly.

Square Wave and Triangle Wave Generation

4. Write a program so that your system will generate a 2Vpp square wave with 1 VDC offset and period of 20 ms using timers.

5. Write a program so that your system will generate a 2Vpp triangle wave with 1 VDC offset and period of 20 ms using timers.

Note: Please supplied file *A5_DriveDAC_MSP432.c* as a starting point for this assignment.

Deliverables

1. A single pdf document with:
    a. Link to YouTube demo of triangle wave running on oscilloscope.
    b. Screen shot of square wave output on oscilloscope.
    c. Screen shot of triangle wave output on oscilloscope.
    d. Your code (nicely formatted of course).

A7 - Execution Timing

Instructions:

1. Set up an oscilloscope to catch a single pulse to collect accurate timing.
2. Set up an oscilloscope to catch a single pulse to collect accurate timing.
3. Use the provided code framework to time various arithmetic operations with the specified variable types and sizes. For consistent timing run the MSP432 at the default speed of 3 MHz.
4. P2 is toggled before and after `TestFunction` subroutine and P1 is toggled before and after the arithmetic function.
5. Adjust `TestFunction(num)` to test the various operations and variable types.
6. Fill in the table with the timing results obtained.

To time how long it takes to enter the subroutine, look at the difference between P2 going high and P1 going high. You do not need to calculate this time for each arithmetic function, use the simple testVar = num for filling this time in the table. For all other times, use the width of P1.

Timing Function	int8_t (8 bit)	int32_t (32 bit)	int64_t (64 bit)	float (32 bit)	double (64 bit)
Subroutine call					
testVar = num					
testVar = num + 1					
testVar = num + 7					
testVar = num * 2					
testVar = num * 3					
testVar = num / 3					
testVar = sin(num)					
testVar = sqrt(num)					
testVar = abs(num)					

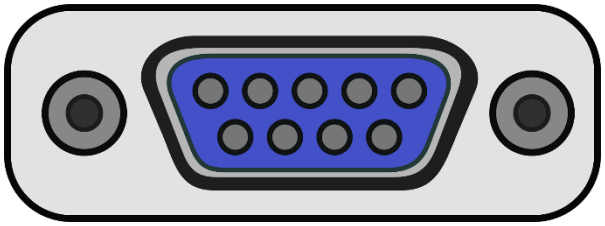
Deliverables:

1. A single pdf document of the completed timing function table.

ExecutionTIme.c

```
#include "msp.h"
#include <math.h>

var_type TestFunction(var_type num);

int main(void) {

    var_type mainVar;

    WDTCTL = WDTPW | WDTHOLD; // Stop watchdog timer

    P1->SEL1 &= ~BIT0; //set P1.0 as simple I/O
    P1->SEL0 &= ~BIT0;
    P1->DIR |= BIT0;   //set P1.0 as output

    P2->SEL1 &= ~(BIT2 | BIT1 | BIT0); //set P2.0-2.2 as simple I/O
    P2->SEL0 &= ~(BIT2 | BIT1 | BIT0);
    P2->DIR |= (BIT2 | BIT1 | BIT0); //set P2.0-2.2 as output pins

    P2->OUT |= (BIT2 | BIT1 | BIT0); // turn on RGB LED

    mainVar = TestFunction(15); // test function for timing

    P2->OUT &= ~(BIT2 | BIT1 | BIT0); // turn off RGB LED

    while(1)        // infinite loop to do nothing
        mainVar++; // increment mainVar to eliminate not used warning
}

var_type TestFunction(var_type num) {

    var_type testVar;

    P1->OUT |= BIT0; // set P1.0 LED on

    { insert_function_here (ie testVar = num;) }

    P1->OUT &= ~BIT0; // set P1.0 LED off
    return testVar;
}
```

A8 - UART Communication



Instructions:

1. Review the UART mode section of the Technical Reference Manual to familiarize yourself with the registers and configuration settings on the MSP432.

2. Review Technical Note 5 - ISR Communications for support information on schemes for communicating between main and an ISR.

MSP432 Transmitting Characters

3. Use the UART mode of USCI_A0 to get your MSP432 communicating with a terminal on your computer with RS-232. This port is connected through the debugger and uses the same usb cable that is used to program the board.

   Remember the UART only sends 1 byte at a time so you may want to start by getting the MSP432 to print a single character to the terminal. (Reference textbook Program 4-1)

MSP432 Receiving Characters

4. After the MSP432 is able to send a character to the terminal, you can have it receive a character from the terminal. (Reference textbook Program 4-2) *Note: Typing a character in the terminal transmits it rather than causes it to be displayed on the screen. If you want to see what you are typing, your program will need to echo the character it received.*

ISR Processing of Characters with DriveDAC in main

5. Implement your receive a character code as an interrupt service routine for the UART. When a character is sent to the UART, the interrupt will read the transmitted character and parse it to create a numerical value (int). *Note: This step requires the ISR take previous characters received into account as each character is received.* Some error checking needs to be done to ignore characters other than numbers (0 - 9) and return (enter key). You may need to refer to an ASCII table to complete this assignment. All entered characters should be echoed to the terminal so the user can see the characters as they are typed into the terminal.

   The ISR will read only 1 new character per call. The user will use a return character to signify the end of a numerical input. When a return is input, the ISR should set a flag (global variable) for the main program to update the DAC. The ISR cannot make any function calls, including library functions. (No atoi() or similar allowed. Sorry... Not Sorry)

   The program in main will continuously monitor the flag to determine if a new value has been entered by the user. When a new value is entered, main will send the entered value to the DAC. The program

needs to do error checking on the entered value before sending to the DAC. Only values from 0 to 4095 should be transmitted.

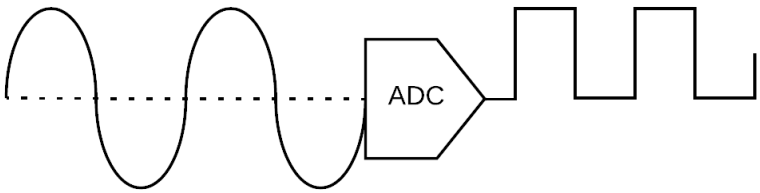
Bonus (10%):

6. Encapsulate your UART routines to eliminate all global variables. Instead of checking a flag in your main program, it will call a function to read the flag. The main program will still call DriveDAC, but will use a function to get the numerical value. Your UART functions will all go in a separate .c file and have static global variables for a flag and numerical value to limit their scope to a single source file. You will also need to create a matching header file (.h) which will list the function prototypes. This header file will need to be included in your main program.

Deliverables:

1. Single PDF document with:
    a. Link to YouTube demo of working device. The user should enter various values into the terminal and the output analog voltage should change. Either the scope or multimeter can be used to show the output voltage signal.
    b. Your code (nicely formatted and commented of course)

A9 – ADC14 Analog to Digital Conversion



Instructions:

Background Review

1. Review the ADC14 module section of the Technical Reference Manual to familiarize yourself with the registers and configuration settings on the MSP432
2. Review the Technical Note on using the hardware debugger to watch a variable without halting the execution of the program

Use the oscilloscope wavegen

3. Use a BNC to grabbers or alligator to connect the output signal (far left BNC connector) of the oscilloscope to the analog input you are using on the MSP432. *Note: If you try to use a probe to connect the output signal to your MSP432 you will make your electronics instructor cry.* **Make sure to connect the ground of the oscilloscope to the MSP432 ground.** Use the wavegen function of the oscilloscope to create a DC signal. You should hook up a probe to the output signal to measure and verify the voltages. **Be sure to never set the voltage above 3.3V!**

Use the debugger to verify ADC14 is working

4. Write a program that will take a single sample from the ADC14 using the 3.3 V reference. Use an ISR to detect when the conversion is complete and read in the value, save it in a static local variable, and set a global flag. Your main program run in an infinite loop checking for the global flag. When the flag is set, it should initiate a new sample for the ADC14 and reset the flag. *Note: Refer to sample program msp432p401x_adc14_01.c for help getting started.*
5. To verify your program is working, set a breakpoint inside the ADC14 ISR to update the expression view in the debugger. Set an expression for the variable you are saving the ADC value to. Run your program in the debugger and verify the variable in your ISR changes as you vary the voltage of the oscilloscope.

Calibrate your ADC14

6. Use the debugger to calibrate your ADC14 to get the voltage value that matches the oscilloscope from the 14 bit ADC value. This value should be accurate to +/- 0.01 V. *Note: this calibration should not happen inside the ISR. You can create a global variable for saving the ADC value.*

Print the voltage to the terminal

7. Print the calibrated voltage value to the UART. Remember that the UART can only transmit a single character at a time, so you will have to break down your calculated value into individual digits and transmit them sequentially. Printing to the UART cannot make use of any library functions. (No itoa() or similar allowed. Still not sorry.) Note: The UART transmission is slow. You should not sample the ADC significantly faster than you can print the output.

Bonus (10%): Encapsulate your ADC routines to eliminate all global variables.

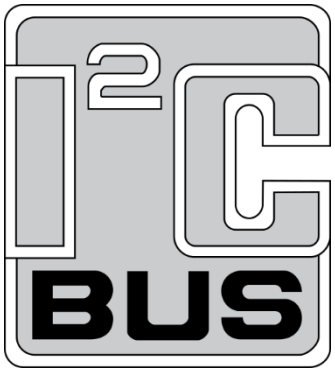
Deliverables:

1. A single pdf document with:

    a. Link to YouTube demo of working device. The video should show the oscilloscope display and terminal output as the voltage is adjusted from 0 to 3V.

    b. Your code (nicely formatted and commented of course)

A10 - I²C EEPROM



Instructions:

Background Review

1. Review the eUSCI I2C section of the Technical Reference Manual to familiarize yourself with the registers and configuration settings on the MSP432
2. Review the Microchip 24LC256 data sheet to understand the behavior of the EEPROM.

Organize Hardware / Software

3. Draw your own schematic of the MSP432 interfaced to the Microchip 24LC256 I2C EEPROM and wire up the circuit on a protoboard.
4. Review the C code provided and draw a detailed flowchart for describing the data flow and sequence of transmissions on the I2C bus for a write and read transmission.

I2C Bus Analysis

5. Modify the C code to write a byte of your choice to an address of your choice. Note the 5 ms delay between byte write and byte read is required by the EEPROM. Failure to keep this inter-byte delay will cause the EEPROM to not behave as desired.
6. Take two screenshots of the SCL (I2C clock) and SDA (I2C data) together:
    a. Screenshot 1 – byte write command
    b. Screenshot 2 – byte read command

    Crop and enlarge each screen shot in landscape format so that all parts of the byte write and read commands are visible on paper. Annotate each screenshot to identify:

    a. Start and stop condition
    b. Slave address
    c. Data write/read addresses
    d. Data byte written/read
    e. R/W bit condition and ACK bits

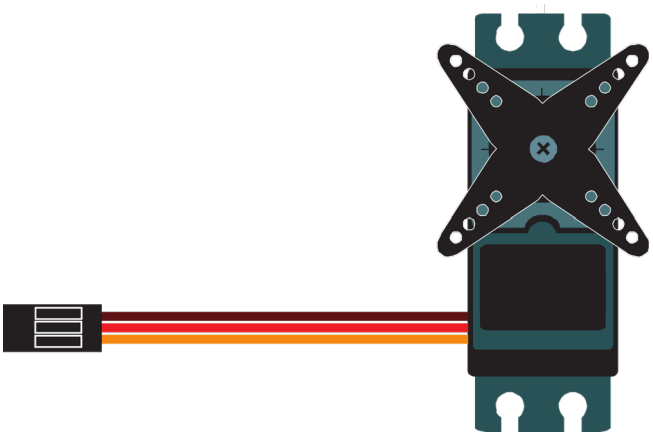
I^{2}C Hardware Tests

7. Remove the pull-up resistor on one of SCL or SDA and observe what happens.
8. Remove the wire to the SDA input on the slave (EEPROM), being sure to keep the pull-up resistor on the MSP432 intact. Rerun your program watching the MSP430's SDA with your oscilloscope. Observe what happens. Is the first ACK bit affected? What about the rest of the bits?

Deliverables:

1. A single pdf document with:
    a. Schematic of the MSP432 interfaced to the Microchip 24LC256 I2C EEPROM
    b. Annotated screenshots of SCL and SDA
    c. Answers to the questions
    d. Flow chart of the sample code

A11 - Pulse Width Modulation and Servos



Instructions

1. Review the Timer_A section of the Technical Reference Manual to familiarize yourself with the registers and configuration settings on the MSP432.
2. Review the Parallax standard servo documentation on PolyLearn.
3. Checkout a servo from the senior project lab.
4. Build a servo controller that meets the following specifications:

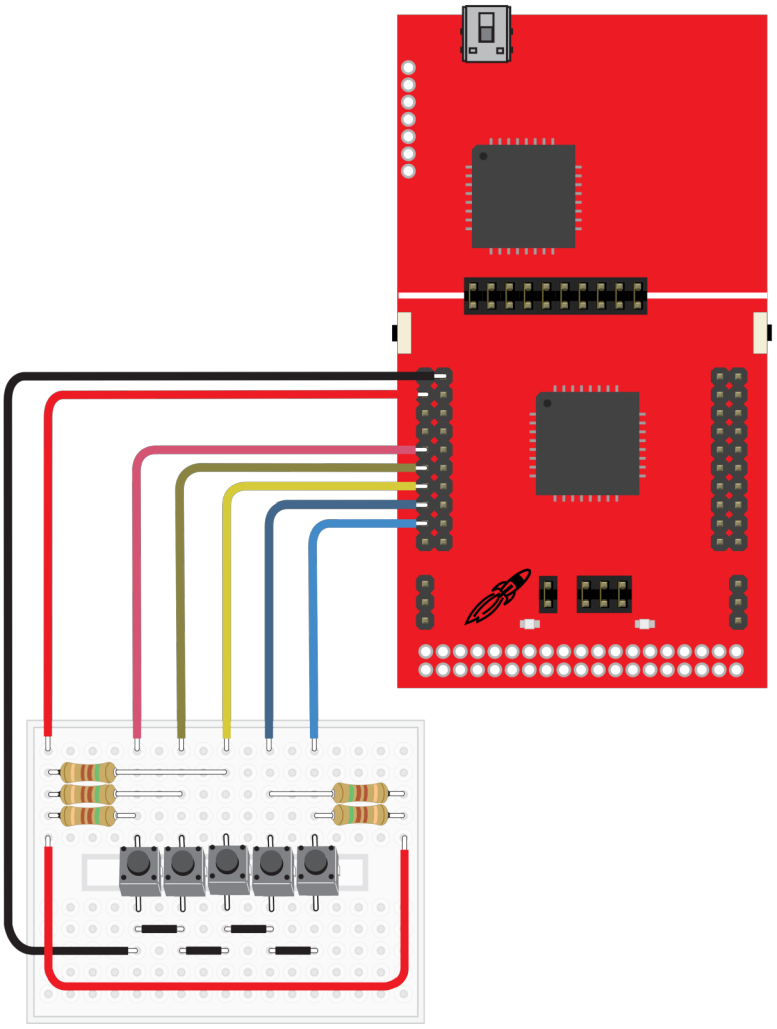
Device Specifications

1. The device shall control a single standard servo
   - 1.1. The servo shall operate from 0^{o} to 180^{o}
   - 1.2. The servo position shall move in increments of 10^{o}
2. The device shall use a keypad for input
   - 2.1. The device shall accept numerical inputs
     - 2.1.1. The numerical inputs shall be 2 digits
     - 2.1.2. The numerical input shall range from 00 to 18
     - 2.1.3. The device shall position the servo at the location specified by the numerical input as increments of 10^{o} from 0^{o} to 180^{o}
   - 2.2. The device shall accept # and $*$ input keys
     - 2.2.1. The device shall change the servo position 10^{o} counter clockwise when # is entered
     - 2.2.2. The device shall change the servo position 10^{o} clockwise when $*$ is entered

Deliverables

1. A single pdf document with:
   a. Link to YouTube demo of working device. The video should show the servo rotating to various angles entered by the user.
   b. Your code (nicely formatted and commented of course)

Projects



P1 - Hello World Electronic Lock

Introduction

The purpose of this project is to integrate the LCD module and the 12-key keypad to create an electronic lock. The lock combination will be stored in a .h file. The LCD will print Hello World when the correct combination is entered.

Objectives

1. Integrate the LCD and keypad to run together on the MSP432 LaunchPad.
2. Develop an MSP432 application which allows the user to open the lock by entering the correct combination.

Design Steps

1. Draw a single schematic diagram of the keypad and LCD integrated with the LaunchPad.
2. Confirm that the keypad and LCD run properly with the LaunchPad
3. Implement combination lock functionality.

Integrate the LCD and Keypad

The LCD Module requires 7 GPIO pins when running in nibble mode, the keypad requires 7 additional GPIO pins. The MSP432 LaunchPad has a total of 35 GPIO pins available on headers J1, J2, J3, and J4, as shown below.

Develop the Combination Lock

Utilize work done in the preceding steps to implement the electronic combination lock.

Tips and Notes

Successful completion of this project requires you to assimilate a variety of information from different sources and implement a system solution. Although this process may seem daunting at the start, a methodical step-by-step approach can be helpful. Here are suggested steps for your project implementation.

1. Give all data sheets and reference manuals a read to begin your understanding of both devices/components and the reference material available to you. There is no substitute for reading the data sheets to get started. Aside from understanding key aspects of the device required for successful interfacing and operation, reading the data sheets will potentially illuminate extended or extra features of the device which may be advantageous to the designer.

2. Utilize accurate schematic diagrams to guide your work. The final schematic for this project should include the MSP432, the LCD display, the keypad, any required buttons and resistors, and all required power supply connections. Please see the example schematic at the end of this procedure for the Commodore 64. Common schematic practices include:

    a. Labeling each chip, component, etc. with a reference designator, e.g. U1, R1, C21, etc.
    b. Label each chip/component with the value/part number for the component, e.g. 10K, .01uf, MSP430G2553, etc.
    c. Numbering all pins above the trace outside of the chip.
    d. Labeling the chip pin name, e.g. P1.0, inside the chip next to where the wire attaches to the chip.
    e. Labeling all wires with a signal name, e.g. A21, D7, chip_select, etc.

    The process of drawing a first-cut schematic does not have to be long or involved, rather can be a quick effort using pencil and paper in which key information such as pin names and pin numbers are given to aid in wiring and troubleshooting. A formal schematic can be constructed at a later time once the design is solid.

3. Sketch out what the software will do. Aids such as flow charts, pseudo-code, etc. may be used to do this. This is also a good time to start the project using CCS or preferred development tool and start creating **#define** constants that will be useful.

4. Prototype key aspects of software operation to get the basics working. Get the keypad working. Get the LCD working. Tie them together in a simple fashion. Then build your application. Don't forget that the main point of prototyping is to test ideas and get the basics working. Once key functionality such as printing a letter to the screen of the LCD has been achieved, code and documentation cleanup is easy.

5. Don't forget to make plenty of backups along the way to preserve the good work that has been accomplished. It is not a good feeling to get a major part of the system working, continue working on

the code, break the code, and then have to spend time fixing what was broken. Making frequent backups will allow the developer to roll back to a known development state when something goes wrong. Although there are various comprehensive version control products available for this purpose, a simple approach such as maintaining version directories of a development project on Dropbox.com or similar are satisfactory for projects such as this.

6. Clean up source code and documentation. Document source code, break system functionality out into neatly abstracted functions such as InitLCD(), SetCursorLocation(), WriteCharacter(), WriteLine(), etc. so that the software can be easily reused at a later time. This is also a good time to move the source code written into a dedicated set of files such as LCD.c and LCD.h so that functionality can be easily used in any future project.

System Requirements

Upon starting, the LCD will display:

LOCKED
ENTER KEY

Pressing keys on the keypad will display on the bottom row after KEY
After 4 digits, the display will be cleared and either display HELLO WORLD if the key was correct or display the same LOCKED ENTER KEY screen and wait for a new key sequence.

If the $*$ key is pressed, the LCD will be cleared, the LOCKED ENTER KEY screen will be displayed and it will wait for a new key sequence to be entered.

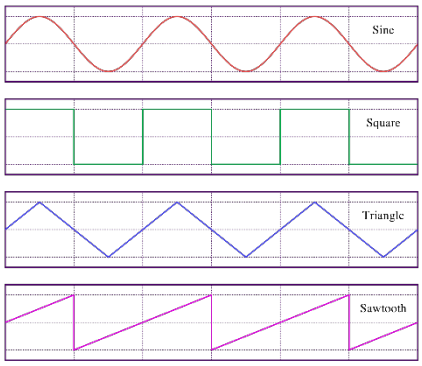
Project Demonstration and Lab Report Submission

Your lab report should follow the lab submission guidelines (see PolyLearn)

Project Grading

Simply completing the project assignment is only sufficient to get you 90% of a complete grade. In order to get 100%, you will need to complete an extension of your choosing

P2 - Function Generator



Learning Objectives

1. Understand how to use timers and interrupt to generate timing events
2. Understand how to use a serial-peripheral interface (SPI)
3. Understand how to use digital-to-analog (DAC) converters to generate analog signals from a microcontroller

Introduction and Overview

You are to design a function generator using a microcontroller. The microcontroller should be connected with an external DAC to generate the analog waveforms required by the function generator. This DAC should have an SPI interface. The waveforms that the function generator must generate include a saw tooth waveform, a square wave with a variable duty cycle, and a sinusoidal waveform. The frequency of the waveforms will also be variable. The keypad will be used to select the output waveform type, set the frequency of the waveform and set the duty cycle of the square wave.

System Requirements

1. SPI DAC drive shall be optimized and documented in your report to show:
    a. Total time required to complete the transmission of 16 bits and raise /CS at the end. This should include a screen shot of clock, data, and /CS for the transmission of 2 bytes via SPI. Please demonstrate that you have optimized DriveDAC from a time perspective; no time is being wasted with unnecessary clock cycles or delays and hence it would not be practical to transmit two bytes faster than they are configured.
    b. Calculations which demonstrate the potential maximum update rate for the DAC and hence an upper bound on the resolution of your function generator when generating a 500 Hz sine wave.
2. The function generator shall use a microcontroller and an external DAC.
    a. The external DAC shall have an SPI interface and a minimum of 8 bits of precision
3. The function generator shall be capable of producing
    a. A square wave with variable duty cycle
    b. A sinusoidal waveform
    c. A sawtooth waveform

4. All waveforms shall be DC-biased around Vdd/2 with Vdd set to 3.3 V.
5. All waveforms shall have adjustable frequencies:
    a. 100 Hz, 200 Hz, 300 Hz, 400 Hz, 500 Hz
    b. The frequency of the waveforms should be within 5% of these frequencies. Less than 2.5% error for full-credit.
    c. The function generator shall have an output resolution of at least 70% of the maximum theoretical from (1.b) above.
    d. The output rate (points / sec) shall not change with waveform frequency (i.e., the 100 Hz waveform will output 5x the number of points in one period as the 500 Hz waveform).
6. Upon power-up, the function generator shall display a 100 Hz square wave with 50% duty cycle.
7. The keypad buttons 1-5 shall set the waveform frequency in 100 Hz increments (1 for 100 Hz, 2 for 200 Hz, etc).
8. The keypad buttons 7, 8, and 9 shall set the output waveform to Square, Sine, and Sawtooth.
9. The keypad buttons ✱, 0, and # shall change the duty cycle of the square wave.
    a. The ✱ shall decrease the duty cycle by 10% down to a minimum of 10%.
    b. The # shall increase the duty cycle by 10% up to a maximum of 90%
    c. The 0 key shall reset the duty cycle to 50%
    d. The keys ✱, 0, and # shall not affect the sin or sawtooth waveforms.
10. You must use the on-chip timer to generate interrupts that generate timing events for your function generator. You may not use software delays (ex: delay_ms) to generate timing events for outputting to the D/A. It would be more difficult to use software delays, anyhow. You may use software delays to help debounce buttons (if necessary).

Project Demonstration and Lab Report Submission

Your lab report should follow the lab submission guidelines (see PolyLearn)

Grading

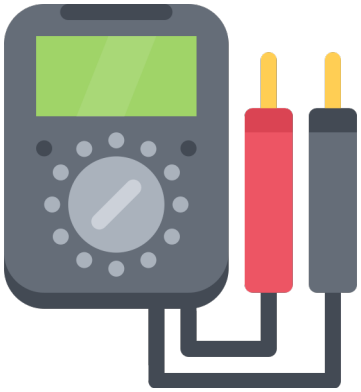
Simply completing the project assignment is only sufficient to get you 90% of a complete grade. In order to get 100%, you will need to complete an extension of your choosing. Some *suggested* extensions include:

a. Use the LCD screen to display the frequency/duty cycle/waveform type
b. Add a DC offset feature
c. Add an option that uses the ADC to sample an input waveform and then output it using the DAC.

Tips and Notes

1. Start with developing the square wave with the timer and interrupts.
2. Develop a plan of how many points you want to write to the DAC per period of each of the waveforms.
3. It is tempting to use the C programming language's sin() function, however this function uses floating point numbers and should not be used "at run time".
4. One common microcontoller "Trick" is to use pre-computed values stored into a look-up table so that complex computations do not have to be done by the microcontroller. You may or may not need to use such tricks as the MSP432 has a floating point unit. Please ensure that any use of floating point math, e.g. calculation of a sine function, does not degrade system performance in a significant way, e.g. making it too slow.
5. It may take you some effort to fine tune your function generator to make it meet specifications.
6. Page 7 of the 4921 DAC datasheet provides useful information for optimizing DriveDAC(). Note the parameters *"/CS Fall to Frist Rising CLK Edge"* and *"SCK Rise to /CS Rise Hold Time"* in the datasheet snip below. These indicate the precise amount of time that /CS must be low before and after the 16 SPI clocks.

P3 - Digital Multimeter



Instructions

Project 3 requires the design and implementation of a digital multimeter (DMM) that can measure voltage and frequency. Please see requirements below for this DMM.

Function Requirements

1. The DMM shall measure voltage.
    1.1. Voltage measurements shall be limited to 0 to 3.3 volts.
    1.2. Voltage measurements shall be limited to 0 to 1000 Hz.
    1.3. Voltage measurements shall be accurate to +/- 1 mv for AC and DC.
    1.4. The DMM shall have a DC setting.
        1.4.1. DC measurements shall average over a 1 ms time period.
        1.4.2. DC measurements of a sinusoidal waveform should be equivalent to the DC offset of the sinusoid.
    1.5. The DMM shall have an AC setting.
        1.5.1. AC measurements shall be true-RMS.
        1.5.2. AC measurements shall display the various components.
            1.5.2.1. AC voltage measurements shall give the true-RMS (includes DC offset).
            1.5.2.2. AC voltage measurements shall give the peak-to-peak value.
        1.5.3. AC measurements shall work for various waveforms
            1.5.3.1. Sine waves shall be measurable
            1.5.3.2. Triangular waves shall be measurable
            1.5.3.3. Square waves shall be measurable
            1.5.3.4. Other periodic waveforms shall be measurable
        1.5.4. AC measurements shall work for waveforms of various amplitudes and offsets
            1.5.4.1. The maximum voltage that shall be measured is 3V

1.5.4.2. The minimum voltage that shall be measured is 0V

1.5.4.3. The minimum peak-to-peak voltage that shall be measured is 0.5V

1.5.4.4. Offset values of up to 2.75V shall be measurable

2. The DMM shall measure frequency.
   2.1. Frequency measurements shall be limited from 1 to 1000 HZ.
   2.2. Frequency measurements shall be accurate to within 1 Hz.
   2.3. Frequency measurements shall work for various waveforms.
      2.3.1. Sine waves shall be measurable.
      2.3.2. Triangular waves shall be measurable.
      2.3.3. Square waves shall be measurable.
      2.3.4. Other periodic waveforms shall be measurable.

3. The DMM shall have a terminal-based interface.
   3.1. The terminal shall operate at a frequency greater than 9600 baud.
   3.2. The terminal shall utilize the VT100 protocol.
      3.2.1. The terminal shall display all fields in non-changing locations
   3.3. The terminal shall display AC voltages as described above.
   3.4. The terminal shall display DC voltages as described above.
   3.5. The terminal shall display frequency as described above.
   3.6. The terminal shall organize the presentation of information.
      3.6.1. AC, DC, and frequency shall be simple to read.
      3.6.2. The display may use horizontal and vertical lines (borders) to organize the presentation of information.
   3.7. The terminal shall use bar-graphs for voltages being measured.
      3.7.1. The terminal shall have a bar-graph for “true-RMS”.
      3.7.2. The terminal shall have a bar-graph for DC voltages.
      3.7.3. The bar graphs shall be shall have delineators, e.g. a scale, indicating the equivalent voltage being measured.
      3.7.4. The bar graphs shall be a single line of pixels, characters, etc.
      3.7.5. The bar graphs shall have length that is proportional to the voltage being measured.

3.7.6. The bar graphs shall respond in real-time to changes in AC or DC voltage

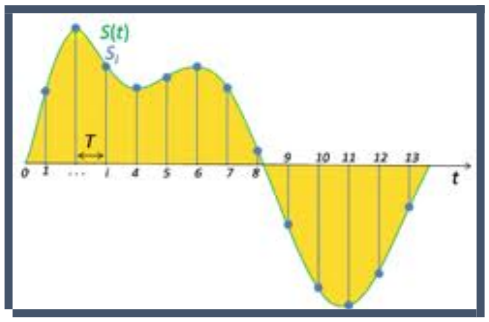
Development Constraints

DMM source code shall utilize #defines from msp432p401r.h

Video Demonstration

1. DC Voltage
    a. Two different DC values between 0 and 3.3 V
2. AC voltage
    a. 3 Vpp sine wave with 1.5 V DC offset at 500 Hz
    b. 1 Vpp sine wave with 0.5 V DC offset at 1 kHz
    c. 1 Vpp sine wave with 2.5 V DC offset at 1 kHz
    d. 3 Vpp triangle wave with 1.5 V DC offset at 50 Hz
    e. 1 Vpp square wave with 1.5 V DC offset at 10 Hz
3. Frequency Specific
    a. 3 Vpp sine wave with 1.5 V DC offset at highest possible frequency (1 kHz to meet spec)
    b. 3 Vpp square wave with 1.5 V DC offset at lowest possible frequency (1 Hz to meet spec)

P4 – ADC Design and Development



Introduction

Lab Improvements

1. Add a section on how resistor network works to create sum of taps. Possibly include a video of microcap simulation. Show based on EE112 familiar superposition.
2. ADD REQUIREMENTS FOR SYSTEM:
   a. Vin >= 1Vpp with no visible distortion, document results in final specs.
   b. Sampling rate greater than 1K samples per second, document results in final specs.
   c. DriveDAC output should be at a constant repetition rate. This implies that system will need to be timer-based for sampling and that DriveDAC can't simply be called when the value is found.

This project pulls together various EE and CPE skills you have developed to date. Resistive networks (EE 112), storing charge on a capacitor (EE 211 and EE 307), use of comparators/op-amps (EE 211), transmission gates to pass or block a voltage (EE 307/347), and use of a microcontroller to automate a task (CPE 329).

Figure 1 shows the basic model of the sample and hold ADC which you will construct in this project. This circuit is described in Maxim Semiconductor Application Note #5074, *"Implementing an ADC with a Microcontroller, an Op Amp, and Resistors".*

Resistor Network to Generate DC Voltages

The resistor network is utilized to allow digital controls to supply a variety of voltages to the **+** comparator input. These voltages are compared against the sampled input voltage, which is applied to the capacitor across the **–** input to the comparator. The comparator output will be 0V (the ground rail) when the non-inverting input is less than the inverting input. The comparator output will be at the positive rail, e.g. 3.5V, when the non-inverting input is less than the voltage being sampled at Ain and is across capacitor C3.

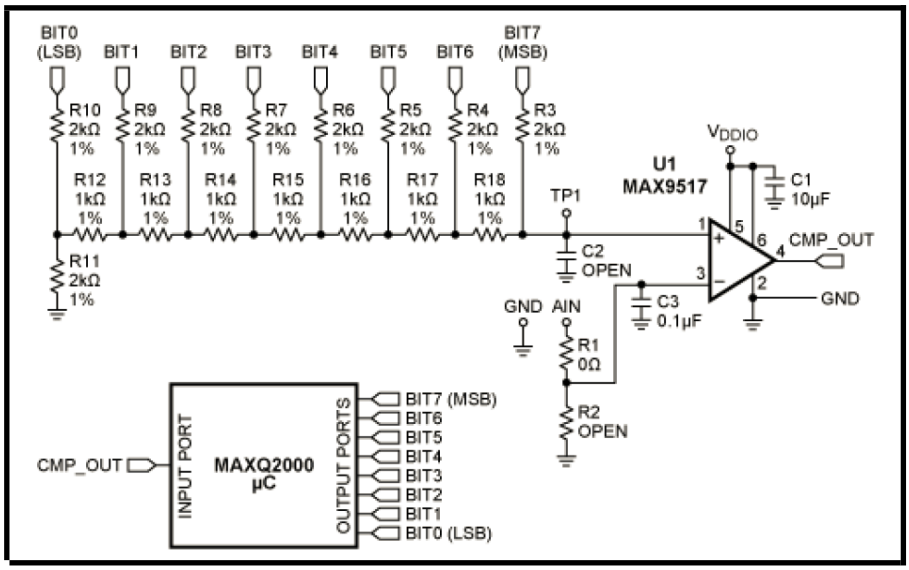


Figure 1. Sample and Hold Analog to Digital Converter Prototype

Transmission Gate to Sample Input Voltage

An electronic switch, which allows the voltage at Ain to be applied to capacitor C3 and be across the inverting input, is not shown in this circuit. This switch, which will likely transmission gate such as the CD4016 you utilized in the EE347 Dynamic Storage experiment. This transmission gate allows the analog input voltage to be sampled periodically and stored across C3 so that the microcontroller can test it against a variety of voltages generated by digital values applied to the resistive network Bit0 – Bit7 inputs. The digital number which causes the non-inverting input to be closest to the value sampled and held on the inverting input will deemed to be the equivalent value of the analog input.

Microcontroller Tasks

The microcontroller's performs three key functions:

1. Generate a sample clock which will activate the transmission gate once per sample to allow the analog input voltage on the input to be stored across the capacitor on the inverting input of the comparator.
2. Apply a range of digital values across Bit7 downto Bit0 on the resistor network.
3. Test the output of the comparator to determine which digital value applied to the resistive network inputs is closest to the analog voltage which is at the inverting input of the comparator.

The process of generating voltages with the resistive network may be accomplished in a variety of ways. A simple ramp function may be used where the test values applied to the non-inverting input may be used, although this method suffers from speed problems due to having to test 256 values to determine which one closest to the analog value. Other search schemes similar to a binary search offer the ability to estimate the value with many less iterations. The example information in the rest of this lab procedure is based on the concept of testing for the actual analog value by applying the simple ramp to the non-inverting input and looking for the value which is closest to the actual analog input.

Instructions

This project requires you to successfully implement a variety of circuits, each of which is not terribly difficult in and of itself. These circuits, when taken as a group, can be daunting to make work as a group. It is recommended that you perform the following steps to aid in your successful execution of development of this ADC.

Create a Complete Schematic and Build the Circuit

1. Make a simple but complete, hand-drawn schematic of the entire system. This system should be similar to the circuit shown below, but show all components, including the transmission gate, the comparator you will be using, the capacitor used to hold the sampled analog value, the resistive network and the MSP432. Your circuit should take no more than a sheet of paper and should show all connections, pin numbers, etc. Also, be sure to label each chip as U1, U2, etc.

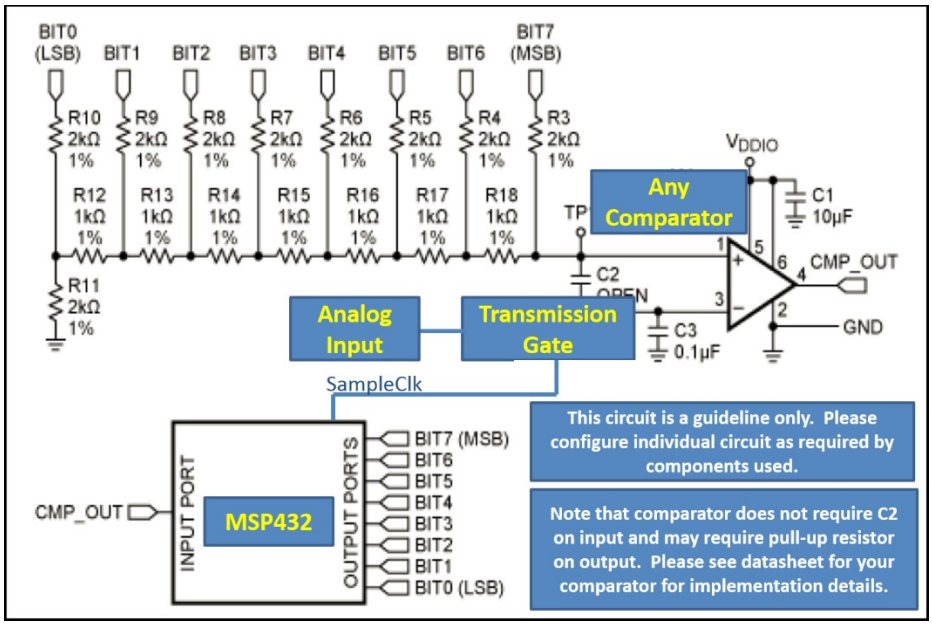


Figure 2. Sample and Hold Analog to Digital Converter circuit modified to show parts required in CPE 329.

Include the SPI-driven DAC in your system. This will allow you to directly output any input value you sample to the DAC so you can observe it on the oscilloscope to see what the ADC is sampling. You will need to plan pin utilization on your MSP432 carefully to ensure that you can make all peripherals together.

2. Please Have your schematic approved by the instructor with initials in the upper right corner. It is expected that you keep this schematic with you during all development work on this project.

3. Obtain necessary parts, get all datasheets on hand, and build your circuit.

Write Code to Generate the Ramp Function

4. Write a C function which drives Bit7 downto Bit0 to count from 0 to 255 (0xFF) and also creates a positive pulse called SampleClk to drive the transmission gate control input. Test this code to ensure that you see a pulse turning the transmission gate on and then a ramp voltage on the non-inverting input of the comparator.

   These pulses should look something like the following screen shot, where Channel 1 is non-inverting input of comparator (output of resistor network). Note that it transitions from 0V to about 3.4V in a fairly linear fashion as the 8-resistor control bits count from 0x00 to 0xFF. Channel 2 is sample/hold pulse applied to CD4016 transmission gate, which is used to allow the Vin signal to be sampled through to the inverting input of the comparator.

   You may want to run this code in a loop of some sort or call it periodically with an interrupt. Be sure to note how long your SampleCLK combined with your ramp function takes to complete so that you know the minimum amount of time to complete a single sample.

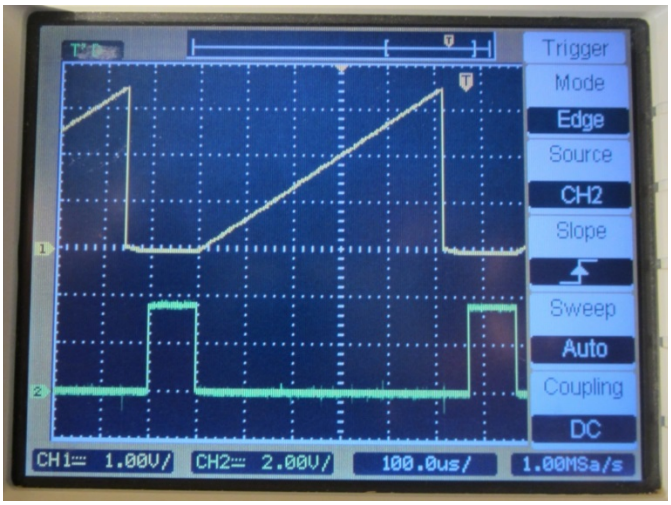


Figure 4 – Analog ramp voltage on output of resistive network (CH1) and SampleCLK (CH2).

Test Operation of the Transmission Gate and Sample Clock

5. With your SampleCLK and resistor ramp running, test the operation of the transmission gate. One way to accomplish this would be to take a 0 to 3V ranging sine wave and apply it to the input of the transmission gate. This waveform may need to be very slow so that the ADC sampling rate faster than the period of your input waveform. Knowing how fast you can run the sampling system will let you know what frequency of input waveform you can test with.

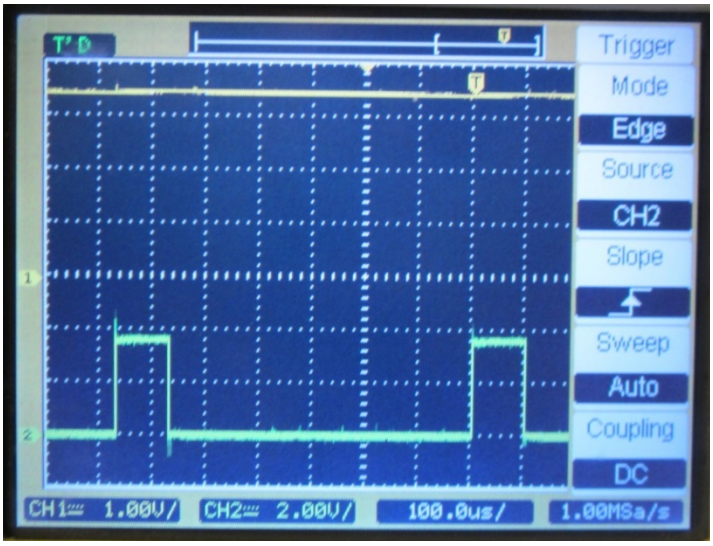


Figure 5 – SampleClk and output of transmission gate with sampling capacitor in place. +3.4 VDC system power was applied to the input of the 4016 transmission gate to test the hold aspect of the system. Channel 1 is the output of the transmission gate, with 3.4 volts holding steady for the duration of the conversion process, which is between the two pulses.

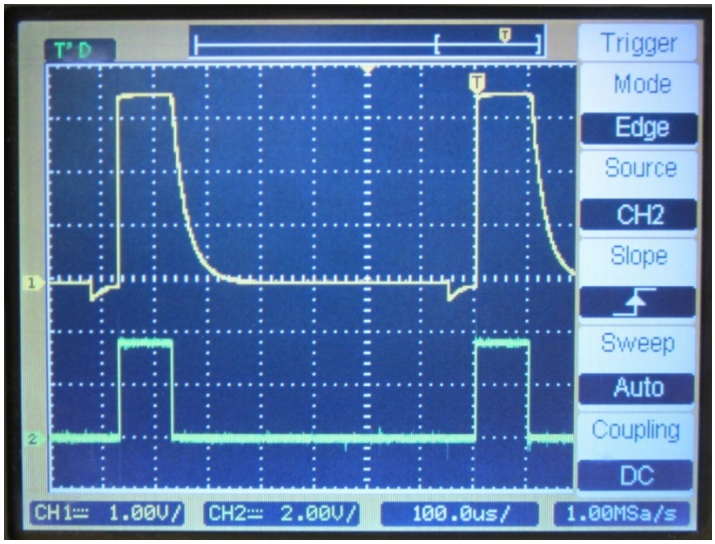


Figure 6 - Channel 1 is the output of the 4016 transmission gate with the 0.1 uF capacitor removed. Note that the voltage does not hold for the conversion period.

Use your oscilloscope to watch both the input and output of the transmission gate. You should see the raw waveform on the transmission gate input a sampled version on the output. Once this is working and you verify that you have both the ramp voltage and the sampled input waveform, you are ready for the next step.

6. With your sampling system running, apply a constant DC voltage to the analog input. Observe the output of the comparator to ensure that it flips from 0V to 3.5V (the rail) whenever the sampling ramp voltage

exceeds the DC input voltage. You should see something similar to the following figure, where the sampled input voltage holds steady between SampleClk pulses.

Monitor Sampling with the DAC

7. Insert a call to DriveDAC into your code so that it is called every time you complete an ADC sample. You will now be able to apply a waveform to be sampled on the analog input and see the sampled and then DAC generated waveform regenerated on the output.

P5 - Final Project

Final Design Project Overview and Timeline

Your CPE 329 final project should be culmination of embedded systems topics learned and experience gained this quarter. This project should utilize a sensor or other peripheral as a central theme to the system you develop. This sensor must be interfaced to and calibrated, which will be an intermediate deliverable. The final system developed needs to be a stand-alone system which provides utility; it must perform a useful job for the user, provide entertainment, etc. Good examples of a stand-alone embedded systems are things like digital multimeters, hand-held medical devices such as thermometers, cordless phone handsets, etc. Your final project system ***may*** have other interesting components/subsystems besides the sensor/key peripheral. These other subsystems may include things like wired and wireless communications links, displays, other sensors, other systems, etc.

Sensor/Peripheral Interfacing and Calibration/Validation

The objective of Part I of Project 45 is to select a sensor/peripheral, interface to it, calibrate it in the real world, and then build a simple system which monitors the sensor and displays the output of the sensor on some sort of a display, e.g. the LCD panel. Your choice of sensor/peripheral will somewhat impact what you learn in the course of this project. Generally, most sensors produce an analog output, must be sampled and converted using an A/D. The microcontroller then can actuate some output based on this sensed input. Other sensors which utilize serial or other interfaces are also likely candidates for this project, as long as they can be calibrated and documented so that their real-world performance is repeatable during system development part of this project (Part II). Please see *Technical Note 8 (TN8)* for guidance on calibrating your sensor/peripheral.

Table 1 at the end of this document provides links to a variety of sensors and peripherals which may be suitable for this project. Some are simply sensors and others are packaged devices. Please note that this list is only a suggestion and you are free to find other sensors and peripherals. Each group will purchase all sensors and components required for this project. Don't forget to incorporate delivery time into your project schedule.

Serial Console -or- Battery Powered Operation

The final system should utilize either a serial console or have battery powered operation, as described below. One or the other is required, but not both.

Serial Console in Final System

If a serial console is selected for use within the final system, it must be capable of monitoring system parameters in real time and allow system configuration without recompilation in the development environment. This functionality should be clearly documented in a section in the report document titled "Serial Console", which shows commands given, responses, etc.

A VT100 Terminal emulator provides a nice way to build a gui of sorts for a terminal into a MSP432. Various VT100 libraries and source code may be found on the Internet to support VT100 operation on the MSP432 side of things. A VT100 capable terminal program will be required to communicate with your system.

Battery-Powered Operation

If battery-powered operation is selected, the system must be demonstrated to operate with battery power. The design and operation of the battery power system, including calculations and measurements to predict system autonomy (battery life), should be documented in a section titled "Battery-Powered Operation" in the final report.

Device Enclosure

The final device should be housed in a self-contained enclosure. The enclosure can be a standard size box (sold by digikey and others), custom 3D printed, or fabricated in any means that results in a professional looking device. Cut outs for displays, buttons, and connections should be made cleanly. Any necessary wiring leaving the enclosure should be bundled neatly. Cardboard boxes, reused consumer packaging, etc. are not suitable enclosures.

Project Demonstration and Presentation

Each lab group will be required to give a demonstration of your project and a presentation on the last day of class: Presentations will be 5 minutes in length need to include:

1. Technical description of your project
2. Live demonstration
3. Lessons learned and next steps for system developed

Presentations will be given via the projector at the front of the room and should utilize a formal presentation tool such as Microsoft Powerpoint.

Grading

Final projects will be graded based on the following metrics.

Meeting Specification
Does the final result meet the specification/design outlined or as approved by instructor? Does it work properly?

Fit and finish
How does the end result look from the perspective of enclosure, user interface, etc.?

Final Report and Presentation
Quality of final report and presentation.

Ranking by class members and degree of difficulty
Class members will be vote on projects, scoring them based on end result, favorite project, etc.
Degree of difficulty assigned by instructor.

Deliverables and Report Format

Please submit two files:

1. A pdf of your project report named as:
   cpe329_s2017_proj_ report_lastname1_lastname2.pdf

2. A zipped file of your entire Code Composer project titled:
   cpe329_s2017_proj_dev_lastname1_lastname2.zip

The final report should contain the sections shown below. Project reports should adhere to all standards previously presented in the quarter. This report is a reflection upon you and Cal Poly - it should be portfolio worthy. Please note that project reports will be compared to each other as a part of report grading; the quality of your work must live up to the quality of work of your peers. Please include these sections in your report:

A. Cover page

   a. Project name/title
   b. Names, class number, section #, quarter, experiment #, title, date, and instructor's name.
   c. A tasteful graphic if desired.

B. Introduction

The introduction should include a summary of the project, how it works, and key items in the report. This is a good place for a link to your own YouTube video of your system operating.

C. Requirements

Requirements, as completed previously.

D. System Specification

Spec. sheet, as previously described.

E. System Architecture and Design

Provide a complete technical description of your system, how it works, etc. This section should utilize architecture diagrams, schematics, flowcharts, etc. to describe the operation of your system. This section is effectively a theory of operation and will let the reader know how your system works.

F. Bill of Materials

BOM, as previously described.

G. Discussion, Conclusions, and Recommendations

Describe the high and low points of your system integration and development process, give conclusions and recommendations for continuing the work.

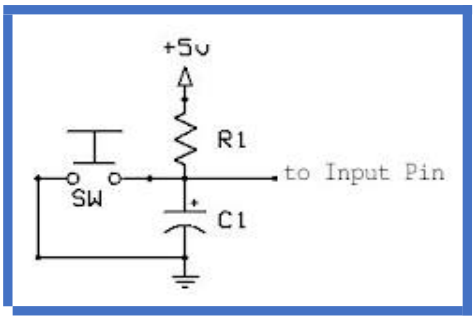
Appendices

Appendix A - User's Manual
Appendix B - Formatted code
Appendix C - any other supplemental material deemed necessary.

Potential Sensors and Peripherals
www.sparkfun.com *
www.digikey.com *
http://store.digilentinc.com/pmod-modules/
www.adafruit.com (beware of parts with no datasheets!)
http://www.ti.com/lsds/ti/tools-software/launchpads/boosterpacks/select/select.page
*SparkFun and Digikey provide reliable 2-day USPS priority mail shipping for a reasonable price.

Technical Notes

TN1 - MSP432 GPIO Interrupts



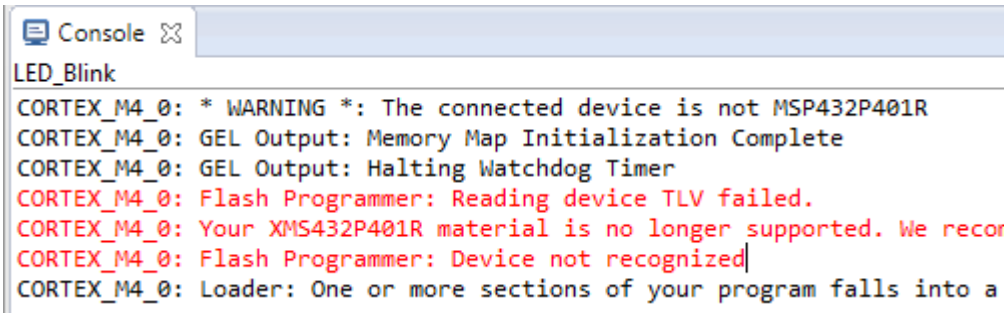
Circuit for generating interrupts via button connected to P1.3. LED on P1.0 toggles based on button press. Note that code has options for running in low power mode 4 or with main loop.

```
#include <msp432.h>
int main(void) {
    WDTCTL = WDTPW | WDTHOLD;    // Stop watchdog timer

    P1->DIR |= BIT4 + BIT0;       // P1.4 and P1.0 output bits for toggling
    P1->OUT |= BIT4 + BIT0;       // Start with LEDs illuminated

    P1->IE  |= BIT3 + BIT2;       // Enable interrupts for P1.3 and P1.2
    P1->REN |= BIT3 + BIT2;       // Add an internal pullup resistor to P1.3 and P1.2
    P1->IES |= BIT3 + BIT2;       // Select high to low edge Int on P1.3 and P1.2
    P1->IFG &= ~(BIT3 + BIT2);    // Clear the interrupt flags to ensure system
                                  // starts with no interrupts

    // Code to run with interrups, sleeping deep between interrupts
    NVIC->ISER[1] = 1 << ((PORT1_IRQn) & 31);  // Enable Port 1 interrupt on the NVIC
    SCB->SCR |= SCB_SCR_SLEEPONEXIT_Msk;       // Do not wake up on exit from Int
    SCB->SCR |= (SCB_SCR_SLEEPDEEP_Msk);       // Setting the sleep deep bi
    __enable_irq();                            // Enable global interrupt


    // Main loop version with no interrupts
    while(1);                    // main loop, looking for an interrupt,
                                 // just wasting CPU cycles otherwise
}
/* Port1 ISR */
void PORT1_IRQHandler(void)

   // Check and clear each interrupt individually.  Note that both are always
   // checked with no 'else' to ensure one is not missed. Toggle bit as
   // appropriate in interrupt.
   if (P1->IFG & BIT2){
      P1->OUT ^= BIT0;              // Toggle P1.0
      P1->IFG &= ~BIT2;             // Clear the Bit 2 interrupt flag,
                                    // leave all other bits untouched
   }
   if (P1->IFG & BIT3){
      P1->OUT ^= BIT4;              // Toggle P1.4
      P1->IFG &= ~BIT3;             // Clear the Bit 3 interrupt flag,
                                    // leave all other bits untouched
   }
}
```

TN2 - Factory Reset MSP432

Error Requiring Factory Reset

If you get a "**`CORTEX_M4_0:* WARNING *:The connected device is not MSP432P401R`**" or you get an error similar to that below and are unable to connect to your MSP 432 to program it, you must factory reset the device

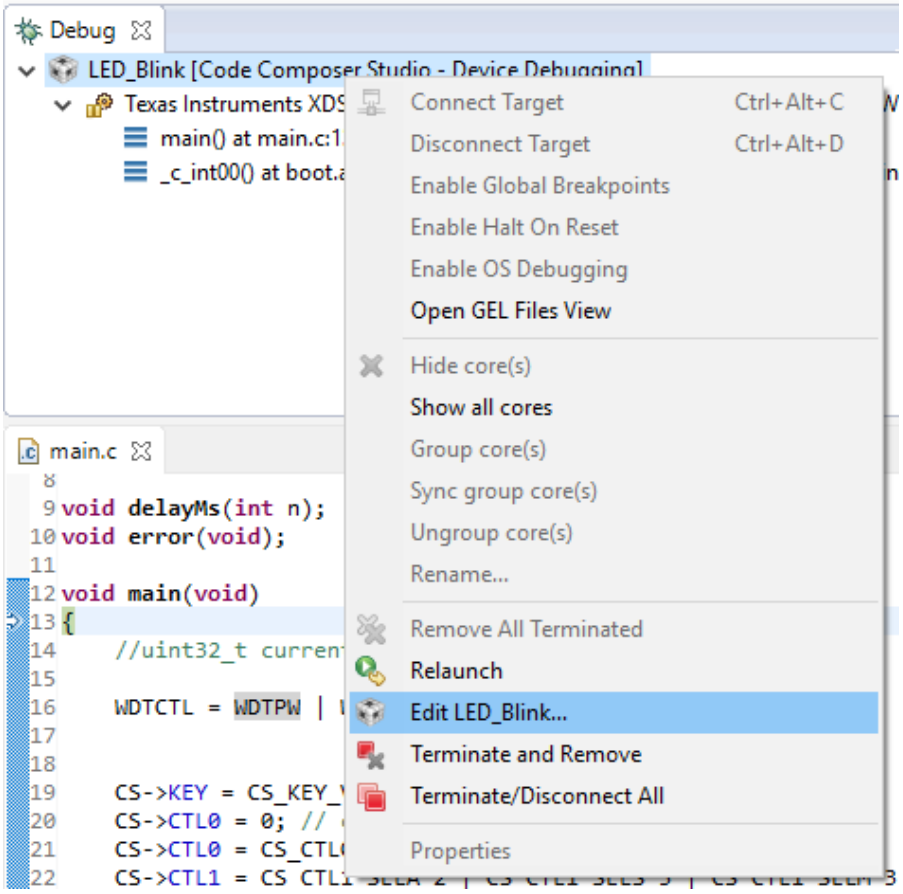


If Code Composer Studio pops up an error dialog specifying that it failed to load your *project_name.o* file and closes the debug interface, you will need to change your target settings before you can connect to the Debug Probe. If you do not get this error, you can skip to Step 4.

Reset Procedure

Step 1. Change CCS to the Debug Perspective by selecting with the button in the top right corner.

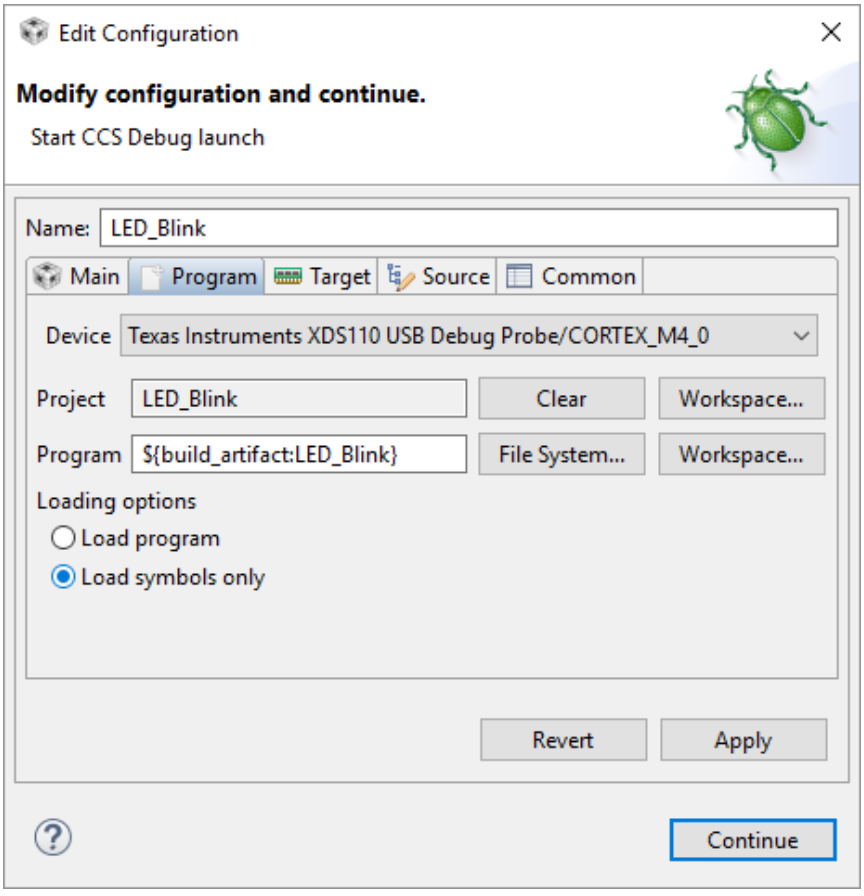
Step 2. From the debug perspective, right click on your **CCS – Device Debugging** and select Edit *project_name* …



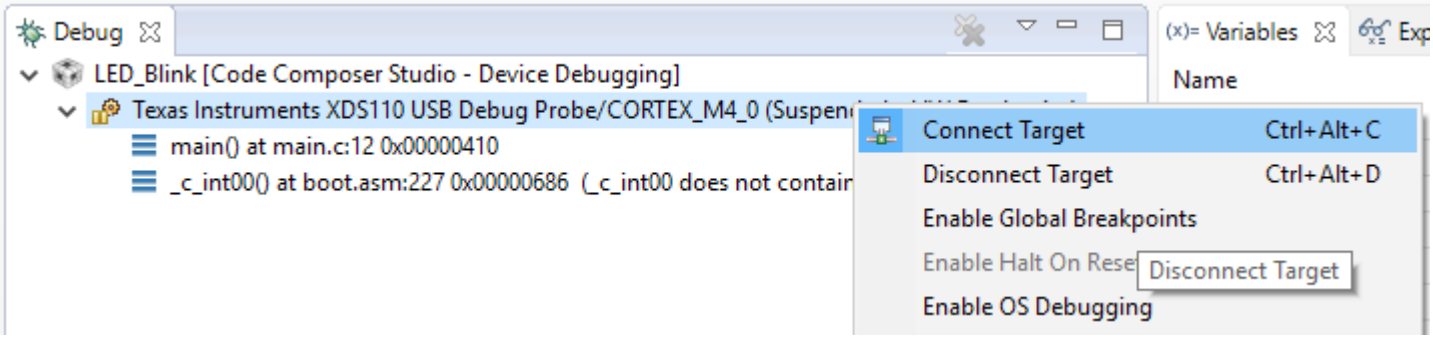
Step 3. Select the **Program** Tab and choose **Load symbols only**.

Click **Apply** and then **Continue**

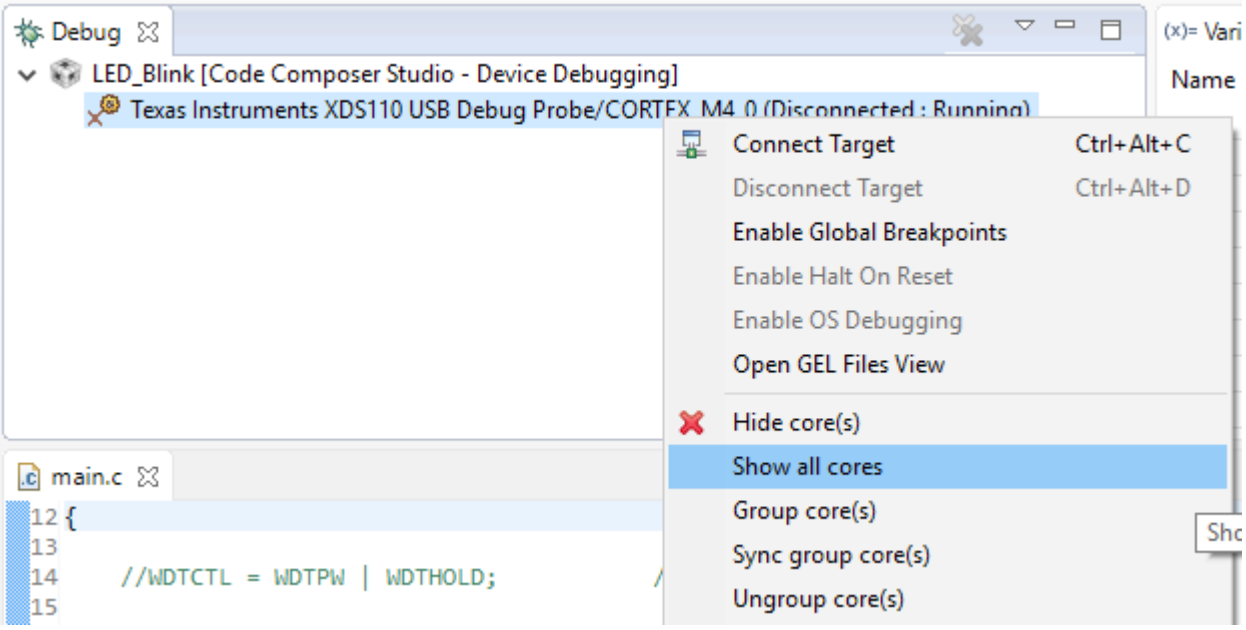
Try to program your MSP432 again by clicking on the Debug button like normal. Programming should complete with no error popup dialog.



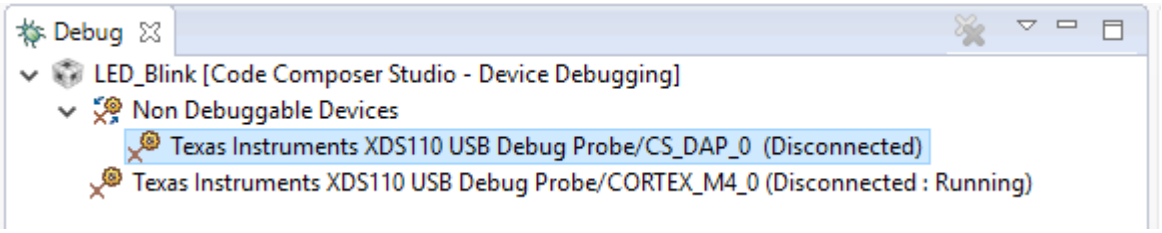
Step 4. Select the **XDS110 USB Debug Probe** from the Debug window, right click and select **Connect Target**



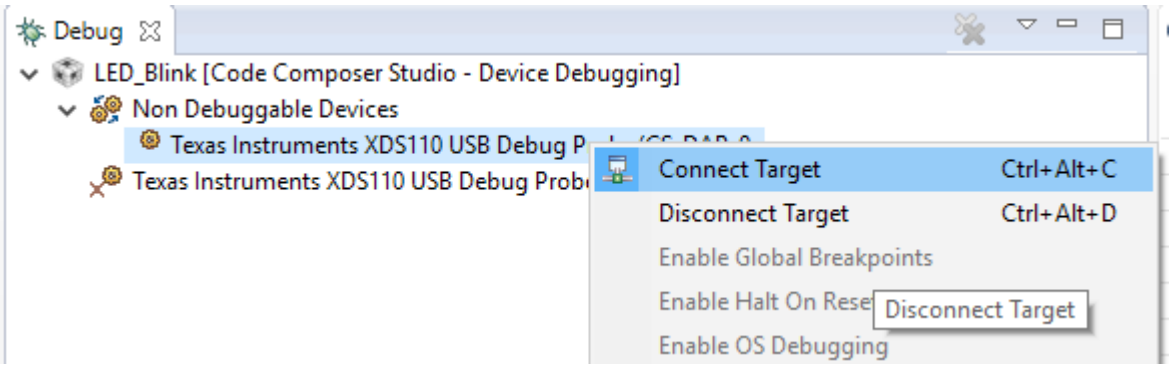
Step 5. After connecting, select the same **XDS110 USB Debug Probe**, right click and select **Show all cores**



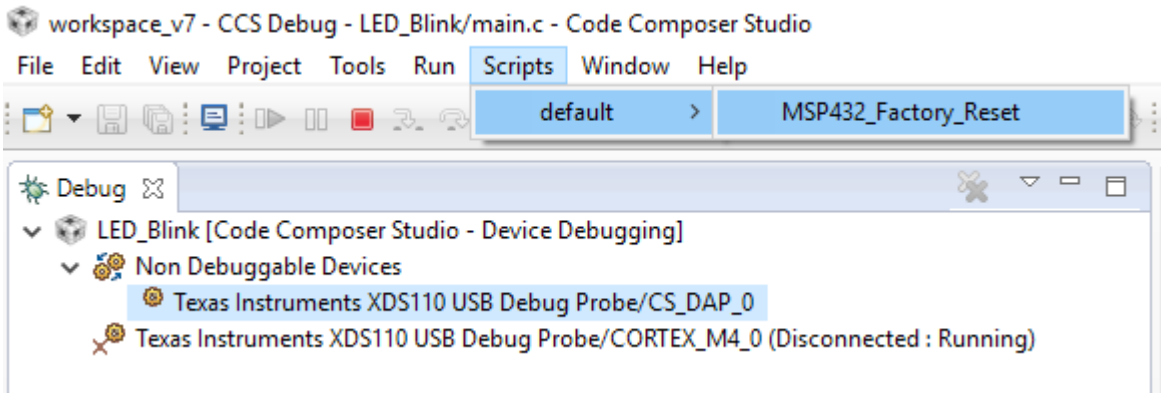
Step 6. A new device should show up on your debug window. **XDS110 USB Debug Probe / CS_DAP_0**



Step 7. Select **XDS110 USB Debug Probe / CS_DAP_0**, right click and select **Connect Target**

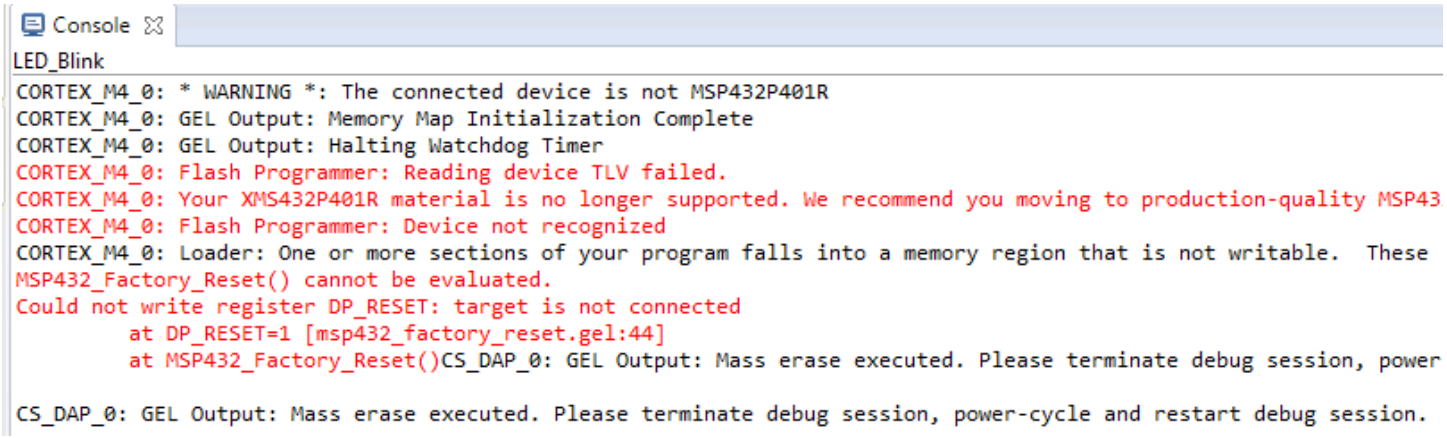


Step 8. After connecting to **XDS110 USB Debug Probe / CS_DAP_0**, the text "(Disconnected)" should no longer appear. Select the **XDS110 USB Debug Probe / CS_DAP_0**. Then go to the menu on Code Composer studio and select **Scripts → default → MSP432_Factory_Reset**



The console should signify the Factory reset executed with the line.

```
CS_DAP_0: GEL Output: Mass erase executed. Please terminate debug session,
power-cycle and restart debug session.
```



Step 9. If you had to go through Steps 1-3 first, you must now change the target settings back
Repeat steps 1 through 3 except choose **Load program** from the Edit Configuration dialog.

Step 10. Unplug your MSP432 from your computer, close your debug session on Code Composer Studio.
Plug your MSP 432 back into your computer and reprogram it with the debug button as normal.
Your MSP 432 should be detected and program as usual.

TN3 - MSP432 Clock System

Background

The MSP432 has multiple clock signals to allow the processor and peripherals to operate independently at a range of different frequencies. The system clock can also change operating frequencies during runtime to optimize power and performance. During periods of low or no activity the system can lower the clock frequency to save power and then ramp the frequency up when faster processing is needed. A portion of the MSP432 clock system is shown in Diagram 1 below.

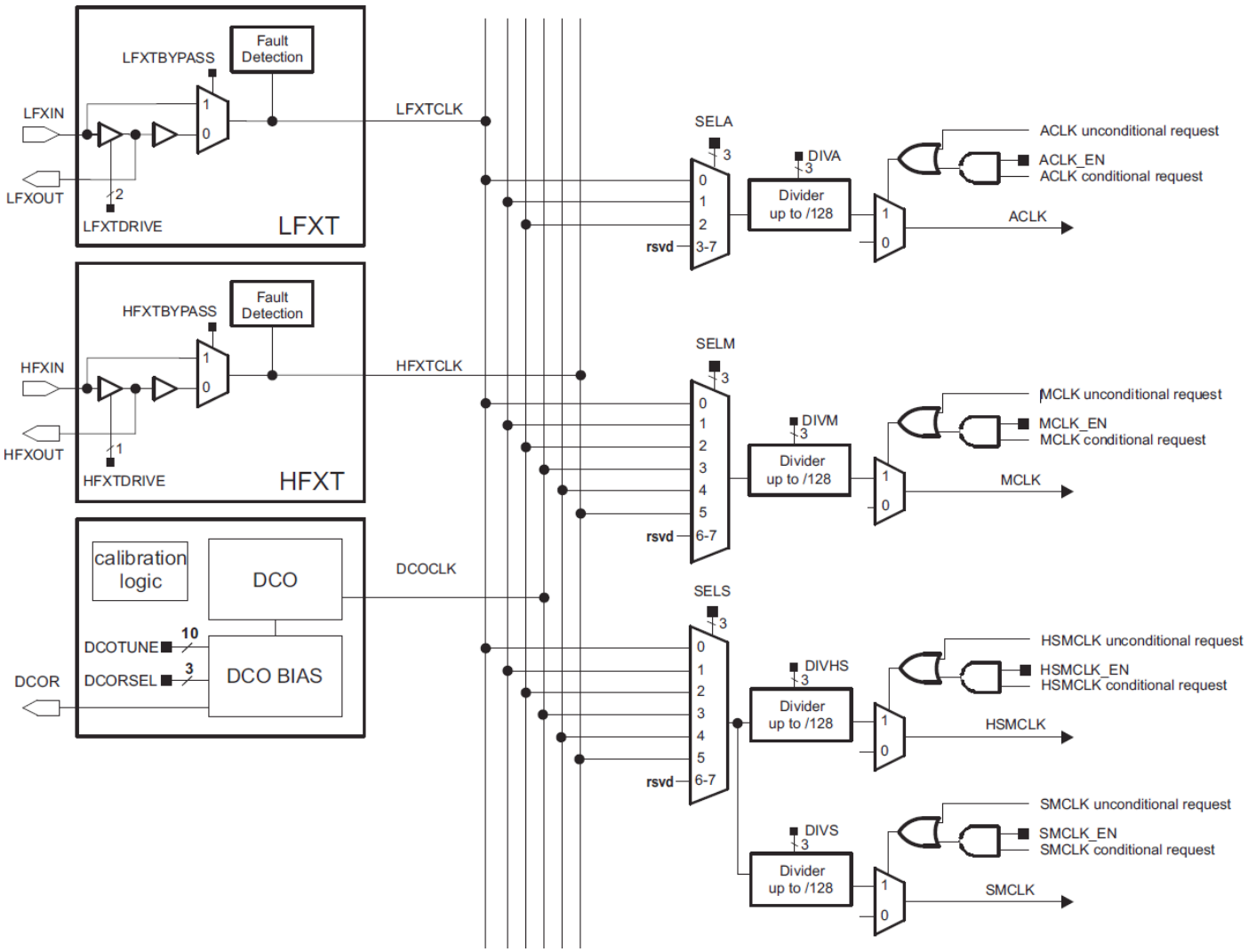


Diagram 1: MSP432 Clock System

The MSP432 has 7 clock sources available. LFXT and HFXT are connected to external crystal oscillators. The MSP432 development board uses a 32.768 kHz crystal for LFXT and a 48 MHz crystal for HFXT. The digitally controlled oscillator (DCO) can vary from 1 to 48 MHz. The internal very low power low frequency oscillator (VLO) operates at a set 9.4 kHz. The internal low power low frequency oscillator (REFO) operates at either 32.768 kHz or 128 kHz. The internal module oscillator (MODOSC) operates at a set 24 MHz. The internal system oscillator (SYSOSC) operates at a set 5 MHz.

The DCO on the MSP432 is adjustable in 6 tunable frequency ranges: 1.5, 3, 6, 12, 24, and 48 MHz. Details regarding the MSP432 clocking system and control of the DCO can be found in Chapter 5 of the MSP432 Technical Reference Manual. The following code can be used to modify the DCO frequency. Please see the note below regarding setting the DCO to 48MHz as special precautions are required for this.

```
// change DC0 from default of 3MHz to 12MHz.
CS->KEY = CS_KEY_VAL;                           // unlock CS registers
CS->CTL0 = 0;                                   // clear register CTL0
CS->CTL0 = CS_CTL0_DCORSEL_3;                   // set DCO = 12 MHz
// select clock sources
CS->CTL1 = CS_CTL1_SELA_2 | CS_CTL1_SELS_3 | CS_CTL1_SELM_3;
CS->KEY = 0;                                    // lock the CS registers
```

Running the MSP432 at 48 MHz

Setting the MSP432 DCO clock to 48 MHz requires additional steps beyond setting the clock system.

The power control manager (PCM) requires a higher Vcore voltage. If the MCLK is changed before ensuring the Vcore is sufficient will cause the MSP432 to issue a Power On/Off Reset (POR) that can cause a hardware lockup requiring a factory reset. The Vcore voltage takes time to transition and settle at the new voltage. No change in MCLK should occur until after this settling has occurred.

The flash controller also needs to adjust wait states for reading and writing for proper operation at higher clock frequencies. The flash controller is configurable in terms of the number of memory bus cycles it takes to service any read command. This allows the CPU execution frequency to be higher than the maximum read frequency supported by the flash memory. If the bus clock speed is higher than the native frequency of the flash, the access is stalled for the configured number of wait states, allowing data from the flash to be accessed reliably.

Code for setting Vcore to Level 1 for 48 MHz operation

```
/* Transition to VCORE Level 1: AM0_LDO --> AM1_LDO */
while ((PCM->CTL1 & PCM_CTL1_PMR_BUSY));
    PCM->CTL0 = PCM_CTL0_KEY_VAL | PCM_CTL0_AMR_1;
while ((PCM->CTL1 & PCM_CTL1_PMR_BUSY));
```

Code for setting flash controller wait states for 48 MHz operation

```
/* Configure Flash wait-state to 1 for both banks 0 & 1 */
FLCTL->BANK0_RDCTL = (FLCTL->BANK0_RDCTL &
     ~(FLCTL_BANK0_RDCTL_WAIT_MASK)) | FLCTL_BANK0_RDCTL_WAIT_1;
FLCTL->BANK1_RDCTL  = (FLCTL->BANK0_RDCTL &
     ~(FLCTL_BANK1_RDCTL_WAIT_MASK)) | FLCTL_BANK1_RDCTL_WAIT_1;
```

Code for setting DCO to 48 MHz operation

```
/* Configure DCO to 48MHz, ensure MCLK uses DCO as source*/
CS->KEY = CS_KEY_VAL ;          // Unlock CS module for register access
CS->CTL0 = 0;                   // Reset tuning parameters
CS->CTL0 = CS_CTL0_DCORSEL_5;  // Set DCO to 48MHz

/* Select MCLK = DCO, no divider */
CS->CTL1 = CS->CTL1 & ~(CS_CTL1_SELM_MASK | CS_CTL1_DIVM_MASK) |
    CS_CTL1_SELM_3;
CS->KEY = 0;                // Lock CS module from unintended accesses
```

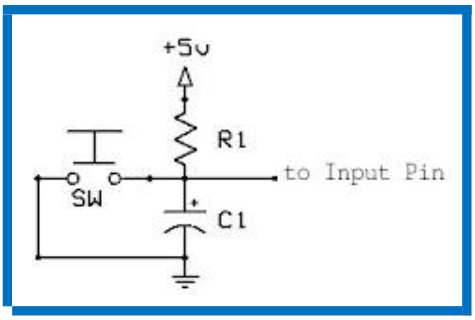
Using external crystal oscillators

For higher precision clocking with the MSP432 the system clocks can run from one of two external crystals. The low frequency crystal, connected to LFXT oscillates at 32,767 Hz. The high frequency crystal, connected to HFXT oscillates at 48 MHz. To use the high frequency crystal, the same power and flash settings are needed as outlined above. The code snippet below shows how to set the clock system to use the HFXT for operation. It is worth noting the HSMCLK is able to operate at 48 MHz while SMCLK is limited to a maximum frequency of 24 MHz. This is why the SMCLK was not set to use the DCO clock in the code example above. The SMCLK and HSMCLK have independent divisors so they can both use the same source oscillator and operate at different frequencies.

Code for using external 48 MHz crystal

```
/* Configure HFXT to use 48MHz crystal, source to MCLK & HSMCLK*/
PJ->SEL0 |= BIT2 | BIT3;         // Configure PJ.2/3 for HFXT function
PJ->SEL1 &= ~(BIT2 | BIT3);

CS->KEY = CS_KEY_VAL ;        // Unlock CS module for register access
CS->CTL2 |= CS_CTL2_HFXT_EN | CS_CTL2_HFXTFREQ_6 | CS_CTL2_HFXTDRIVE;
while(CS->IFG & CS_IFG_HFXTIFG)
    CS->CLRIFG |= CS_CLRIFG_CLR_HFXTIFG;

/* Select MCLK & HSMCLK = HFXT, no divider */
CS->CTL1 = CS->CTL1 & ~(CS_CTL1_SELM_MASK | CS_CTL1_DIVM_MASK |
    CS_CTL1_SELS_MASK | CS_CTL1_DIVHS_MASK) | CS_CTL1_SELM__HFXTCLK |
    CS_CTL1_SELS__HFXTCLK;
CS->KEY = 0;              // Lock CS module from unintended accesses
```

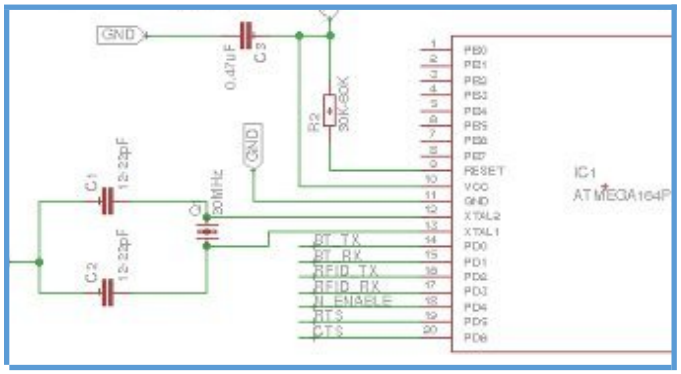
TN4 - Schematic Diagram Best Practices

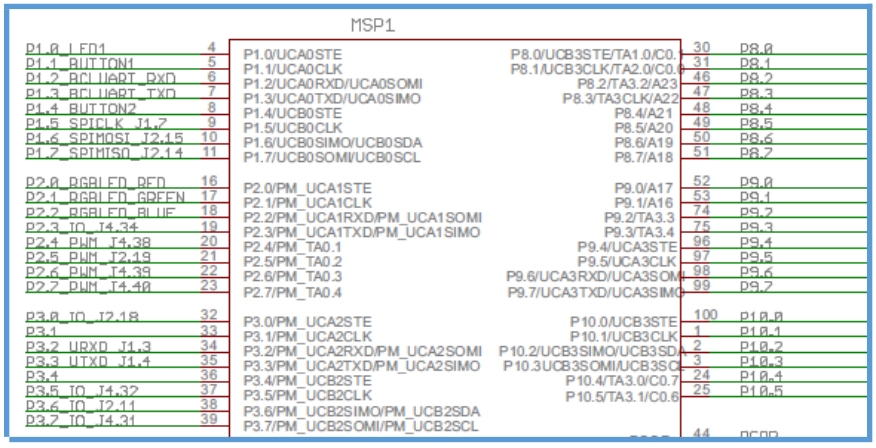


Chip Identifiers, Pin Numbers, and Signal Names

Label all chips with a unique identifier. The first symbol should be a letter (U for chips, C for capacitors, L for inductors, R for resistors, S for switches, etc.) The second symbol should be the unique number of the component. This will allow a to have identifiers such as U1, U2, C1, C2, R1, etc.

The pin number of the chip should be outside of the chip directly above or next to the line that represents the wire. The signal name specific to the chip, e.g. /CE, should reside inside the chip directly centered on the wire that leaves the chip. Signal names of wires should be centered over the wire some distance from the chip. Here are some examples. Note that the MSP432 LaunchPad schematic starting on p. 36 of the MSP432 LaunchPad User's Guide provides an example of a nicely drawn schematic.





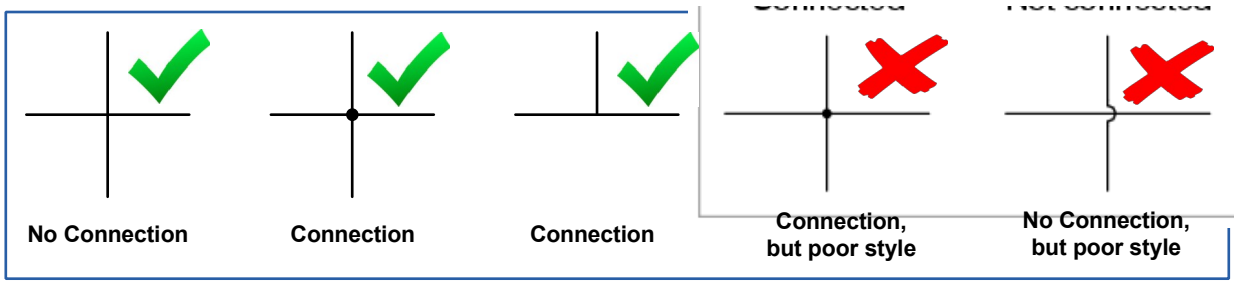
Signal names on chips should not necessarily follow the order of pins on the chip, rather be oriented to allow ease of reading. For example, you might put inputs on the left and outputs on the right. A schematic should let the reader's eye follow the flow of signals when possible.

Power Connections and Drawing Wires

Power connections should generally point upward on the sheet, ground connections downward.

Signals should be spaced uniformly on a schematic.

Do not show wires jumping over other wires. Do not use dots to connect wires unnecessarily. Please see the example below.



Layout Considerations and General Notes

Try to maintain come consistency with respect to chip sizes; don't draw 10 different sized rectangles and shapes on one schematic.

Put a title block with relevant information in the lower right-hand corner. A 1/4 inch border around a schematic is also nice.

Please do not fill components with a color and be sure that component edges are not too thick.

Additional Resources and Tools

Please see the attached link for additional thoughts on drawing schematics.

https://electronics.stackexchange.com/questions/28251/rules-and-guidelines-for-drawing-good-schematics

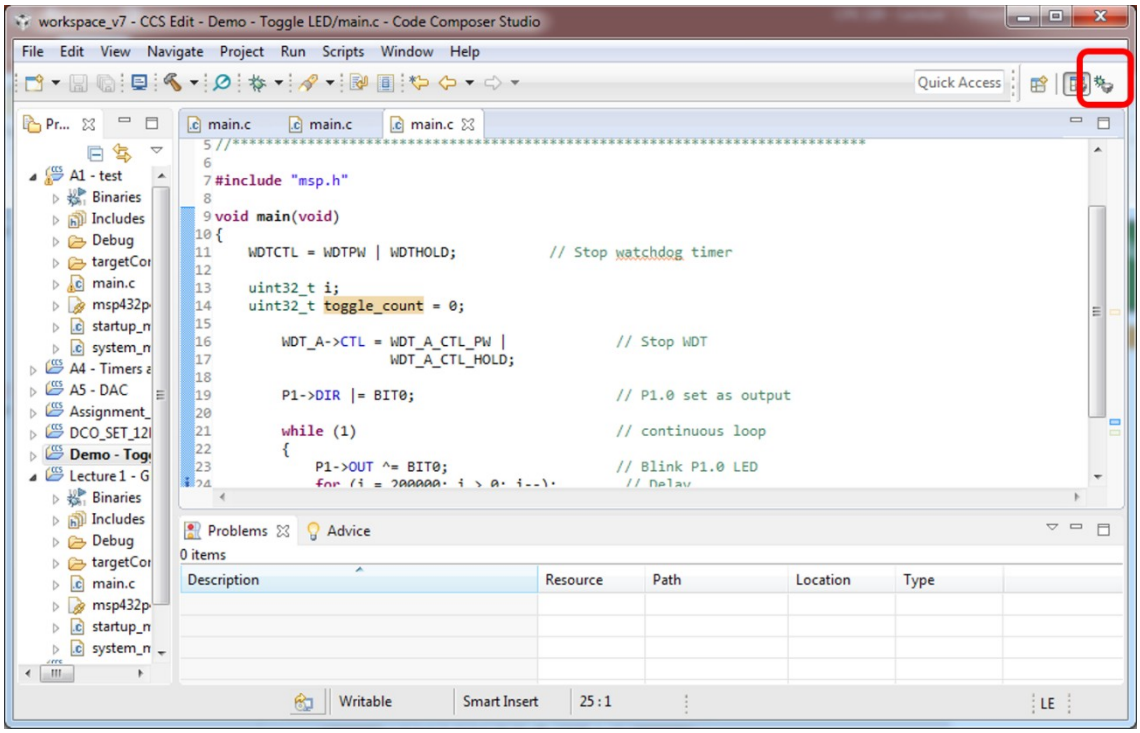
Here are a few tools you can use to draw schematics:

Visio – http://calpoly.onthehub.com
Circuit Lab - https://www.circuitlab.com/

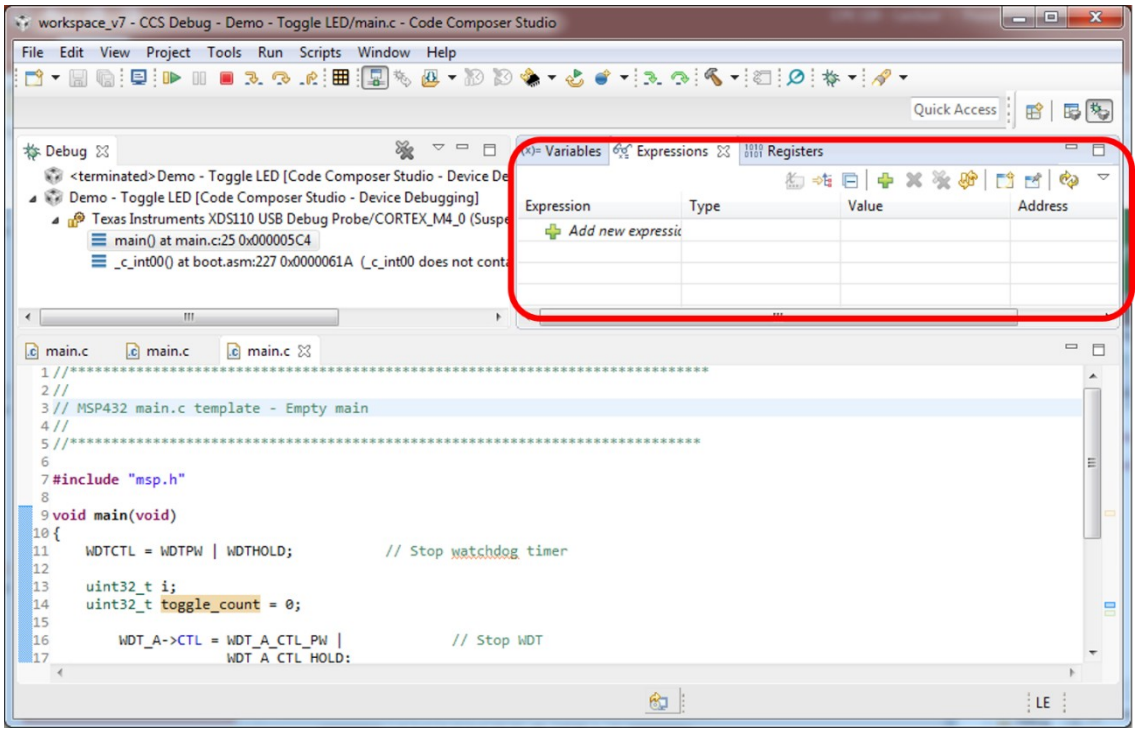
TN5 - Watching Expressions in Code Composer Studio

Here is a step-by-step procedure for watching expressions in Code Composer Studio.

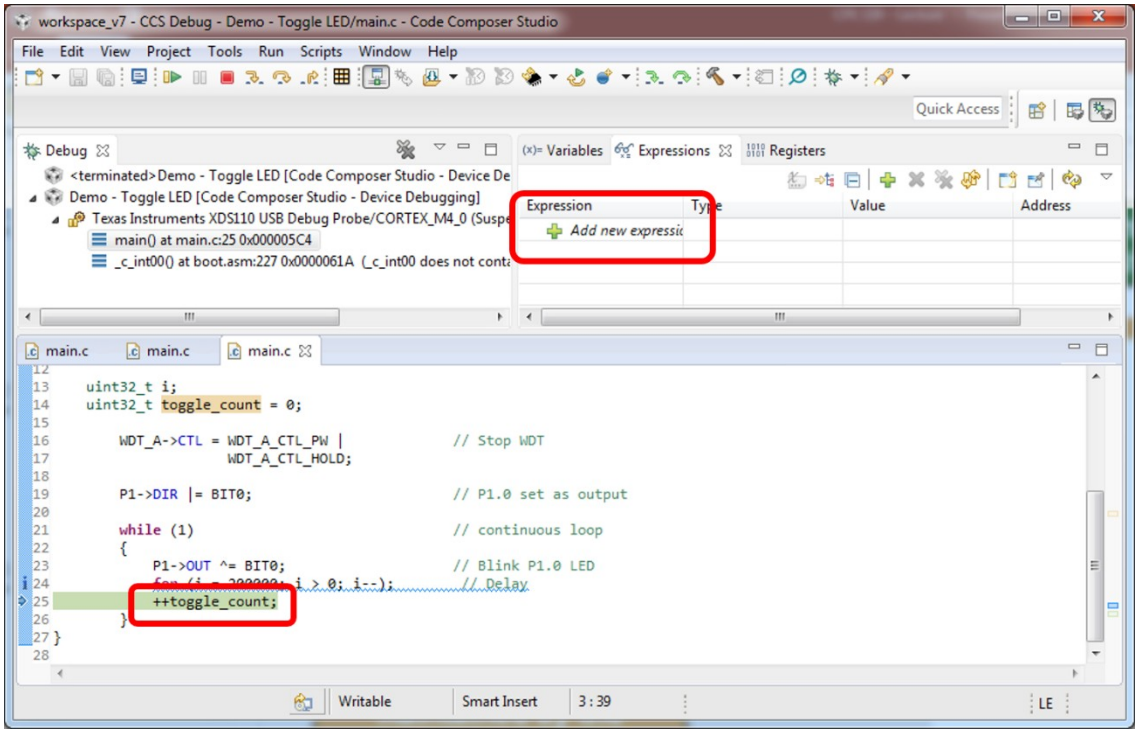
1. Enter debug mode in the project.



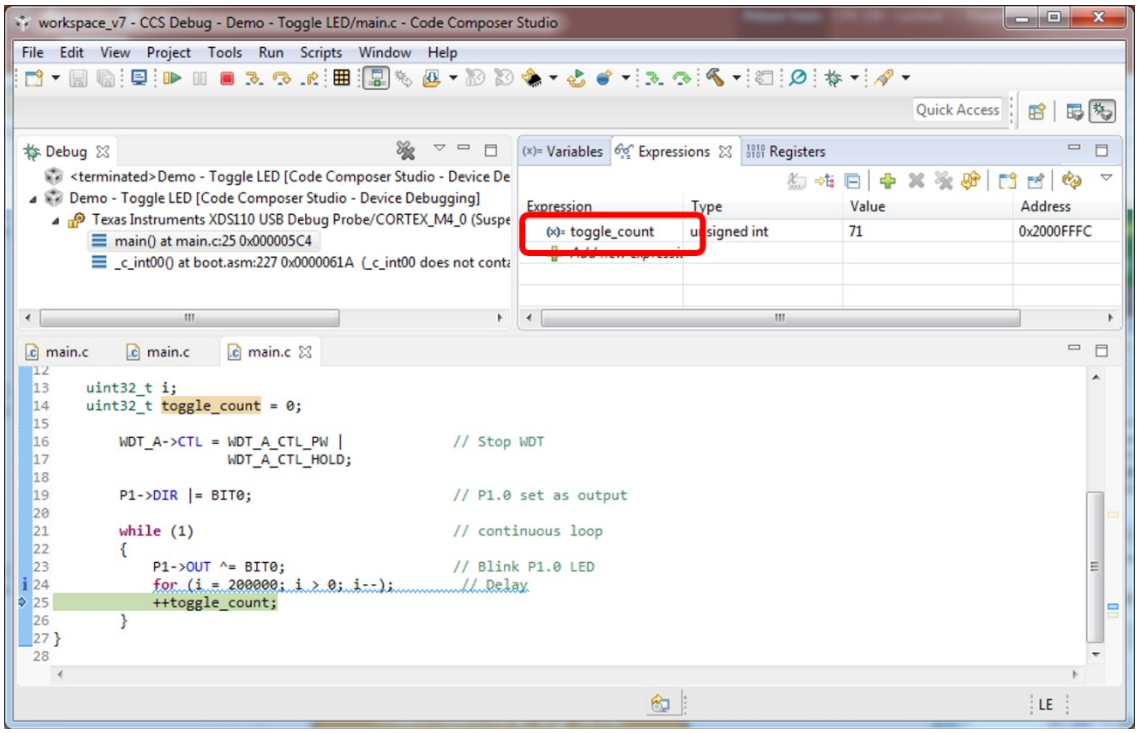
2. It may be necessary reset the perspective to show the debug pane.
   Select ***Window->Perspective->Reset Perspective***



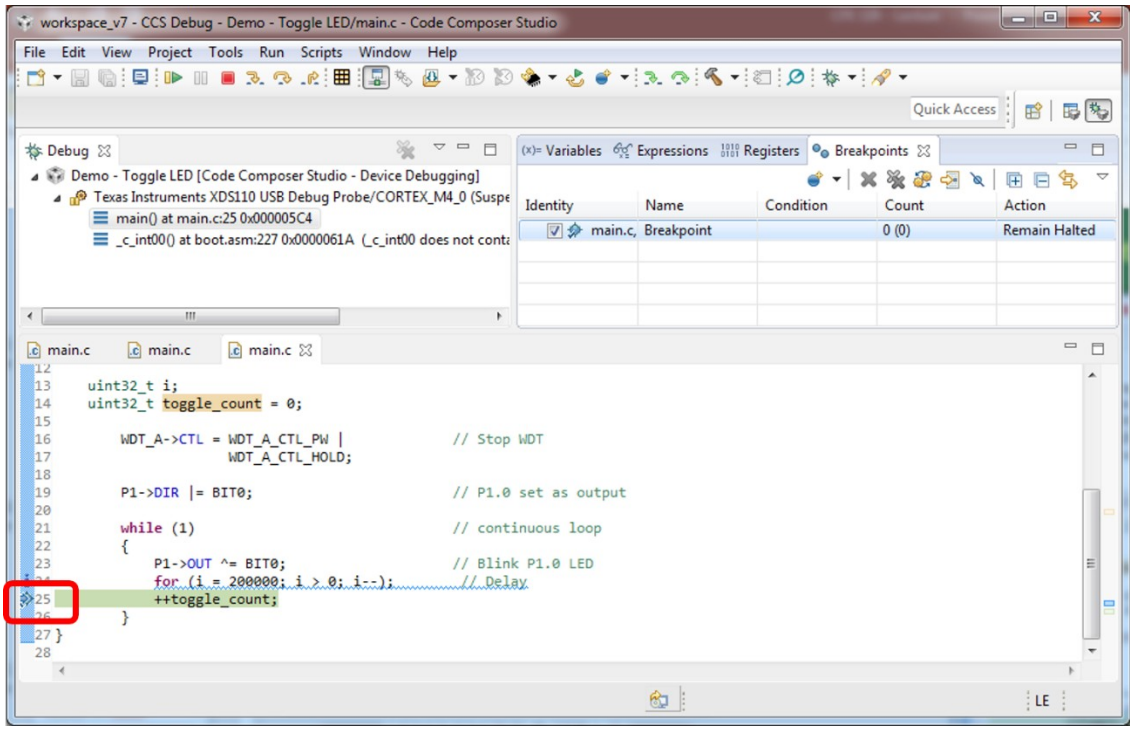
3. Click ***Add new expression*** and enter expression to watch (toggle count)



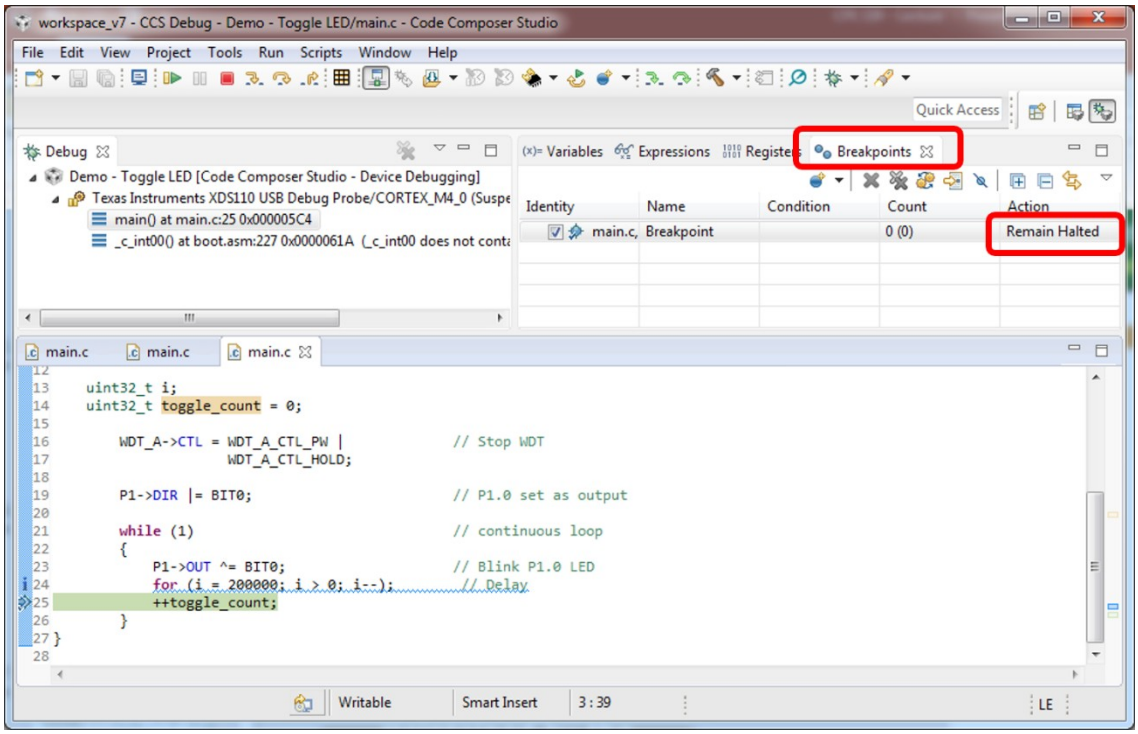
4. The expression toggle count is now ready to watch as the program executes.



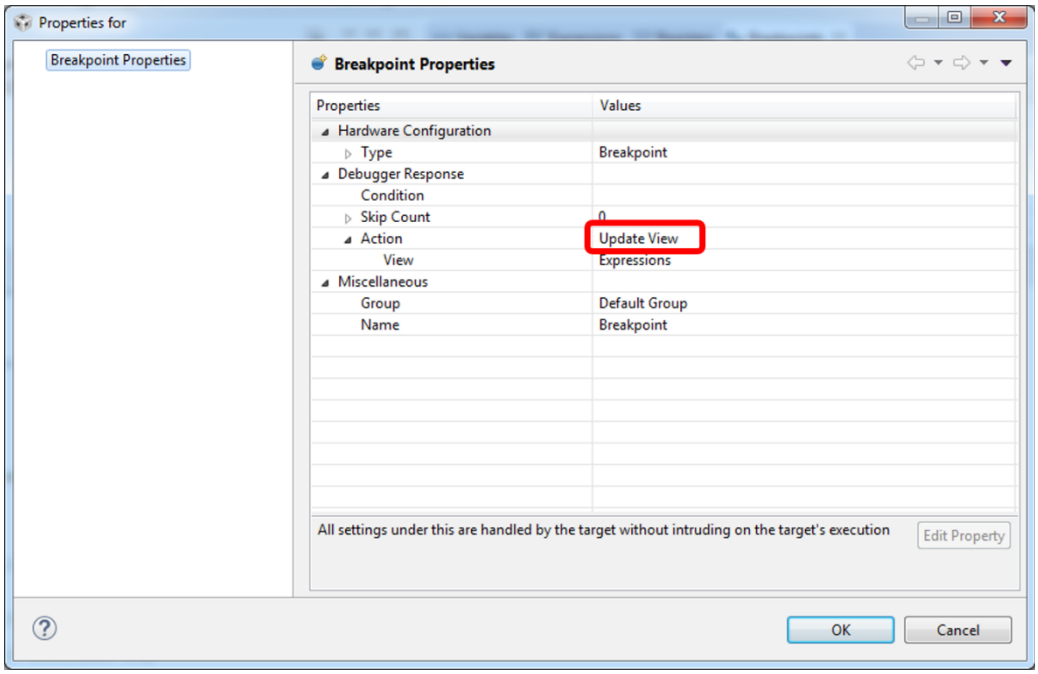
5. Right click in the left margin on a line of code near the expression.
   Select ***Breakpoint (code composer studio) -> Breakpoint***



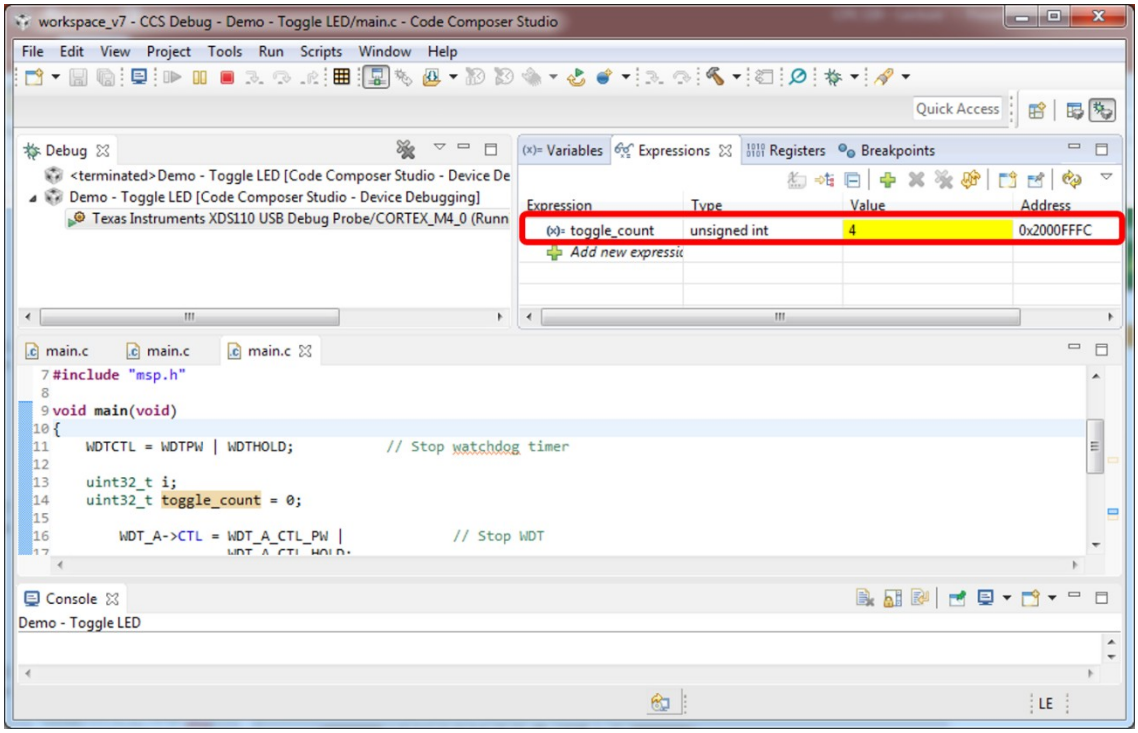
6. The Breakpoints tab will be added to the debug pane.
   Right click on ***Remain Halted*** and select ***Breakpoint Properties***



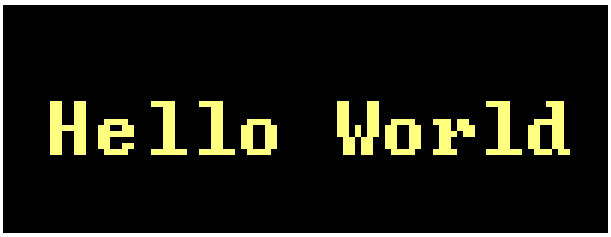
7. Set the action for the breakpoint to **Update View.** This will allow code to continue to execute while continuing to display the value of the expression in the view window.



8. Restart debug mode. The expression entered will update as it changes in software.



TN6 - VT100 Terminals



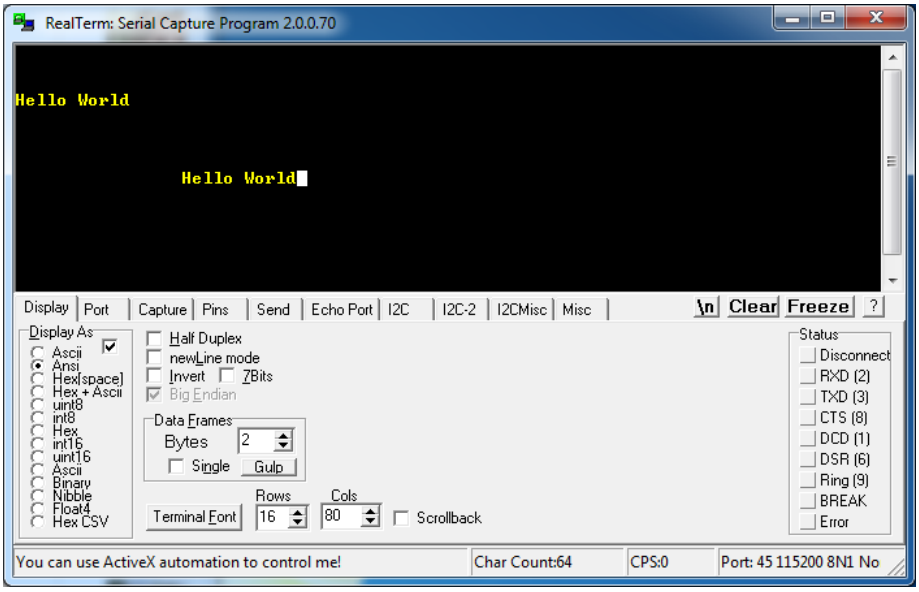
Background

Serial terminals have a long history in computer land (ComputerLand was a store you know – there used to be one on Monterey street in SLO right across from the waterbed store). Serial terminals originally started out as the primary method of gaining access to a time-shared computer system. A user would sit down to a terminal, be provided with a login prompt, login, and perform all work at the command line. Although this sounds fairly primitive, it was a great leap forward from having to enter a program via punch cards.

The VT100 video terminal, which was a physical computer monitor, provided ANSI escape codes for controlling the cursor, blinking text, etc. The VT100 was introduced in 1978 by Digital Equipment Corporation (DEC). There have been many compatible terminals used since the original VT100 but the VT100 is somewhat synonymous with a terminal that interprets escape codes for a variety of cursor and keyboard functions.

Realterm (and other Terminals)

Realterm provides VT100 compatibility by setting **Display As** to **Ansi**, as shown below.



As described above, escape codes (or sequences) are simply use of specific sequences of ASCII characters to instruct the terminal to do things like set the cursor to a specific location, blink text, etc. The general process is to use an escape code to place the cursor at a given location and then write the desired text that will show

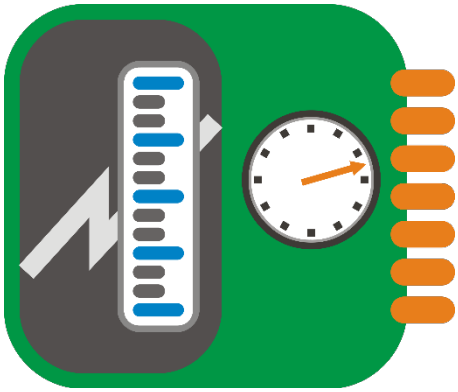
up at that location. The example above uses escape sequences to move the cursor down 3 lines, write “Hello World”, then move the cursor down 5 lines and over 5 spaces to the right, turn on blinking, and then write “Hello World” again. The escape codes and text transmitted to implement the screen above are:

```
Esc[3B          // move the cursor down 3
Hello World
Esc[5B          // move the cursor down 5
Esc[5C          // move the cursor 5 to the right
Esc[5m          // set text to blinking
Hello World
```

Note that Esc is simply 0x1B. A quick search on Google will provide many documents describing how to use escape sequences.

A handy table can be found here http://ascii-table.com/ansi-escape-sequences-vt-100.php

TN7 - Calibrating ADC / Sensor



Background

The analog input on the MSP432 will ideally follow the formula – – — —

$$N_{\text{ADC}} = 2^{\text{ADC\_BITS}} \, x \, \frac{V_{\text{in}} - V_{\text{R}-}}{V_{\text{R+}} - V_{\text{R}-}}$$

The resolution of your input, the value of 1 LSB (least significant bit) is

$$1\,LSB = \frac{V_{\text{R+}} - V_{\text{R}-}}{2^{\text{ADC\_BITS}}}$$

Unfortunately most analog inputs will not perfectly match the ideal formulas above and will need a further calibration formula to get an accurate value from the ADC value. The simplest calibration is to provide a linear approximation across the range of wanted values. If your application only needs to record voltages between 1.2 V and 1.8 V then being calibrated to accurately measure 2.5 V is unimportant, so the calibration should be focused on the desired range of 1.2 to 1.8. Calibration requires you to have a reliable method to measure whatever the sensor is measuring, and the calibration can only be as accurate as the measurement equipment it is being calibrated with. Depending on the sensor, this can sometimes be the most difficult aspect of creating a calibration.

Some sensors will provide a more robust calibration formula to map the digital reading to accurate values within an acceptable error tolerance without need for outside verification. When no reliable equipment can be found for corroborating accurate measurements against, this may be an acceptable alternative. Whenever possible, all sensor data should be verified against known reliable equipment. Many sensors will have a linear relationship between what it is sensing or measuring and its output signal. Sensors that do not follow a linear relationship will typically detail a more appropriate curve fitting equation in their documentation. The type of curve will affect how many data points is needed for creating an accurate calibration.

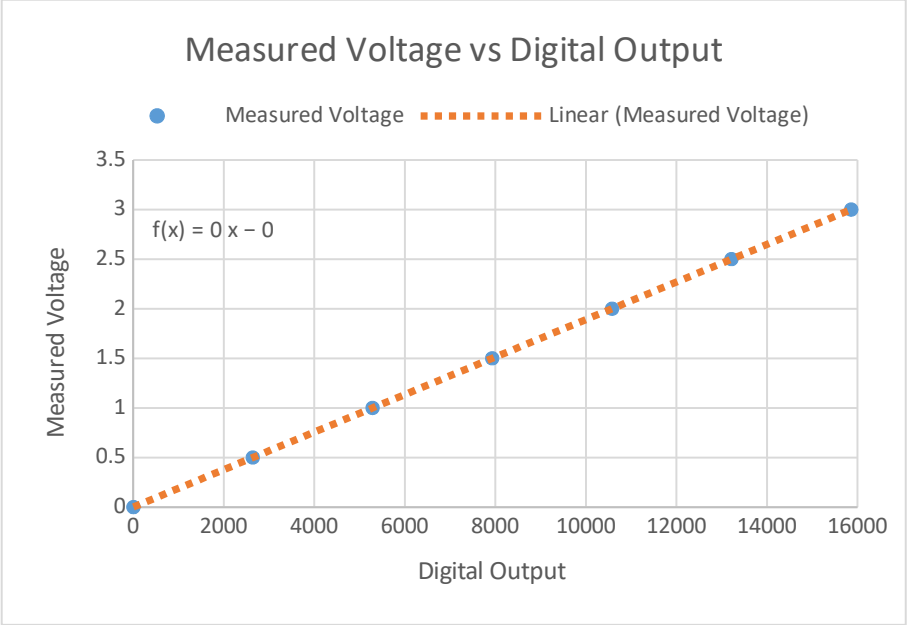
Linear Approximation Calibration Methodology

Sensors that follow a linear shape can be easily calibrated by taking a few measurements and creating a linear regression.

$$Measurement = Digital\_Value \times m + b$$

The first step is to take recordings of several known points over the range of interest. As way of example, the below data points were used in calibrating the ADC14 with a Vref of 3.1 V.

Digital Output	Measured Voltage (V)
6	0
2642	0.5
5288	1
7930	1.5
10579	2
13215	2.5
15861	3



The linear approximation creates a calibration equation for transforming the digital output from the ADC to measured voltage values.

$$Measurement = 1.892 \times 10^{-4} \times Digital\_Value - 5.266 \times 10^{-4}$$

Digital Output	Measured Voltage (V)	Calibrated Value (V)
6	0	0.000609
2642	0.5	0.499340
5288	1	0.999963
7930	1.5	1.499829
10579	2	2.001020
13215	2.5	2.499751
15861	3	3.000375

Linear Approximation without Floats

The above equation unfortunately creates a need for floating point variables. This need for floats can be removed if the value is scaled by multiple orders of magnitude, essentially changing the magnitude of the units. For the above data, the measurement could be saved as µV rather than volts. The new calibration equation becomes

$$Measurement = 189 \times Digital\_Value - 527$$

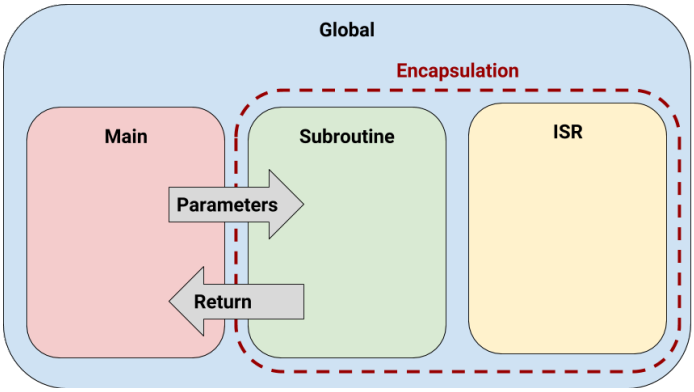
Digital Output	Measured Voltage (V)	Calibrated Value (µV)
6	0	607
2642	0.5	498811
5288	1	998905
7930	1.5	1498243
10579	2	1998904
13215	2.5	2497108
15861	3	2997202

If deciding to scale the measurements be sure to not scale to values beyond the size for integers. If only positive values will be measured, unsigned integer variables will double the size available.

Nonlinear Sensor Response

Some sensors may have a nonlinear response. This can be especially pronounced at the extremes of the measurement range. This can be corrected for by having multiple calibration equations that get applied for different ranges of output values. Ultimately it is up to the engineer to verify the sensor device is accurate for the full range specified.

TN8 - ISR Communications



Variable Scope – main and the ISR live in two different worlds

Scope in a computer program refers to the visibility of variables. Variables defined inside of main can't be seen (or accessed) inside of interrupt service routines and vice versa. Given this, the main loop of a program and ISRs which it may interact with require a method to pass data between with each other. Global variables are an easy way to accomplish this. Here is a quick example of how a global variable may be used to let main know that an interrupt has occurred.

```
static long long interrupt_flag = 0;     // an 8-byte flag!
void main {

   while(1){
      if (interrupt_flag)
         // do something
      else
         // do something else
   }
}

void ISR()
   interrupt_flag = 1;
return
```

Protection of Variables

Consider if a variable being written to in an ISR or read in another part of the program is large enough so that multiple assembly instructions are required to modify it, making it not an atomic data element. Such elements can be corrupted if the read or write of them is interrupted and the element modified prior completion of a read or write. Such actions could result in one half of the element being from the previous value and the other half of the element from a more recent value. Think of a sentence interrupted half way through with one half of the sentence arising from one part of a conversation and the other half from a bit later in the conversation. The end result is a corrupt and unintelligible sentence.

Non-atomic data elements can be protected by disabling interrupts prior to accessing the element and then re-enabled at the completion of reading or writing the element. Here is the example from above with this sort of protection added.

```
static long long interrupt_flag = 0;     // an 8-byte flag!
void main {

   __disable_interrupts();
   while(1){
      if (interrupt_flag)
         // do something
      else
         // do something else
   }
   __enable_interrupts();
}

void ISR()
   __disable_interrupts();
   interrupt_flag = 1;
   __enable_interrupts();
return
```

Encapsulation

Encapsulation may be used to hide (or protect) variables within specific functions which are used to check, set, and clear variables. Encapsulation effectively means creating a driver for a given peripheral that handles all of the details associated with communicating data two and from the peripheral. Here are steps to implement encapsulation in C.

1. Create a separate C file for the driver. Define the variables/flags that you want to use as static globals within this C file. Note that defining these variables as static makes them visible only within this file.
2. Write access functions such as check_flag(), clear_flag(), set_flag(), set_data(), get_data() to allow access to this data.
3. Create a .h file with headers for all of the access functions.
4. Include the .h file in your main C file.
5. Use the access functions both in main and in your ISR anywhere you want to check, use, or change any of your protected variables

main.c

```
#include <msp.h>
#include "mydriver.h"

void main {
uint32_t localValue;
   while(1){
      if (check_flag()) {
         // something important happened!
    localValue = get_data();
      }
         // nothing happened yet
   }
}
```

mydriver.c

```
static uint32_t driverValue = 0;
static uint32_t driverFlag = 0;

uint32_t check_flag(void) {
return driverFlag;
}

uint32_t get_data(void) {
return driverValue;
}

void driver_ISR(void) {    // interrupt on peripheral device
...
driverValue = device_register; // read data from device
driverFlag = 1;                // set flag value
...
}
```

mydriver.h

```
uint32_t check_flag(void);
uint32_t get_data(void);
void driver_ISR(void);
```

TN9 - #include Files

#include "msp.h"

Background

A header file may be used to collect function declarations within a specified file available to the C compiler, allowing specific functions to be available for reuse at a later time. Consolidating functions in this fashion is beneficial from a code reuse standpoint; code written for one module can be reused within other modules. Additional benefits are the reduction of clutter within C modules and reduced software maintenance due to the fact that code can be written and tested once, and then used repeatedly in other functions.

Example

Suppose you have a module called **`blink()`**, which causes an LED to toggle on and off. You've written **`Blink()`** once, have thoroughly tested it, and want to have it reusable in other modules. Here are the required steps:

1. Create a header file such as **`blink.h`** which has the function declaration for **`blink()`** within it. Note that such a header file can have more than one function declaration within it. For example, the contents of **`blink.h`** might be something like this:

```
void blink(void);
```

2. Create a **`.c`** file which has the function that implements blink within it. The contents of **`blink.c`** might look something like this:

```
void blink (void) {
   // code that blinks the LED here
}
```

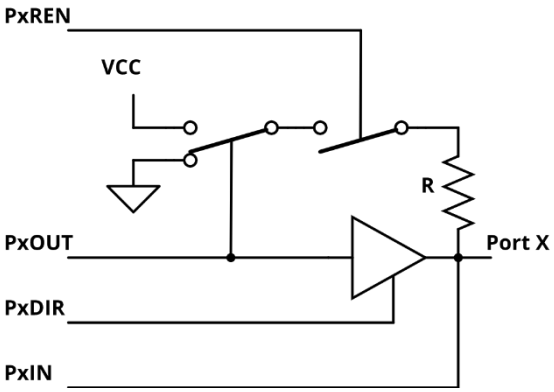
3. Add `#include` **`"blink.h"`** in near the top of **`main.c.`** The compiler will search the local project directory for **`blink.h`** and **`blink.c`** during compilation.

4. Use the function **`blink()`** within main or any other function for which **`blink.h`** has been included.

When you want to reuse your **`blink()`** function, you can add the **`blink.h`** and **`blink.c`** files to this new project and add `#include` **`"blink.h"`** to the **`main.c`** file as before.

Problems of Interest

POI 1 - GPIO Operations



Problem 1 – Bit Toggling

Please show how you would connect an LED to **Port 2.6** and a Switch to **Port 2.7** on the MSP432 on the LaunchPad, including pin numbers. Write a complete C program which continuously polls the switch and illuminates the LED when the switch is on without disturbing anything else on Port 2.

```
int main(void) {
    WDTCTL = WDTPW | WDTHOLD;    // Stop watchdog timer

}
```

Problem 2 – Bit Manipulation

Write a complete C program that continuously reads bit 3 or Port 1 and manipulates Port 2.

- If bit 3 or Port 1 is a '1', then it write a '1' to bit 1 of Port 2 and toggle bit 2 of Port 2.
- If bit 3 of Port 1 is a '0', then it write a '0' to bit 1 of Port 2.

Your program should not affect any other port bits in the system.

Problem 3 – Bit Manipulation with Various Duty Cycles

Please design a *complete* system including schematic diagram and C code which has switches on P2.5 and P2.4 an LEDs on P1.5 and P1.4. P1.5 toggles at 100 Hz with a 50% duty cycle if P2.5 is low. P1.4 toggles at 200 Hz with a 25% duty cycle if P2.4 is low. Please note that only one of P2.5 and P2.4 will be low at a time.

```
#include <msp.h>
int main(void) {
   WDTCTL = WDTPW | WDTHOLD;        // Stop watchdog timer
   Set_16MHz();                     // Set MSP to run at 16 MHz
}
```

Problem 4 – Bit Manipulation using Delay Cycles

Please design a *complete* system including schematic diagram and C code which has a switch on P1.3 and LEDs on on P2.1 and P2.2. Your program should continuously read bit 3 of PORT1 and use __delay_cycles() for timing.

- If bit 3 or PORT1 is a '1', then it writes a '1' to bit 1 of PORT2 and toggles bit 2 of PORT2 at a 1000 Hz rate.
- If bit 3 of PORT1 is a '0', then it writes s '0' to bit 1 of PORT2 and holds bit 2 of PORT2 to a '0'.

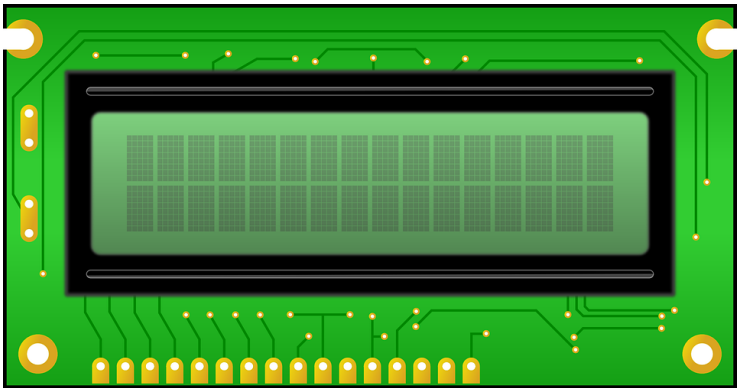
```
#include <msp.h>
int main(void) {
   WDTCTL = WDTPW | WDTHOLD;        // Stop watchdog timer
   Set_16MHz();                     // Set MSP to run at 16 MHz
}
```

Problem 5 – Timed Button Presses

If the MSP432 is setup with SMCLK running at 12 MHz, setup the C code below to create a timer based system to measure the time between button presses connected to P1.6. The time should be converted to ms and saved in the integer variable TimeMs

```
#include <msp.h>
int TimeMs;
int main(void)
{
 WDTCTL = WDTPW + WDTHOLD; // Stop WDT
 SMCLK_12MHZ();            // SET SMCLK AT 12MHZ

}

void PORT1_IRQHandler(void)
{

}
```

POI 2 - LCD Control



Problem 1 – LCD Schematic

Please create a detailed schematic diagram of how the MSP432 would be properly connected to the LCD display to be used nibble mode (same as Project 1). Use the following pin assignments:

E = P1.0	DB7 = P1.7	DB5 = P1.5
RS = P1.1	DB6 = P1.6	DB4 = P1.4
RW = P1.2		

Problem 2 – LCD Code Writing

Please write C code that will control the LCD to place an uppercase 'C' in the 5th location of the second row of the display. Please use .h defined BIT2, BIT1, and BIT0 for manipulating the required port bits. Comment your code for readability. Assume the MSP432 is running at 12 MHz and the LCD is connected in 8 bit mode with the following port assignments:

E = P2.0	DB7 = P1.7	DB3 = P1.3
RS = P2.1	DB6 = P1.6	DB2 = P1.2
RW = P2.2	DB5 = P1.5	DB1 = P1.1
	DB4 = P1.4	DB0 = P1.0

You can assume that the display has already been initialized, cleared, and is ready to for receiving characters.

Problem 3 – Function Set in 4-bit Mode

Consider that your MSP432 is running at 16 MHz is connected to the LCD as such:

```
E    = P1.0      DB7 = P1.7     DB5 = P1.5
RS   = P1.1      DB6 = P1.6     DB4 = P1.4
RW   = P1.2
```

Please draw a complete schematic diagram and write a snippet of C code that would execute the ***Function Set*** command for Nibble mode (4-bit interface), two line display, and 5x8 dots.

Problem 4 – Write Character in 8-bit Mode

Consider that your MSP432 is running at 16 MHz is connected to the LCD as such:

```
E   = P2.0     DB7 = P1.7    DB6 = P1.6   DB5 = P1.5    DB4 = P1.4
RS  = P2.1     DB3 = P1.3    DB2 = P1.2   DB1 = P1.1    DB4 = P1.0
RW  = P2.2
```

Please draw a complete schematic diagram and write a snippet of C code that would write a letter 'X' to the third location from the left on the bottom row of the LCD display. Please consider that your LCD has previously been initialized to run with two line display, 5x8 dots, display on, etc.

Problem 5 – LCD Timing Diagram

Draw the timing diagram for sending an ascii 'a' (0x61) to the LCD screen in 8 bit mode. Be sure to include RS, RW, and E. Include a scale for voltage and mark any important timing transitions.

POI 3 - Timer A

Problem 1 – Timer Control with Interrupts 1

Please complete the following source code to utilize interrupts and Timer A in continuous up-count mode to cause the MSB of Port 0 to toggle with an ON (logic '1') time of 900 us and an OFF (logic '0') time of 1.4 ms. The LED should go low to start. Consider that your MSP is running at 1 MHz.

```
int main(void) {
    WDTCTL = WDTPW | WDTHOLD;     // Stop watchdog timer
    DCO_1_MHz();                  // Set DCO to 1 MHz
}

// Timer A0 interrupt service routine
void TA1_0_IRQHandler(void) {

}
```

Problem 2 – Timer Control with Interrupts 2

Please complete the following source code to utilize interrupts and Timer A in continuous up-count mode to cause the MSB of Port 1 to toggle with an ON (logic '1') time of 500 us and an OFF (logic '0') time of 700 us. The LED should go low to start. Consider that your MSP is running at 16 MHz.

```
int main(void) {
    WDTCTL = WDTPW | WDTHOLD;     // Stop watchdog timer
    DCO_16_MHz();           // Set DCO to 16 MHz
}

// Timer A0 interrupt service routine
void TA1_0_IRQHandler(void) {

}
```

Problem 3 – Low Frequency Crystal Calculations

Suppose your LaunchPad had a 210.5 KHz crystal installed and Timer A was set to use ACLK with IDx set to 11 (ID_3). What is the absolute lowest frequency square wave you could generate if you were to toggle a port bit as output once per interrupt? Note that software counters may not be utilized for this example.

Problem 4 – Timer A Calculations

Assume that SMCLK was set to 16 MHz and TimerA is set for repeating up-count with no divided clock cycles

a. What would be the effect of setting CCR0 to 25,000 upon timer initialization and startup and then never changing it again?
b. Please draw a timeline and annotate with specific time values showing when interrupts would occur with t=0 being the point where the timer is initialized and started.

Problem 5 – Bit Toggling with SMCLK = 16 MHz

Please complete the following source code to utilize interrupts and Timer A in continuous up-count mode to cause the MSB of Port 1 to toggle with an ON (logic '1') time of 10 ms and an OFF (logic '0') time of 200 ms. The LED should go low to start. Consider that your MSP is running at 16 MHz, that you should not use any data types longer than 16 bits (no long ints), and that a top score on this problem will interrupt as infrequently as possible.

```
int main(void)
{
  WDTCTL = WDTPW + WDTHOLD;
  Set_16_MHz();
}

// Timer A0 interrupt service routine
void TA1_0_IRQHandler(void) {

}
```

Problem 6 – Bit Toggling with ACLK = 32,768 Hz

Please set up a timer on ACLK with a crystal at 32,768 Hz that will toggle the MSB of Port 1 ON (logic '1') for 12 hours and an OFF (logic '0') for 24 hours, repeating. The port bit should go low to start. Consider that your MSP is running at 32,768 Hz, that you should not use any data types longer than 16 bits (no long ints), and that a top score on this problem will interrupt as infrequently as possible.

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

// Timer A0 interrupt service routine
void TA1_0_IRQHandler(void) {

}
```

Problem 7 – Timer Control without Interrupts

If the MSP432 is setup with SMCLK running at 8 MHz, develop the C code to create a PWM at 1 KHz and 25% duty cycle waveform. No interrupts can be used in your code.

```
int main(void) {
    WDTCTL = WDTPW | WDTHOLD;     // Stop watchdog timer
    SMCLK_8_MHz();          // Set SMCLK to 8 MHz


}
```

POI 4 - SPI and DAC

Problem 1 – SPI Data Transmission Time Calculations

Consider that you have an SPI port configured with the following setup. SMCLK = 1 MHz and ACLK = 32,768 Hz. How long would it take to transmit a single byte out of the SPI port, disregarding setup times with CE low, etc.? Please draw a picture and show your work.

```
UCB0CTL0 |= UCCKPL + UCMSB + UCMST + UCSYNC;
UCB0CTL1 |= UCSSEL_1;
UCB0BR0 |= 0x40;
UCB0BR1 |= 0x01;
UCB0CTL1 &= ~UCSWRST;
```

a. 9.76 ms
b. 78.1 ms
c. 0.320 ms
d. 2.56 ms

Problem 2 – DAC Interfacing and Operation

a. Please draw a detailed schematic diagram (pin numbers, signal names, etc.) showing how the MSP 432 would be interfaced to the 4921 DAC (as used in class) using the USCI A channel.
b. What 16-bit HEX value would you send to the 4921 DAC with a reference voltage of 3.5V to cause it to output 3.0 Volts?
c. Suppose you were required to use an MCP 4901 DAC with gain set to 1 and Vref = 3.5. What analog voltage would be output with a hex value of 0x0A as the digital input?

Problem 3 – DAC Control Word

a. Please fill in the bits below that are sent to the DAC via TXBUF with 1s and 0s if DriveDAC was called with DriveDAC(200) and being used with the 4921 DAC, and Gain=2.

D15														**D0**

b. Please fill in the control byte that is sent to the DAC via TXBUF with 1s and 0s if DriveDAC was called with DriveDAC(100) and being used with the 4921 DAC, and Gain=1.

D15															D0

c. Given an MCP 4911 DAC with gain set to 1 and Vref = 2.5V. What analog voltage would be output with a hex value of 0x3A as the digital input?
   a. 141.74 mv
   b. 141.60 mv
   c. 78.4 mv
   d. 78.0 mv

d. What HEX value would you send to the MCP4921 DAC to cause it to output 3.0 V if gain were set to 2. Consider that the DAC has sufficient Vcc to support such an output and Vref = 3.5V.
   a. $DB7_{16}$
   b. $DB6_{16}$
   c. $6DB_{16}$
   d. $6DC_{16}$

Problem 4 – SPI DAC System

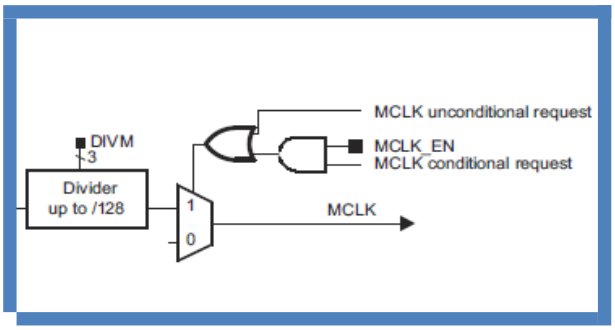
Please design a complete system, including schematic diagram and C code, which:

a. Initializes the SPI communication to run at 12 MHz
b. Sends the necessary commands and data to the DAC to output the voltage OutVolt

Consider that the DCO and SMCLK is set to 24 MHz. Run DAC unbuffered with gain = 1.


```
uint8_t OutVolt;
int main(void)
{
      WDTCTL = WDTPW + WDTHOLD; // Stop WDT
      SMCLK_24MHZ();            // SET DCO and SMCLK AT 24MHZ
}
```

POI 5 - Clock Module



Problem 1 – DCO vs. Crystal Oscillator

Please give two advantages of using a DCO-based clock instead of an external crystal.

a.
b.

Problem 2 – MCLK at Slow Clock Rates

What is the slowest speed that MCLK could be configured to be with the MSP432?

Problem 3 – Advantages of External Crystal

Please give two advantages of using an external crystal instead of SMCLK.

a.
b.

Problem 4 – ACLK at Slow Clock Rates

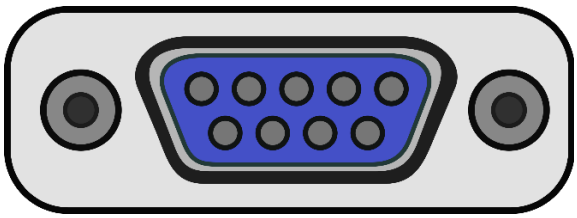
What is the slowest speed that ACLK could be configured to be if a 26 KHz crystal were used with the MSP432?

Problem 5 – Setting Clock Rates

Consider that you have a MSP432 with the 32768 crystal installed and the DCO set to run at 12 MHz. Please write the C code which will cause:

c. ACLK to be 4096 Hz
d. MCLK to be 32768 Hz
e. SMCLK to be 2 MHz

POI 6 - UART



Problem 1 – UART Frequency Error

Please describe what what would happen if the baud rate of a UART receiving data were 10% higher, e.g. it's system clock was 10% faster, than the UART that was transmitting the data. Assume that the transmitting UART is running at 9600 baud. Please draw a picture to demonstrate your answer.

Problem 2 – Baud Rate Divisor

What baud rate divisor settings would you use to have the UART running at 9600 with a 16 MHz SMCLK as the clock source?

Problem 3 – UART Setup and Transmission

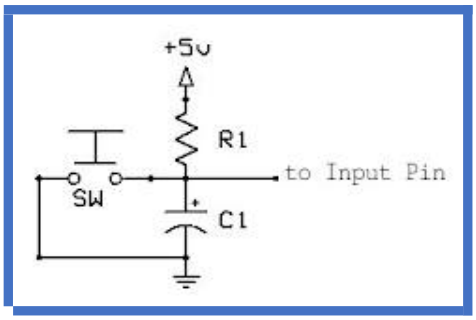
If the MSP432 is setup with SMCLK running at 12 MHz, setup the C code below to setup the UART to operate at 9600 baud with 1 stop bit and even parity. Then transmit the text "Hello"

```
int main(void)
{
      WDTCTL = WDTPW + WDTHOLD; // Stop WDT
      SMCLK_12MHZ();            // SET DCO and SMCLK AT 12 MHZ

}
```

Problem 4 – UART Timing Diagram

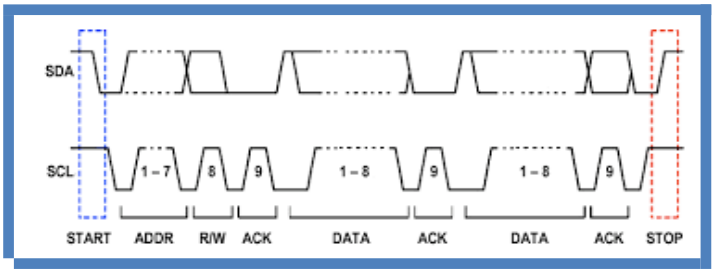
Draw the timing diagram for transmitting an ascii 'a' (0x61) via RS232 with 1 stop bit and odd parity. Include a scale for time and voltage.

POI 7 - ADC 14



ADC 14 problems to be provided

POI 8 - I2C



Problem 1 - I2C EEPROM Interfacing and Analysis

a.) Please draw a *detailed* schematic diagram (pin numbers, signal names, etc.) showing how the MSP 430 would be interfaced to the Microchip 24xx256 serial EEPROM as used in Assignment 8.

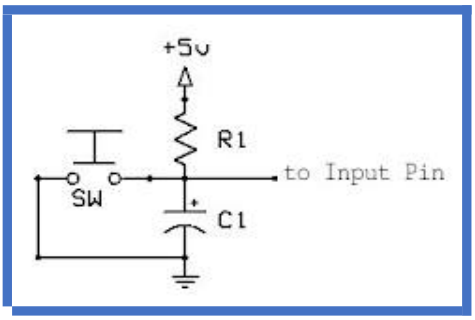
b.) Please annotate and complete the figure below showing the bits (and how they are identified) that would appear on the I2C serial data line when the byte 0xAC were to be written to address $20{,}000_{10}$ in the EEPROM. Please note that the diagram below does not show the Start or Stop conditions.

1							9								18								27								36

c.) Suppose you needed to have two of these EEPROMs interfaced to your MSP430. How would you modify your circuit to accommodate this extra EEPROM.

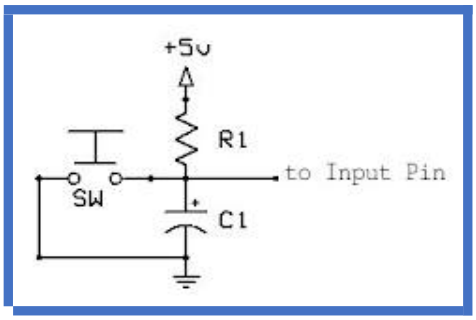
d.) How would your byte read timing diagram from part b.) change if the EEPROM were absent from the system?

POI 9 - Pulse Width Modulation



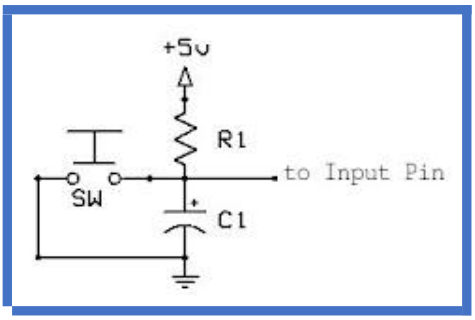
PWM problems to be provided

POI 10 - Real Time Clock



Real-Time Clock problems to be provided

POI 11 - General Support Questions



Problem 1 – UART Baud Rate

What is the slowest possible UART baud rate if SMCLK were set to 16 MHZ for the MSP432?

Problem 2 – I2C vs. SPI

Please compare and contrast I2C and SPI?

Problem 3 – I2C vs. UART

Please compare and contrast I2C and UART?

Problem 4 – Multiple Uses for MSP Pins

Why are pins used for multiple purposes on the MSP432?

Problem 5 – SPI Baud Rates

Given that the DCO is set to 16 MHz, please write C code which will set the baud rate of the SPI interface to be 80 KHz?

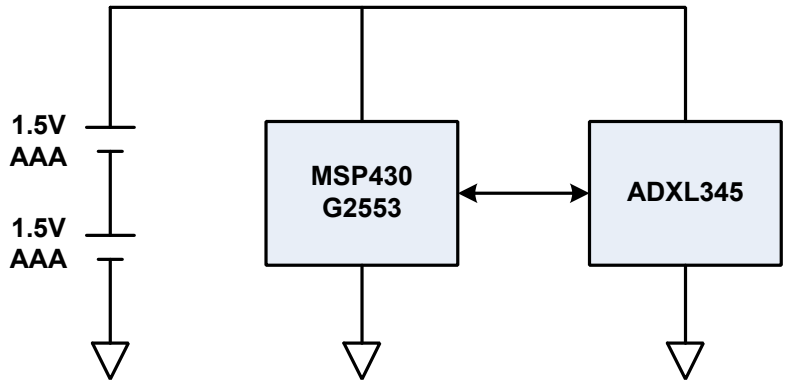
Problem 6 – UART Baud Rates

What is the slowest possible UART baud rate if SMCLK were set to 16 MHZ for the MSP432?

POI 12 - Power and Autonomous Operation

Problem 1 – MSP432 Running with Accelerometer.

Consider that you have an MSP432 and an Analog Devices ADXL 345 accelerometer running off of two AAA batteries. Please see attached data sheets. For every second, the MSP432 runs for 50 ms at 3MHZ and then sleeps for 950 ms in LPM3 running off ACLK with the 32,768 external crystal. The ADXL 345 runs with a 100 HZ data rate for the 50 ms the MSP 430 is running and is in standby mode for the 950 ms that the MSP is asleep. Please answer the questions below.



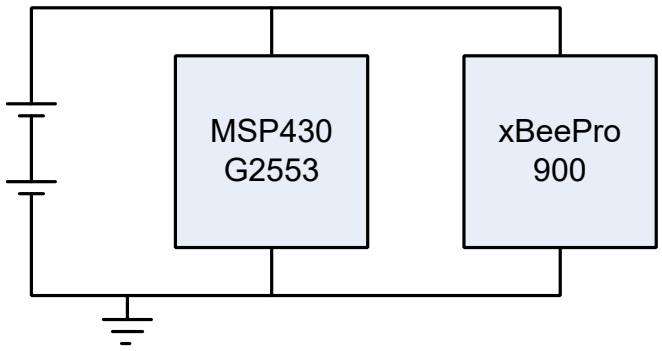
a.) What is the average current draw of the MSP430?

b.) What is the average current draw of the ADXL345?

c.) What is the autonomy in days of this system?

Problem 2 - MSP432 with XBee Pro.

Consider that you have an MSP 432 and a XBee Pro radio running off of two D-cell batteries. Please see attached data sheets. For every second, the MSP430 transmits data for 100 ms, receives data for 100 ms, then sleeps for 800 ms. This pattern is repeated every second. Consider that the MSP runs at 16 MHz when both transmitting and receiving and then sleeps in LPM3 with ACLK running. Please answer the questions below.



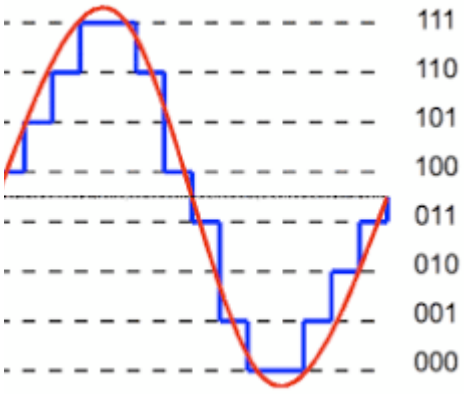
d.) What is the average current draw of the MSP432?

e.) What is the average current draw of the xBee?

f.) What amp-hours of storage are available from the batteries for the load presented by the MSP430?

g.) What is the autonomy in seconds of this system?

POI 13 – ADC Design and Development



Problem 1 - ADC Development.

a. The oscilloscope traces below show the inverting and non-inverting inputs of the op-amp used in Assignment #5, ADC development. What happens in the ADC during the time when the lower trace is a logic high?

b. What determines how wide this pulse (the lower trace) needs to be? For example, you might want to make the pulse 10% as wide as it is. What might keep you from doing this?

c. Please describe what is going on with the upper trace in the oscilloscope screen shot. What is it for/what does it do?

d. Suppose you wanted the ADC to linearly sample a signal which ranged from 0 to 3 volts. What component in the ADC as used in lab could prevent your system from accurately (e.g. linearly) sampling any voltage in this 0V to 3V range? *Please note that this question regards to accurately representing the voltage of the signal and is not related to speed or max sample rate of the ADC.*