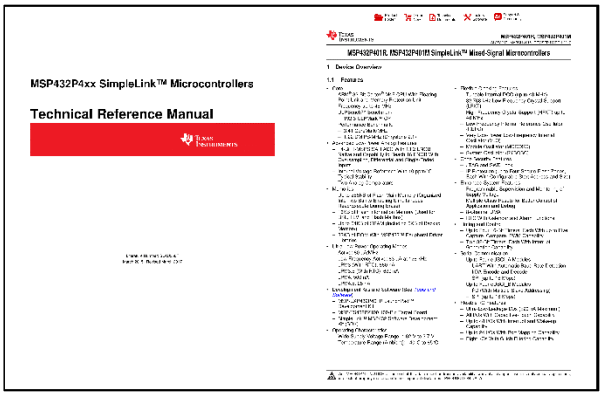
Microprocessors Laboratory Manual
Texas Instruments MSP432

Paul Hummel and Jeff Gerfen

Table of Contents

Lab Manual Resources



Texas Instruments Reference Documentation:

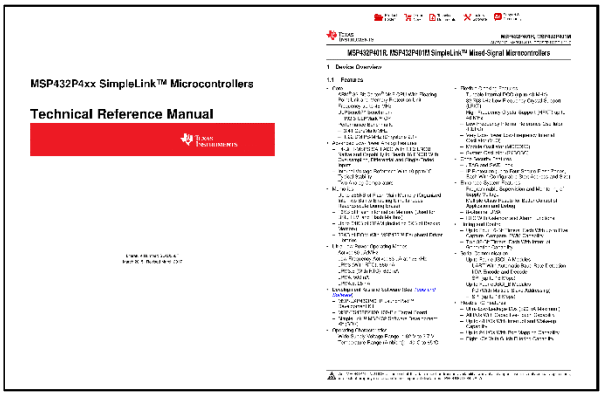
1. MSP-EXP432P401R LaunchPad Quick Start Guide (QSD) http://www.ti.com/lit/pdf/SLAU596
2. MSP432Pxx Technical Reference Manual (TRM) http://www.ti.com/lit/pdf/SLAU356
3. MSP432P401 Datasheet (MDS) http://www.ti.com/lit/pdf/SLAS826
4. MSP432 Launchpad User's Guide (LUG) http://www.ti.com/lit/pdf/SLAU597
5. Code Composer Studio User's Guide (CUG) http://www.ti.com/lit/pdf/SLAU575
6. MSP432 Texas Instruments Reference Code http://www.ti.com/lit/zip/SLAC698

Lab Kit Part List:

1. MSP-EXP432P401R LaunchPad
2. Parallel LCD 2x16 (Recommend NHD-0216HZ-FSW-FBW-33V3C)
3. 12 button matrix keypad (Recommend COM-08653)
4. 12-bit SPI DAC (Recommend MCP4921)
5. 256k EEPROM I2C (Recommend 24LC256)
6. Positional Rotation Servo (Recommend SG90 micro servo)

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). 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 book.* Notice the TRM specifies details for an entire family of devices (MSP432P4) while the MDS is specific to only 2 chips, MSP432P401R and MSP432P401M. The TRM will often times list a range of possible values because different devices in the MSP432P4 family are made with a range of specifications. Looking up details on how a peripheral to the MSP432 functions, like TimerA, would be found in the TRM. Details on what pins are connected to TimerA or how many TimerA peripherals are available would be specific to the chip and found in the MDS.

Instructions

1. Gather the MDS and TRM for the MSP432 from Texas Instruments. (While links are provided for each of these documents in Lab Manual Resources section, it would be worthwhile trying to find these documents directly from Texas Instruments to learn how to find datasheets in the 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

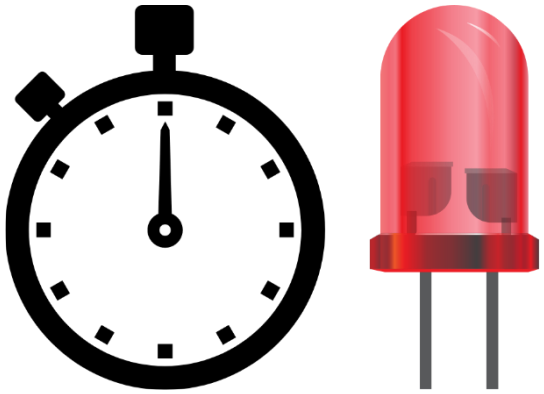
1. Create a memory map for Code and Peripheral address spaces of the MSP432P401R.
2. How many internal oscillators does the MSP432 have?
3. How many timers does the MSP432P401R have? What size are the timers?
4. What is the maximum sampling rate of the analog to digital converter on the MSP432P401R?

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



Reference Material

- MSP432 Technical Reference Manual (TRM) – Digital I/O, Clock System
- LaunchPad User's Guide (LUG) – Schematics
- MSP432P401 Datasheet (MDS) – Port / Pin Function Tables

GPIO and 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 a combination of the colors (red, green, blue) can be created. Please find LED1 and LED2 in the schematics of the LUG and then trace the connections back to the MSP432 to find the pin values for controlling the LEDs. These LEDs can be controlled as GPIO outputs. Read the TRM chapter on Digital I/O to learn which registers control the GPIO ports configuration and behavior. Verify you are able to control the LEDs by writing a short program that is able to turn LED1 on and off or change the color of LED2.

Clock System

The MSP432 has a robust and configurable clock system to allow the ARM Cortex M4F and peripherals to run at a range of frequencies independent of each other. The ARM CPU timing is controlled by the master clock signal (MCLK) while the peripherals use one of the other clock signals, typically AUX, SMCLK, or HSMCLK. Each of these clock signals can use different clock sources and be adjusted independently. Beyond using external oscillators as clock sources, the MSP432 contains a digitally controlled oscillator (DCO) which is software programmable to run at frequencies between 1 and 48 MHz. Read the section on the DCO and familiarize yourself with the DCO's main control register CSCTL0. Technical Note 3 in this book introduces the MSP432 clock system and provides some example code on how to adjust the clock frequencies.

Software Delays

Controlling devices connected to a microprocessor often require specific timing for the control signals. For some devices the control timing can be easily achieved with software delays. In general software delays are not ideal for precise or efficient timing because the CPU is kept busy doing nothing and cannot perform other operations while waiting. In some instances like device initialization or printing to a display this is less of an issue because there is nothing else the CPU could do until the necessary time has passed. Future assignments will cover better timing options available to the MSP432. Software delays are created by having the CPU perform an operation a set number of times. By controlling how many times this instruction occurs, the wait

time can be set. This is usually achieved with empty for loops. The CPU will cycle through the for loop, doing nothing. Adjusting the number of iterations of the loop allows the delay timing to be controlled.

Instructions:

Create a function to change the MSP432 MCLK using the DCO

1. Create a function to use the DCO as the source for MCLK and set the DCO frequency on the MSP432. The function should be able to set the MSP432 to run at any of the nominal DCO frequencies from 1.5 to 48 MHz. The lone parameter for the function will determine which nominal frequency to set. Create #defines as appropriate for representing the various system frequencies to make your code easier to read. For example, your function calls should look something like this
   `set_DCO(FREQ_12_MHz)`

2. Verify your function works using an oscilloscope. The MCLK signal can be brought out on P4.3 using the appropriate SEL0/1 values. Refer to the MDS P4 pin function table to know how to properly configure the pin to output the MCLK signal.

Create Delay Function

3. Create a function **`delay_us()`** to cause a software delay of a specified time. This functions should take in the number of microseconds as an integer and provide the appropriate delay up to 50 ms.

   The delay function will be dependent on the CPU frequency (MCLK). For simplicity your function can assume the MCLK is being sourced by the DCO. The current configuration for the DCO can be determined by reading CSCTL0.

   Not all integer values passed to the function may be achievable at all operating frequencies. For example, when operating at 1.5 MHz, adding a single iteration to a for loop may add 7 us delay. If the the delay function is able to achieve a 10 us delay, the next shortest delay would be 17 us. If a value of 15 was passed to the function, this exact time is not achievable. In this case, the delay function should run for the shortest time that is at least as long as the specified time. In this case, the function should delay for 17 us. (*These numbers are completely fabricated and should not be used for the basis of or as comparison to your delay function*).

   There is an inherit delay in calling and returning from a function. This delay should be accounted for in your function timing. This timing is difficult to determine from C code, but it can be measured with an oscilloscope and toggling an LED.

   Calculating timing from exact operation cycles is difficult without looking at the actual assembly code from the compiler. It is far easier, and faster, to iteratively determine the timing and calibrate the for loop iteration values.

   Avoid non-integer math. Loops cannot iterate for 0.3 cycles, so any decimal math is ultimately pointless (pun intended). There is also a time penalty as you will see in later assignments for using non-integer variables.

   Start by creating a delay function for a single frequency like 1.5 MHz. Calibrate your delay loops and verify the function works for the necessary time range. Then expand the function to work for the next frequency.

Validate Delay Accuracy

4. Verify your delay function timing by turning an LED for 40 ms and viewing the pulse on an oscilloscope for measurement.

5. Change DCO frequencies and verify the delay function timing is still accurate. The only change to your code should be the parameter you pass to the **`set_DCO()`** function.

6. Repeat steps 4 and 5 with a 40 us delay.

Generate and Observe Short Pulses

7. Use an oscilloscope to determine the shortest timing delay possible with your function for each DCO frequency.

Building a Header File and Associated C file

8. Place the code that implements **`delay_us()`** and **`set_DCO()`** functions in a separate file such as **`delay.c`** with an accompanying **`delay.h`**. This will allow these functions to be easy included in later projects in the course. Technical Note 9 on 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. Scope captures of 40 ms pulse at each DCO frequency 1.5MHz, 6 MHz, 24 MHz, and 48 MHz

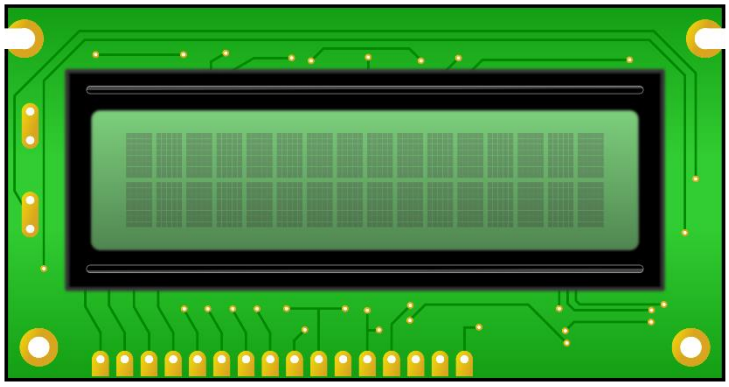
    b. Scope captures of 40 us pulse at each DCO frequency 3 MHz, 12 MHz, and 48 MHz

    c. Scope captures of the shortest pulse generated at each DCO frequency 1.5 MHz, 3 MHz, 24 MHz, and 48 MHz.

    d. Project code properly formatted*

**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 the reader in quickly interpreting what they are looking at, etc. For printing purposes, a syntax highlighting site such as https://emn178.github.io/online-tools/syntax_highlight.html can be helpful. (Style Xcode is recommended)*

A3 – LCD Display



Reference Material

- MSP432 Technical Reference Manual (TRM) – Digital I/O
- LCD / LCD Controller Datasheets

Instructions

1. Gather and review the data sheet for the LCD display. Be sure to look over the schematic, instruction table, initialization sequence, and write data timing diagram.

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. When wiring devices to the MSP432 it is important to check all signals connected to the MSP432 ports. If any voltage above 4.7 V is applied to any port, it is likely to damage the port and possibly the MSP432 in general.

Code Design

4. Write a short program to initialize the LCD and write a single character to the display.

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 to 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.

7. Write a program using your library functions to print “Hello World” on the top line and “Assignment 3” on the bottom line of the LCD.

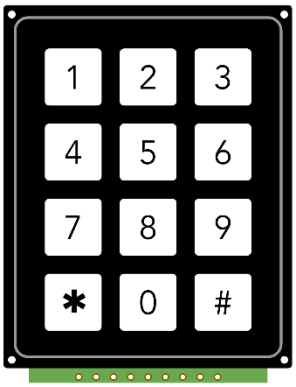
Questions

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 assuming the LCD is connected in nibble mode.

Deliverables:

1. Create a simple pdf document (not a full lab report) containing the following:
    a. Questions written out with the answer
    b. Link to your video demonstration
    c. Source code for your program (properly formatted and commented)

A4 – Keypad



Reference Material

- MSP432 Technical Reference Manual (TRM) – Digital I/O
- MSP-EXP432P401R LaunchPad Quick Start Guide (QSD)
- LCD / LCD Controller Datasheets
- Keypad Datasheet

Matrix Keypad Construction

The keypad is a completely passive device made up of 12 buttons. To reduce the number of wires and connections, the buttons are connected in a matrix of rows and columns as shown in Figure 1 below. Pressing a button on the keypad does not make a connection to a high or low voltage. Instead it makes a connection between a row and column wire. Determining which button is pressed requires determining which row and column is connected. This requires controlling one dimension, either the columns or the rows as outputs while the other is an input.

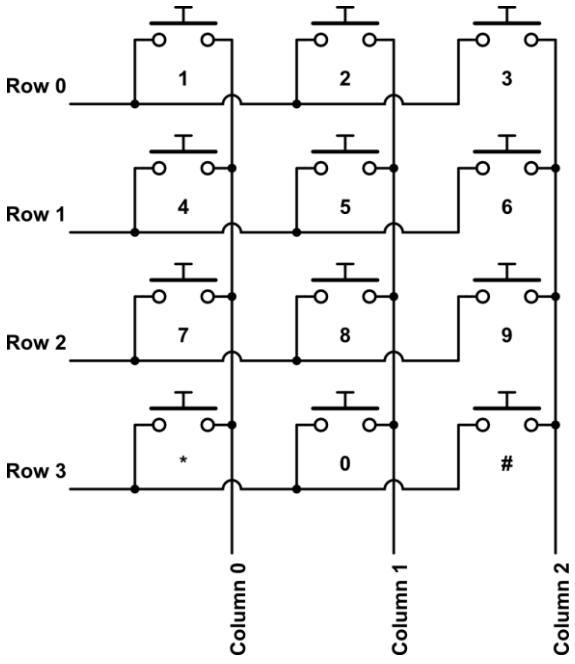


Figure 1: Keypad Schematic

Detecting Key Presses

Determining the button press on a keypad can be performed in 2 stages. The first stage is detecting when a key is pressed while the second is determining which key is pressed. First detecting a keypress can be done by setting all of the columns (or rows) high and monitoring the rows (or columns). By setting all of one dimension of the keypad matrix high (columns or rows), any key press will send one of the wires of the other dimension (rows or columns) high when the button press makes a connection. For example, if Column0 – Column2 is set high, pressing button 8 will cause Row 2 to go high because it will make a connection between Column 1 and Row 2. Notice detecting a button is being pressed does not mean it is possible to determine which key was pressed. Pressing button 7 or 9 will also cause Row 2 to go high the same way as button 8. This requires the second stage of the process to determine specifically which button was pressed. (*Hint: This could be utilized to make the keypad interrupt driven*)

In the above example, the difference between buttons 7, 8, and 9 can be made only be determining which column is being connected to Row 2 by the button press. This determination can be made by selectively cycling which columns are set high and reading the rows. If button 8 is pressed, setting Column 0 high while setting the others low will not result in Row 2 being high. This eliminates button 7 as a possibility. Only by setting Column 1 high will Row 2 go high signifying a connection at button 8. Once a button press has been detected in stage 1, the columns (or rows) can be individually cycled while reading the other.

Pull-up / Pull-down Resistors

When no key is pressed on the keypad, the connections between the rows and columns are open. When his occurs, the input signals being read from the rows (or columns) will not be connected to a low or high voltage. This means the input signal will float. Floating signals can create havoc on digital circuits because when the inputs on a digital signal float they may be read as low (0) or high (1). This will create unpredictable behavior in reading keypresses when no keys are pressed. To keep the inputs from floating, the input signals can be forced to go high or low rather than float. This is accomplished by adding resistors that pull the signal up or pull the signal down. These resistors are called pull up or pull down resistors. When using pull up resistors, the button connection would need to drive the input low. When pull down resistors are used, the button connection would need to drive the input high. In the example behavior described above, the button connections would drive the input high, so pull down resistors will be needed. Built in input resistors are available on all of the GPIO pins of the MSP432 and can be configured as either pull-up or pull-down.

Instructions

1. Create a schematic to connect the keypad and LCD on the MSP432 LaunchPad. Note that some pin assignments may need to be changed from your previous LCD assignment. 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 on the Quick Start Guide. Most ports do not have a continuous 8 pins, so you should consider how you will need to

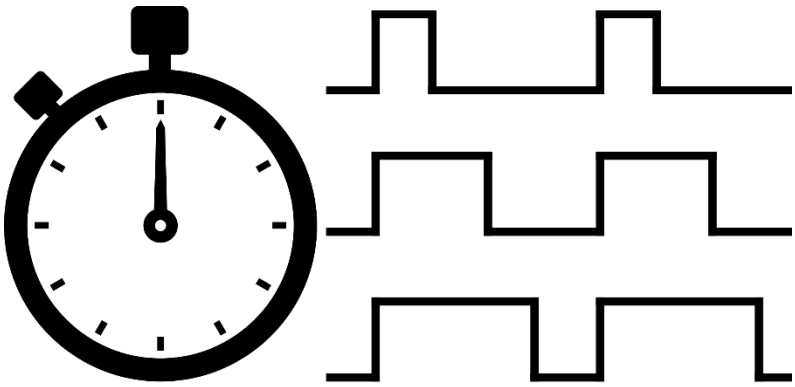
write your program interfacing with each peripheral before deciding how to connect them to the MSP432.

2. Write a keypad function that can detect and determine keypresses
3. Reuse your previous LCD functions to display the key whenever a button press is detected.
4. 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. Link to a video demonstration of the keypad and LCD operating together.
   b. Schematic diagram of the system (Internal pull-up / pull-down resistors should not be included).
   c. Source code for your program (properly formatted and commented)

A5 – Interrupts and Timers



Reference Material

- MSP432 Technical Reference Manual (TRM) – Digital I/O, TimerA
- MSP432P401 Datasheet (MDS) – Interrupts, pin function tables

Timers

Modern MCUs include hardware timers to keep precise timing without the need of software delay loops. This allows the CPU to process other instructions while "waiting" for the specified time. The MSP432 has several different timers, SysTick, Timer32, and TimerA. For this assignment, the most basic functionality of TimerA will be used. The core of a timer is a hardware counter. Timers count clock ticks and can cause interrupts when the count reaches a specified value. This allows easy and precise timing with an MCU without software delay loops.

Interrupts

An interrupt is a hardware mechanism in an MCU that can cause the CPU to stop executing the current instructions and jump to a specific subroutine. Each interrupt can have its own subroutine to allow each interrupt handle a specific task. Each TimerA in the MSP432 has 2 interrupt subroutines (ISR).

Instructions:

Part A: 25 kHz with 25% Duty Cycle Square Wave

1. Calculate the necessary CCR values to create a 25 kHz square wave with a 25% duty cycle using a 24 MHz input clock. Be sure to specify period, high, and low time values.
2. Reuse your previous DCO code to set MCLK and SMCLK to 24 MHz. Using SMCLK for TimerA, create a 25 kHz with a 25% duty cycle square wave by setting a pin high and low in an ISR. Your code for main should only setup TimerA and not keep track of any timing. No software delays should be used.
3. View this output on an oscilloscope and take a screen capture.

Part B: ISR Processing Measurement

4. Change your code above to create a single ISR which generates a 25 kHz square wave with a 50% duty cycle using only CCR0.
5. Add a second GPIO output to your system to measure ISR processing time. This can be done by driving the bit high when first entering the ISR and driven low before exiting.

6. Bring MCLK out of the MSP432 for viewing on the oscilloscope. The MCLK signal can be brought out on P4.3 using the appropriate SEL0/1 values. Refer to the MDS P4 pin function table to know how to properly configure the pin to output the MCLK signal.
7. Watch this ISR execution timing bit on the oscilloscope while watching MCLK on another trace. Count how many MCLK cycles it takes to execute your ISR. Take a scope capture from the oscilloscope.

Part C: Shortest Pulse and Failing ISR

8. Using the same code from Part B, lower the value of CCR0 until the timer fails to generate a pulse width that is proportional to the CCR0 value. Record the smallest value of CCR0 that works and take a scope capture of the resulting 50% duty cycle square wave along with the ISR timing bit output. Does this CCR0 value correlate to the amount of clocks that the ISR requires to execute from Part B?

Part D: 50% Duty Cycle and 20-Second Period Square Wave

9. Reuse your previous DCO code to set MCLK and SMCLK to 1.5 MHz. Using SMCLK for TimerA, create a 50% duty cycle square wave with a 20 second period to toggle an LED. (Hint: TimerA will not be able to count high enough to time 10s, so you will have to count multiple timer events to reach 10s) Confirm your timer operation with a stopwatch, and make a YouTube demo which shows the stopwatch display with the toggling LED.

Part E: 2-Bit Counter

10. Keeping MCLK and SMCLK at 1.5 MHz, use TimerA with CCR0 and CCR1 to create 2 synchronized square waves of different frequencies. CCR0 will create a frequency of 1 kHz while CCR1 will create a frequency of 500 Hz. Each ISR will toggle a different pin. By considering each bit as part of a 2 bit counter, with CCR0 defining the LSB and CCR1 defining the MSB, the result should appear similarly to Figure 1.

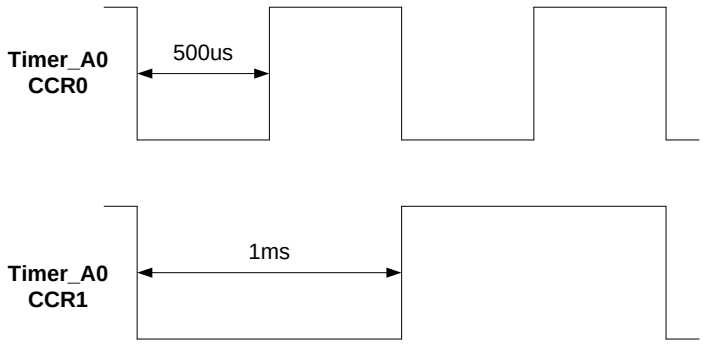


Figure 1: 2-bit Counter

11. View both outputs with an oscilloscope. Take a scope capture which shows both bits to see a complete count from 00 to 11.
12. 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?

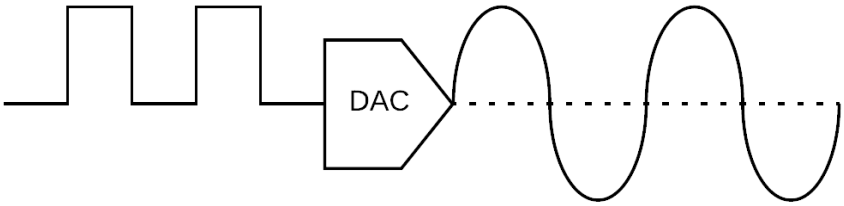
Extra Credit Reflex Game

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 have the other person press button #2 as quickly as possible to test his or her reflexes. This game should automatically repeat after each play of the game. Demonstrate ***Reflex*** with a YouTube video.

Deliverables

1. Create a simple pdf document containing the following:
    a. CCR value calculations from Part A
    b. Screen captures:
        i. Part A – 25 kHz with 25% duty cycle square wave
        ii. Part B – ISR execution timing with MCLK
        iii. Part C – Smallest CCR0 value output
        iv. Part E – Screen captures of 2-bit counter
2. Link to video demonstration of 10s LED toggle
3. Properly formatted code of:
    a. Part A – 25 kHz, 25 % duty cycle square wave
    b. Part D – 10s LED toggle
    c. Part E – 2-bit counter
4. Observations and answers to questions
    a. Part C – Shortest CCR0 pulse
    b. Part E – 2-bit counter
5. Only if doing the extra credit
    a. Link to video demonstration of Reflex Game
    b. Source code for Reflex Game properly formatted

A6 - SPI DAC and Waveform Generation



Reference Material

- MSP432 Technical Reference Manual (TRM) – SPI, TimerA
- Microchip MCP4921 Datasheet
- MSP-EXP432P401R LaunchPad Quick Start Guide (QSD)

SPI

Serial Peripheral Interface (SPI) is a common full-duplex, 4-wire bus used by a variety of digital devices. The MSP432 provides multiple SPI outputs via the eUSCI peripherals. Configuration options can be reviewed in corresponding TRM chapter.

DAC

The MSP432 does not provide any built in analog outputs, so an external device must be used. For this assignment you will use a Microchip DAC (digital to analog convertor). The MCP4921 is a 12-bit DAC, allowing for 2^{12} different output voltages, including 0v. The MCP4921 interfaces with an MCU via SPI. The DAC is a write only device so only 3 of the 4-wires from SPI are necessary. Details on how the MCP4921 is controlled can be found in its datasheet. Figure 1 below shows a sample timing diagram for setting the DAC to output a voltage corresponding to the 12-bit value 0x6A7, unbuffered, and with a gain of 1.

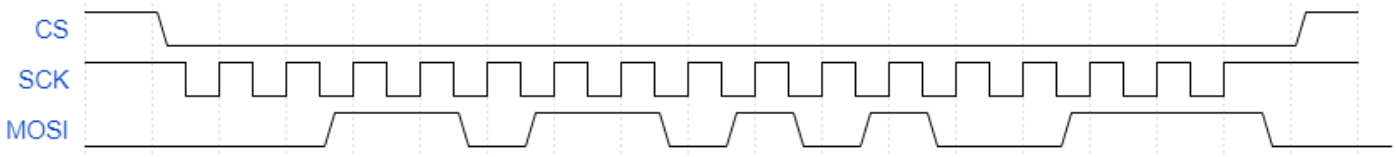


Figure1: SPI Timing Diagram for Unbuffered, 1x Gain, 0x6A7

Instructions:

<u>Interface MSP432 and MCP4921</u>

1. Connect the MCP4921 DAC to the MSP432 via SPI. The MSP432P401R Quick start guide (QSD) or Datasheet (MSD) can be used to find which pins are selectable for eUSCI operations.

2. Use 3.3V for power and references for the MCP4921. No 5V signals are necessary and should be avoided to prevent any unintended overvoltage on the MSP432 pins.

3. Write a small program to control the DAC to output a single DC voltage. Measure the output voltage with a multimeter and compare to the specified voltage. Use this measurement to calibrate your DAC function.

4. Split the working program into 2 functions, one to initialize the SPI bus, and the other to set an output voltage from the DAC.

Square and Triangle Wave Generation

5. Using your working functions, write a program to create a 2Vpp (peak-peak) square wave with a 1V DC offset and a period of 20 ms. (Hint: TimerA is useful for precise timing). View this signal on an oscilloscope and save a screen capture.

6. Write a program to create a 2Vpp triangle wave with a 1V DC offset and a period of 20 ms using timers. Save a screen capture of this signal.

Deliverables

1. Create a simple pdf document containing the following:
    a. Scope captures of square wave
    b. Scope captures of triangle wave
2. Project code properly formatted

A7 – Execution Timing

Reference Material

- MSP432 Technical Reference Manual (TRM) – Digital I/O

Instructions

1. Set up an oscilloscope to catch a single pulse to collect accurate timing.
2. 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.
3. Notice P2.0 is toggled before and after `TestFunction` subroutine and P1.0 is toggled before and after the arithmetic function.
4. Change var_type and Adjust `TestFunction(num)` to test the various variable types and operations
5. Fill in the table with the timing results obtained.
   To time how long it takes to enter the subroutine (Subroutine call), look at the difference between P2.0 going high and P1.0 going high. You do not need to calculate this time for each arithmetic function, use the simple testVar = num for filling in the table. To time the arithmetic function, use the width of P1.0.

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)					

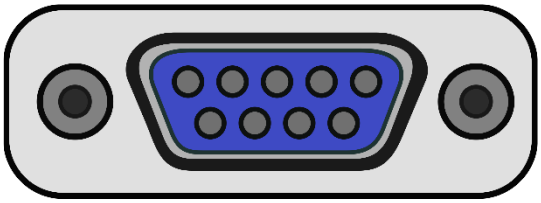
Deliverables:

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 &= ~BIT0;    //set P2.0 as simple I/O
    P2->SEL0 &= ~BIT0;
    P2->DIR |= BIT0;      //set P2.0 as output pins

    P2->OUT |= BIT0;      // turn on Blue LED

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

    P2->OUT &= ~BIT0; // turn off Blue 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;      // turn RED LED on

    { insert_function_here (ie testVar = num;) }

    P1->OUT &= ~BIT0;     // turn RED LED off
    return testVar;
}
```

A8 – UART Communications



Reference Material

- MSP432 Technical Reference Manual (TRM) – UART
- Technical Note – UART Communication

Instructions

UART Baud Rate Calculation

1. Calculate the baud rate and fractional modulation divisors for communicating via UART at 115.2 kHz with the MSP432 running at its default speed of 3 MHz.

Transmit Characters

2. Use the UART mode of USCI_A0 to get the MSP432 communicating with a terminal on the computer via RS-232. The MSP432 has an RS-232 to USB built into the hardware debugger that is connected through the same cable used to program the board.

   Write a simple program that will repeatedly send 1 character via UART and verify it is communicating properly with a terminal program on the computer. Note that terminal programs print the data received as ASCII values.

Receive Characters

3. Change your program above to receive a character from the terminal program. *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, the program will need to echo the character it received back to the terminal.*

Use the Terminal to Control the DAC

4. If the receive character program is not already an interrupt service routine for the UART, reimplement it as an ISR. When a character is received from the terminal via UART, the ISR will read the transmitted character and parse it to create a numerical value (inValue). *Note: This step requires the ISR take previous characters received into account as each character is received.* Some error checking needs to be done as part of the parsing to ignore characters other than numbers (0 - 9) and return (enter key). *Note: an ASCII table can be useful for this step.* 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 number (inValue) will continue to be concatenated until a return character is received. The return key will signify the number (inValue) has been entered completely. When a return is received the ISR should set a flag (global variable) for the main program. 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 (inValue) to the DAC with the previous written DAC function. The program needs to do error checking on inValue before sending it to the DAC. Only values from 0 to 4095 should be transmitted.

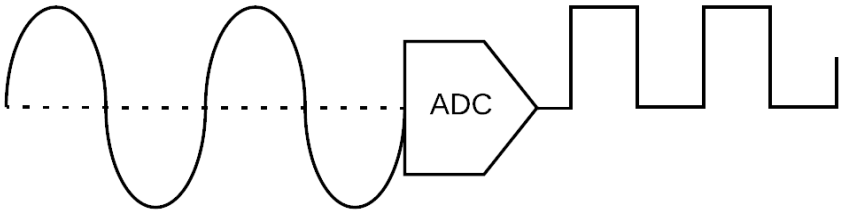
Bonus (10%):

5. Encapsulate the UART routines to eliminate all global variables. Instead of checking a flag variable directly in the main program, the flag status will be checked by calling a function. The main program will still call the DAC function, but will use a function to get the numerical value (inValue) that is to be passed to the DAC function. All UART functions will all go in a separate .c file and use variables with limited scope to protect them. The Technical Note on ISR communications in this book shows how this can be done.

Deliverables:

1. Single PDF document with:
    a. Baud rate calculations
    b. 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.
    c. Project code properly formatted

A9 – Analog to Digital Conversion



Reference Material

- MSP432 Technical Reference Manual (TRM) – ADC14
- Technical Note – Hardware Debugger

Connecting to the ADC

1. Connect an adjustable DC voltage source to an analog input pin on the MSP432.

   BEWARE OF VOLTAGES ABOVE 4.7 V!

   Always be sure to double check the voltage before connecting or powering any analog voltage connected to the MSP432. When connecting the external source, make sure to connect the ground of the source to the MSP432 ground. Possible sources include power supplies, function generators, or wavegen outputs from the oscilloscope. Any source that you can reliably adjust a DC voltage from 0 to 3.3V will work. **To be careful, never set the voltage above 3.3V.**

Instructions

2. Write a program that will take a single sample from the ADC14 using the 3.3 V reference. Configure the ADC to perform a 14-bit conversion with a sample time of 4 clock cycles.
3. 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.
4. Your main program should run in an infinite loop checking for the global flag. When the flag is set, it should reset the flag and initiate a new sample for the ADC14.
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.
6. With the DC voltage source set to 1 V, record the variation in ADC values from the ADC14.
7. Adjust the sample time to 16, 96, and 192 clock cycles, recording the variation in the ADC values for each.

Calibrate your ADC14

8. Select the minimal sample time that provides the required accuracy of +/- 0.01 V.

9. 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 for voltages 0.00 – 3.30 V. Approaches on how to derive a calibration equation can be found in the Technical Note on Calibrating ADC / Sensor. *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

10. Print the calibrated voltage value to a terminal via 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 terminal 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%)

11. Encapsulate the ADC14 ISR routines to eliminate all global variables. Instead of checking a flag variable directly in the main program, the flag status will be checked by calling a function. The main program can still call a UART transmit function, but will use a function to get the ADC value. All ADC14 functions will go in a separate .c file and use variables with limited scope to protect them. The Technical Note on ISR communications details how this can be done

Deliverables:

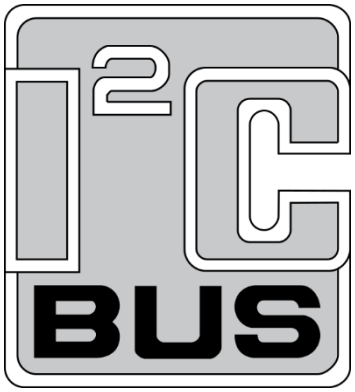
1. A single pdf document with:

    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. Table of ADC14 sample time clocks and corresponding variation of output values.

    c. Source code for your program (properly formatted and commented)

A10 – I²C EEPROM



Reference Material

- MSP432 Technical Reference Manual (TRM) – eUSCI I2C
- Microchip 24LC256 EEPROM Datasheet

Instructions

Organize Hardware / Software

1. Draw your own schematic of the MSP432 interfaced to the Microchip 24LC256 I2C EEPROM and wire up the circuit on the breadboard.
2. Review the C code provided below 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

3. 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.
4. 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^2C Hardware Tests

5. Remove the pull-up resistor on each of the SCL and SDA lines separately and observe what happens when the program is run. What interrupt flag bit is being set and not handled in each of these scenarios?

6. Remove the wire to the SDA input on the slave (EEPROM), being sure to keep the pull-up resistor connected to the MSP432. Rerun the program watching the MSP432's SDA and SCL lines with the oscilloscope. Observe what happens. Is the first ACK bit affected? What about the rest of the bits? What interrupt flag bit is being set and not handled?

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 from I^2C Bus Analysis (steps 3 and 4)
    c. Screenshots and answers to the questions from the I^2C Hardware Tests (steps 5 and 6)
    d. Flow chart of the provided code

Source Code Listing

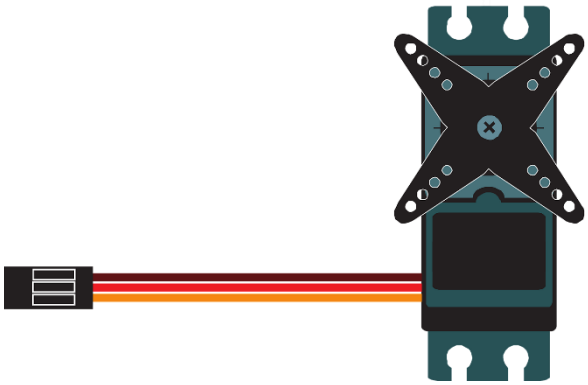
```
//******************************************************************************
//  CPE 329 - Assignment 9
//
//  Description: This demo connects an MSP432 to a Microchip 24LC256 EEPROM via
//  the I2C bus. The MSP432 acts as the master and the EEPROM is a slave.
//  The EEPROM uses 3 external connections A2 A1 A0 to set the lower 3 bits of
//  its bus address. This creates a bus address of "1 0 1 0 A2 A1 A0". The code
//  below assumes those three connections are all connected to VSS (Ground) and
//  are logic 0. This gives the EEPROM a bus address of 0x50.
//
//
//                                /|\  /|\
//               MSP432P401      10k  10k     24LC256 EEPROM
//                 master          |    |          Slave
//             -----------------   |    |   -----------------
//            |     P1.6/UCB0SDA|<-|----|->|SDA (5)          |
//            |                 |  |       |                 |
//            |                 |  |       |                 |
//            |     P1.7/UCB0SCL|<-|------>|SCL (6)          |
//            |                 |          |                 |
//
//   Paul Hummel
//   Cal Poly
//   May 2017 (created)
//   Built with CCSv8.1
//******************************************************************************
#include "msp.h"
#include <stdint.h>

#define EEPROM_ADDRESS 0x50

void InitEEPROM(uint8_t DeviceAddress);
void WriteEEPROM(uint16_t MemAddress, uint8_t MemByte);
uint8_t ReadEEPROM(uint16_t MemAddress);

uint16_t TransmitFlag = 0;

int main(void)
{
    uint32_t i;
    uint8_t value;

    WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;       // Stop watchdog timer

    P2->DIR |= BIT2 | BIT1 | BIT0;                    // Configure LED2
    P2->OUT &= ~(BIT2 | BIT1 | BIT0);

    __enable_irq();                             // Enable global interrupt

    InitEEPROM(EEPROM_ADDRESS);

    WriteEEPROM(0x1122, 0x21);

    for (i = 4000; i > 0; i--);    // Delay for EEPROM write cycle (5 ms)

    value = ReadEEPROM(0x1122);     // Read value from EEPROM

    P2->OUT |= (value & (BIT2 | BIT1 | BIT0)); // Set LED2 with 3 LSB of value

    __sleep();      // go to lower power mode
}
```

```
/*
/  Initialize I2C bus for communicating with EEPROM.
*/
void InitEEPROM(uint8_t DeviceAddress)
{

  P1->SEL0 |= BIT6 | BIT7;                // Set I2C pins of eUSCI_B0

  // Enable eUSCIB0 interrupt in NVIC module
  NVIC->ISER[0] = 1 << ((EUSCIB0_IRQn) & 31);

  // Configure USCI_B0 for I2C mode
  EUSCI_B0->CTLW0 |= EUSCI_A_CTLW0_SWRST;   // Software reset enabled
  EUSCI_B0->CTLW0 = EUSCI_A_CTLW0_SWRST |   // Remain eUSCI in reset mode
          EUSCI_B_CTLW0_MODE_3 |            // I2C mode
          EUSCI_B_CTLW0_MST |               // Master mode
          EUSCI_B_CTLW0_SYNC |              // Sync mode
          EUSCI_B_CTLW0_SSEL__SMCLK;        // SMCLK

  EUSCI_B0->BRW = 30;                       // baudrate = SMCLK / 30 = 100kHz
  EUSCI_B0->I2CSA = DeviceAddress;          // Slave address
  EUSCI_B0->CTLW0 &= ~EUSCI_A_CTLW0_SWRST;  // Release eUSCI from reset

  EUSCI_B0->IE |= EUSCI_A_IE_RXIE |         // Enable receive interrupt
                  EUSCI_A_IE_TXIE;
}

/*
/  Function that writes a single byte to the EEPROM.
/
/  MemAddress  - 2 byte address specifies the address in the EEPROM memory
/  MemByte     - 1 byte value that is stored in the EEPROM
/
/  Procedure :
/       start
/       transmit address+W (control+0)     -> ACK (from EEPROM)
/       transmit data      (high address)  -> ACK (from EEPROM)
/       transmit data      (low address)   -> ACK (from EEPROM)
/       transmit data      (data)          -> ACK (from EEPROM)
/       stop
*/
void WriteEEPROM(uint16_t MemAddress, uint8_t MemByte)
{
  uint8_t HiAddress;
  uint8_t LoAddress;

  HiAddress = MemAddress >> 8;
  LoAddress = MemAddress & 0xFF;

  EUSCI_B0->CTLW0 |= EUSCI_B_CTLW0_TR;          // Set transmit mode (write)
  EUSCI_B0->CTLW0 |= EUSCI_B_CTLW0_TXSTT;       // I2C start condition

  while (!TransmitFlag);                // Wait for EEPROM address to transmit
  TransmitFlag = 0;
  EUSCI_B0 -> TXBUF = HiAddress;    // Send the high byte of the memory address

  while (!TransmitFlag);            // Wait for the transmit to complete
  TransmitFlag = 0;
  EUSCI_B0 -> TXBUF = LoAddress;    // Send the high byte of the memory address

  while (!TransmitFlag);            // Wait for the transmit to complete
  TransmitFlag = 0;
  EUSCI_B0 -> TXBUF = MemByte;      // Send the byte to store in EEPROM

  while (!TransmitFlag);            // Wait for the transmit to complete
  TransmitFlag = 0;
  EUSCI_B0 -> CTLW0 |= EUSCI_B_CTLW0_TXSTP;   // I2C stop condition
}
```

```
/*
/  Function that reads a single byte from the EEPROM.
/
/  MemAddress  - 2 byte address specifies the address in the EEPROM memory
/  ReceiveByte - 1 byte value that is received from the EEPROM
/
/  Procedure :
/      start
/      transmit address+W (control+0)    -> ACK (from EEPROM)
/      transmit data      (high address) -> ACK (from EEPROM)
/      transmit data      (low address)  -> ACK (from EEPROM)
/      start
/      transmit address+R (control+1)    -> ACK (from EEPROM)
/      transmit data      (data)         -> NACK (from MSP432)
/      stop
*/
uint8_t ReadEEPROM(uint16_t MemAddress)
{
  uint8_t ReceiveByte;
  uint8_t HiAddress;
  uint8_t LoAddress;

  HiAddress = MemAddress >> 8;
  LoAddress = MemAddress & 0xFF;

  EUSCI_B0->CTLW0 |= EUSCI_B_CTLW0_TR;      // Set transmit mode (write)
  EUSCI_B0->CTLW0 |= EUSCI_B_CTLW0_TXSTT;   // I2C start condition

  while (!TransmitFlag);                // Wait for EEPROM address to transmit
  TransmitFlag = 0;

  EUSCI_B0 -> TXBUF = HiAddress;    // Send the high byte of the memory address

  while (!TransmitFlag);            // Wait for the transmit to complete
  TransmitFlag = 0;

  EUSCI_B0 -> TXBUF = LoAddress;    // Send the low byte of the memory address

  while (!TransmitFlag);            // Wait for the transmit to complete
  TransmitFlag = 0;

  EUSCI_B0->CTLW0 &= ~EUSCI_B_CTLW0_TR;   // Set receive mode (read)
  EUSCI_B0->CTLW0 |= EUSCI_B_CTLW0_TXSTT; // I2C start condition (restart)

  // Wait for start to be transmitted
  while ((EUSCI_B0->CTLW0 & EUSCI_B_CTLW0_TXSTT));

  // set stop bit to trigger after first byte
  EUSCI_B0->CTLW0 |= EUSCI_B_CTLW0_TXSTP;

  while (!TransmitFlag);            // Wait to receive a byte
  TransmitFlag = 0;

  ReceiveByte = EUSCI_B0->RXBUF;    // Read byte from the buffer

  return ReceiveByte;
}
```

```
/*
/  I2C Interrupt Service Routine
*/
void EUSCIB0_IRQHandler(void)
{
    if (EUSCI_B0->IFG & EUSCI_B_IFG_TXIFG0)     // Check if transmit complete
    {
        EUSCI_B0->IFG &= ~ EUSCI_B_IFG_TXIFG0;  // Clear interrupt flag
        TransmitFlag = 1;                       // Set global flag
    }

    if (EUSCI_B0->IFG & EUSCI_B_IFG_RXIFG0)     // Check if receive complete
    {
        EUSCI_B0->IFG &= ~ EUSCI_B_IFG_RXIFG0;  // Clear interrupt flag
        TransmitFlag = 1;                       // Set global flag
    }
}
```

A11 – Pulse Width Modulation and Servos.



Reference Material

- MSP432 Technical Reference Manual (TRM) – TimerA
- Servo Datasheet

Instructions

Servo Calibration

1. Create a pulse signal with a 1.5ms pulse and 20 ms period using a TimerA output.
2. Connect a servo to the MSP432. Some micro servos can be powered from 3.3V while others require 5V. *If using 5V be cautious to avoid connecting the 5V power line to any pins on the MSP432.*
3. Adjust the width of the pulse to determine the full range of the servo, ideally 0° to 180°. *Note: No two servos are identical. Even the same brand and model will require slightly different pulse widths.*
4. Determine a calibration for rotational control of the servo with a defined 0° for the maximum counter clockwise rotation. All defined angular positions will be relative to this position.

Servo Controller

5. Write a program to control the rotational position of the servo with the keypad to meet the device specifications given below.

Device Specifications

1. The device shall control a single standard servo
    - 1.1. The servo shall operate from 0° to 180°
    - 1.2. The servo position shall move in increments of 10°
2. The device shall use a 12-button 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° from 0° to 180°

2.2. The device shall accept # and ∗ input keys

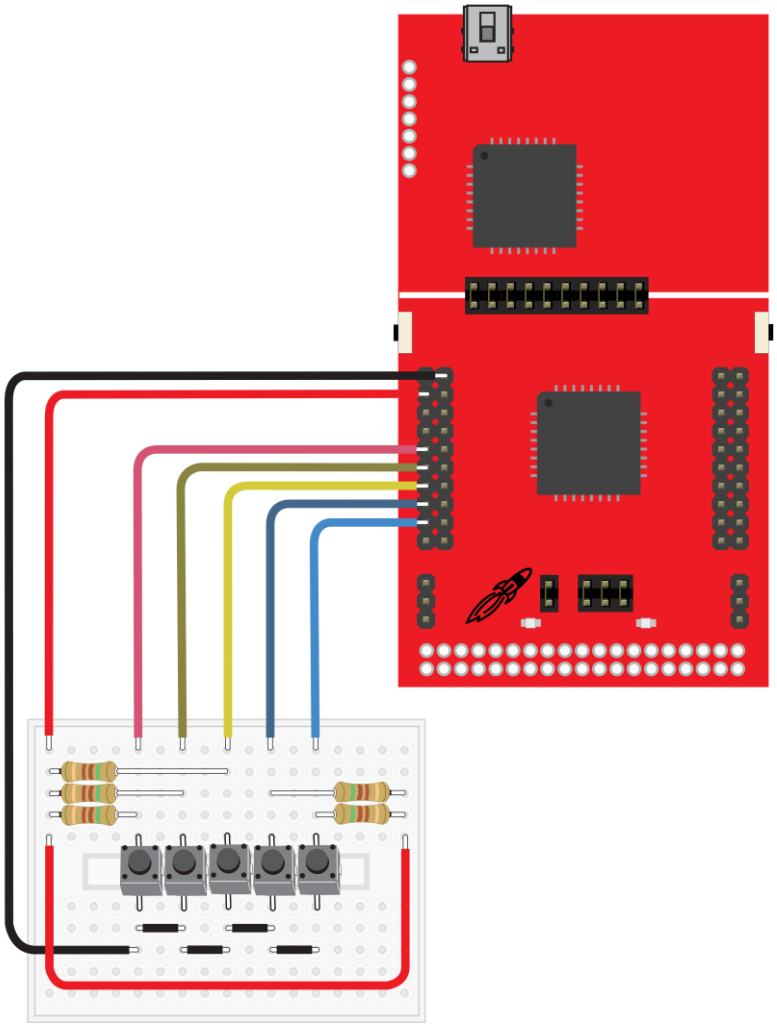
2.2.1. The device shall change the servo position 10° counter clockwise when # is entered

2.2.2. The device shall change the servo position 10° clockwise when ∗ is entered

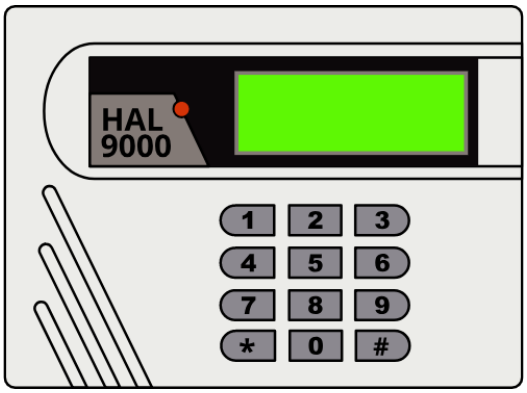
Deliverables

1. A single pdf document with:
    a. Link to a demo video of working device. The video should show the servo rotating to various angles specified by the user using both numerical and incremental (# and ∗) inputs.
    b. Source code for your program (properly formatted and commented)

Projects



P1 – Digital Lock



Reference Material

- MSP432 Technical Reference Manual (TRM) – Digital I/O
- LCD / LCD Controller Datasheets
- MSP-EXP432P401R LaunchPad Quick Start Guide (QSD)

Introduction

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

Objectives

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

System Requirements

- The digital lock will utilize a pin number of at least 4 digits.
- The digital lock device will power up in a locked state and the LCD will display:

```
LOCKED
ENTER KEY
```

- Pressing keys on the keypad will display on the bottom row after KEY
- After a pin value has been entered the display will be cleared, and if the combination was correct, the screen will display UNLOCKED. Otherwise the screen will display the initial LOCKED / ENTER KEY screen and wait for a new pin to be entered.
- The $*$ key will cause the LCD to be cleared, display the LOCKED / ENTER KEY screen, and wait for a new pin combination to be entered.

Suggested Design Steps

1. Draw a single schematic diagram of the keypad and LCD connected to the LaunchPad
2. Confirm that the keypad and LCD each operate correctly. (Reusing your programs from previous assignments can prove useful for this)
3. Implement combination lock functionality

Tips and Suggestions

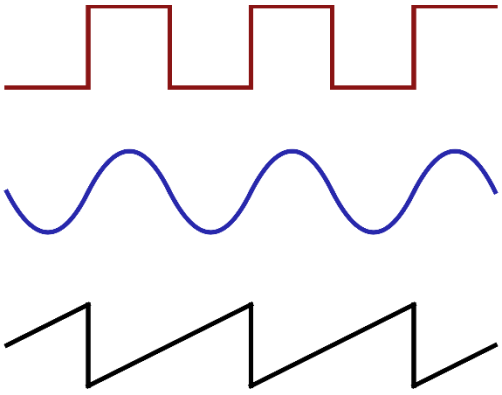
Successful completion of this project requires you to assimilate a variety of information from different sources and implement a system solution. Although this process may seem daunting at the start, a methodical step-by-step approach can be helpful. Here are suggested steps for your project implementation.

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. 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 on the Quick Start Guide. Most ports do not have a continuous 8 pins, so you should consider how you will need to write your program interfacing with each peripheral before deciding how to connect them to the MSP432.
3. Utilize accurate schematic diagrams to guide your work. The final schematic for this project should include the MSP432, the LCD display, the keypad, and all required power supply connections. 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, CS, VCC, inside the chip where the wire attaches to the chip.
    e. Labeling all wires with a signal name, e.g. A21, D7, CLK, etc.

    The process of drawing an initial schematic does not have to be long or involved. It 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 finalized.

4. Sketch out what the software will do. Aids such as flow charts, pseudo-code, etc. may be used to do this.
5. After you have your complete program logic designed, you can start the actual code writing in Code Composer Studio (CCS). You can start by creating #define constants that will be useful.
6. Write your program in modules with a separate functions that perform repeated tasks. Write a function to write to the LCD display and verify it works properly. Comment and document your code as you write it. If you wait until everything works to go back and write comments, it is likely to never happen. Then write a function to read the keypad. The keypad functionality can be verified by writing values to the LCD with the functions you already have working. Then build your application as you designed in your flowcharts. If you run into an issue with the logic of your original design, go back and redesign your program with a new flowchart. Do not try to change aspects of your program without considering the how it functions as a whole.
7. 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.
8. Clean up your source code and add documentation because you probably ignored commenting it earlier. Break system functionality out into dedicated library files such as LCD.c and LCD.h so that functionality can be easily used in any future project.

P2 – Function Generator



Reference Material

- MSP432 Technical Reference Manual (TRM) – TimerA, SPI, Digital I/O
- Keypad Datasheet
- Microchip MCP4921 Datasheet

Objectives

1. Understand how to use timers and interrupts 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 function 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 DAC function from a time perspective such that no time is being wasted with unnecessary clock cycles or software delays.
    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. Run the DAC function with a TimerA ISR

    a. Start with the timer set to the time calculated in 1.b above. This value will be too low to function properly due to the extra delay of the ISR and function call.
    b. Increase the timer until the DAC is functional again. This is the maximum speed the DAC can be controlled with precision timing.
3. 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
4. The function generator shall be capable of producing
    a. A square wave with variable duty cycle
    b. A sinusoidal waveform
    c. A sawtooth waveform
5. All waveforms shall be DC-biased around Vdd/2 with Vdd set to 3.3 V.
6. 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 rate (points / sec) of at least 75% of the maximum determined from (2.b) above.
    d. The output rate shall not change with waveform frequency. The time between outputs from the DAC should be the same for all frequencies. This means the 100 Hz waveform will output 5x the number of points in one period as the 500 Hz waveform.
7. Upon power-up, the function generator shall display a 100 Hz square wave with 50% duty cycle.
8. The keypad buttons 1-5 shall set the waveform frequency in 100 Hz increments (1 for 100 Hz, 2 for 200 Hz, etc).
9. The keypad buttons 7, 8, and 9 shall set the output waveform to Square, Sine, and Sawtooth.
10. 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.
11. You may not use software delays (ex: delay_us) to generate timing events for outputting to the DAC. It would be more difficult to use software delays. You may use software delays to help debounce buttons (if necessary).

Tips and Suggestions

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 your DAC function. Note the parameters *"/CS Fall to First 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

Reference Material

- MSP432 Technical Reference Manual (TRM) – TimerA, ADC14
- Microchip 24LC256 EEPROM Datasheet

Instructions

Project 3 requires the design and implementation of a digital multimeter (DMM) that can measure voltage and frequency to meet all of the specified requirements below.

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 +/- 10 mv for AC and DC.
    1.4. The DMM shall have a DC setting.
        1.4.1. DC measurements shall average over a time period greater than 10 µs and less than 0.1 s.
        1.4.2. DC measurements of a sinusoidal waveform should be equivalent to the DC offset.
    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

Demonstration

Demonstrate the completed multimeter device with each input specified below for an instructor or lab assistant to complete the signoff sheet.

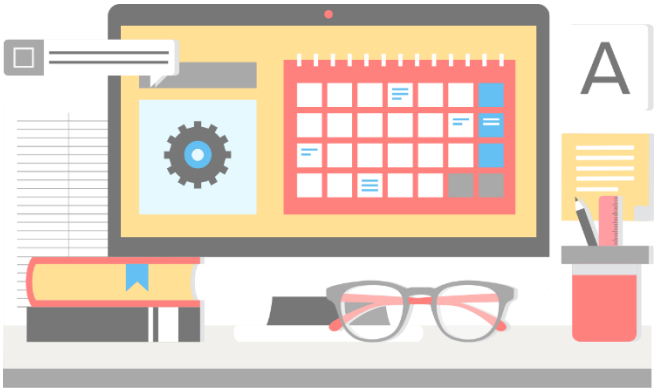
1. DC Voltage
    a. Two different DC values between 0 and 3.3 V
2. AC Voltage
    a. 3 Vpp sine wave with 1.5 V DC offset at 500 Hz
    b. 1 Vpp sine wave with 0.5 V DC offset at 1 kHz
    c. 1 Vpp sine wave with 2.5 V DC offset at 1 kHz
    d. 3 Vpp triangle wave with 1.5 V DC offset at 50 Hz
    e. 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)

Signoff Sheet

Name(s): Date:

		Measurement			
Test	Input	DC	True RMS	Vpp	Frequency
DC Voltage 1					
DC Voltage 2					
AC Voltage 1	3 Vpp Sine 1.5 V DC 50 Hz				
AC Voltage 2	1 Vpp Sine 0.5 V DC 1 kHz				
AC Voltage 3	1 Vpp Sine 2.5 V DC 1 kHz				
AC Voltage 4	3 Vpp Triangle 1.5 V DC 50 Hz				
AC Voltage 5	1 Vpp Square 1.5 V DC 50 Hz				
Frequency 1	3 Vpp Sine 1.5 V DC	Minimum		Maximum	
Frequency 2	3 Vpp Square 1.5 V DC	Minimum		Maximum	

P4 – Design Project



Project Description

The design project should be a culmination of embedded systems topics learned and experience gained in this course. This project should utilize a sensor, actuator, or another new peripheral as a central theme to the system you develop. This peripheral must be interfaced to and calibrated, which will be an intermediate deliverable. The final system developed needs to be a stand-alone device which provides useful or entertaining functionality. Good examples of a stand-alone embedded systems are devices like a digital alarm clock, hand-held medical thermometer, GPS navigation unit, and interactive LED light board. Your design project system may have other interesting components or subsystems besides the sensor, actuator, or key peripheral.

Selecting a Peripheral

Table 1 below provides links to a variety of 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 peripherals. Each group will purchase all components required for this project. Don't forget to include delivery time into your project schedule.

Company	Website
Sparkfun - easy to browse selection by category	https://www.sparkfun.com
DigiKey - extensive selection, but poor browsing	https://www.digikey.com
Digilent - easy to browse, everything uses pmod connection	https://store.digilentinc.com/pmod-modules/
Adafruit - ok browsing, be mindful of poor or no documentation	https://www.adafruit.com
Texas Instruments BoosterPacks - Designed to connect directly to MSP432, but be mindful of third party shipping times	https://www.ti.com/tools-software/launchpads/boosterpacks/select.html

Peripheral Interfacing and Calibration

The first step in building a completed system after selecting a peripheral, is to interface to it, and calibrate it with real world data. To show this is done correctly, you will build a simple proof of functionality system. For a sensor, the calibrated data could monitored and displayed on an LCD or terminal. For an actuator, motion could be controlled with the keypad or terminal. The specific proof of functionality will be dependent upon the peripheral selected. Your choice of peripheral will largely impact what you learn with this project. Most sensors produce an analog output that must be sampled and converted using the ADC14. Other sensors interface serially via SPI, I2C, or UART. View Technical Note 8 for guidance on calibrating your peripheral.

Battery Power

The final device will require battery power operation. This can either be the sole power source or as a battery backup failsafe or UPS. The power regulators on the debugger part of the MSP432 development board can be utilized in the battery power system, but they may not be interfaced via the micro-usb connector on the development board. The final report for the completed system must include a battery power section that shows calculations to predict battery life. Current or power measurements can also be collected and documented in this section. Any optimization of your code to reduce power use or run in lower power states should be documented in this section.

Device Enclosure

The final device is required to be housed in a self-contained enclosure. The enclosure can be a standard size box (available at DigiKey and other electronic component distributors), 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.

A detailed tutorial on creating a 3D printed enclosure is available on Sparkfun
https://learn.sparkfun.com/tutorials/getting-started-with-3d-printing-using-tinkercad/all

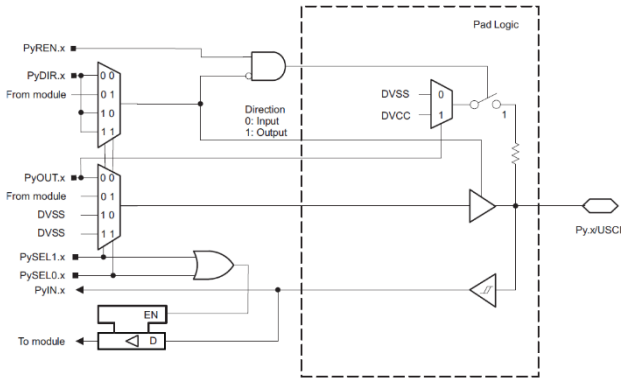
Demonstration

Each lab group will be required to give a short demonstration presentation of the final project. The presentation should last no more than 5 minutes and include the following:

1. Description of the project – what does it do
2. Overview of technical details
    a. what peripherals were used
    b. basic overview of the software architecture
    c. functionality specifications - battery life, etc.
3. Live demonstration of the device

Technical Notes

TN1 – GPIO Interrupts



Background Information:

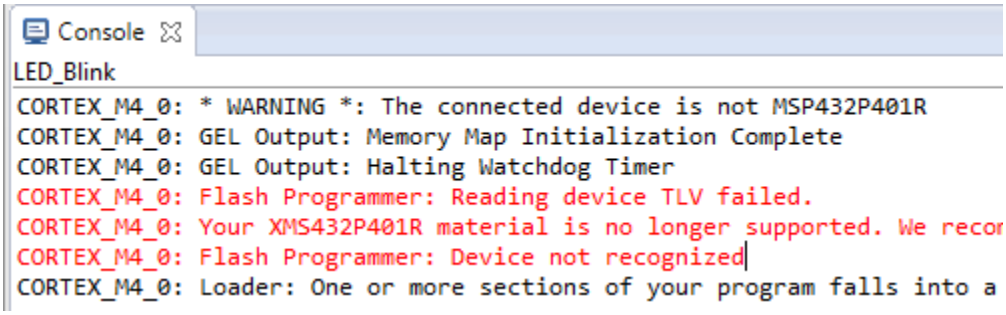
Circuit for generating interrupts via button connected to P1.3. LED on P1.0 toggles based on button press. Note that code has options for running in low power mode 4 or with main loop.

```
#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

The MSP432 has 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

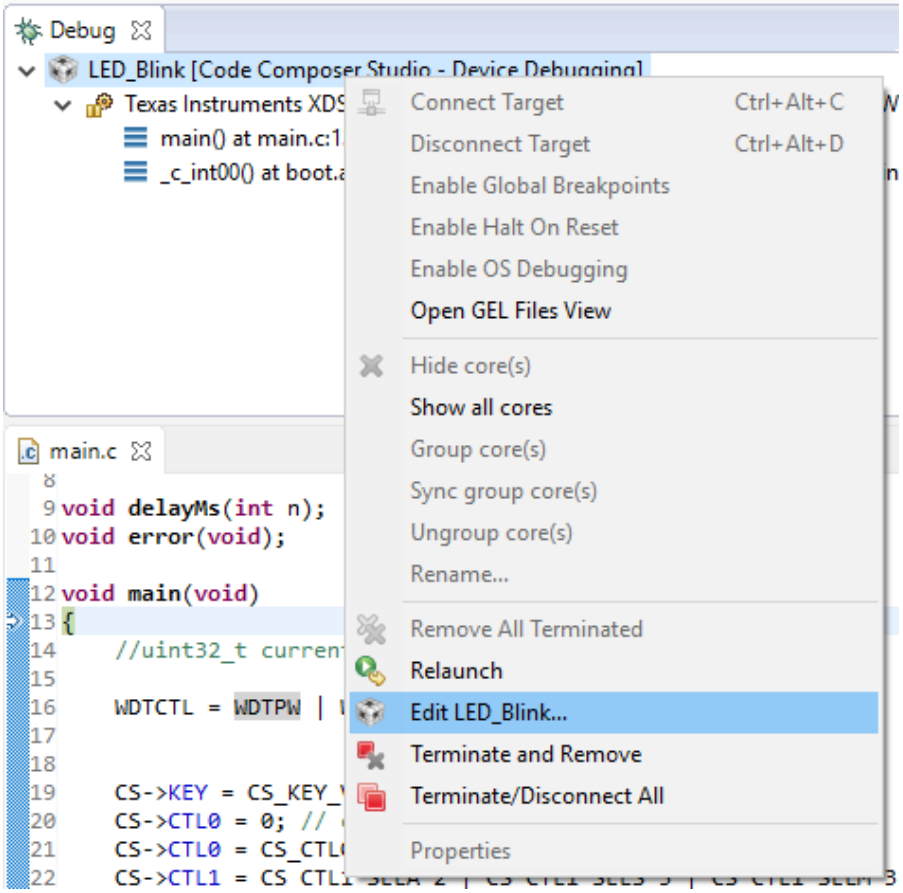


If Code Composer Studio pops up an error dialog specifying that it failed to load your *project_name.o* file and closes the debug interface, you will need to change your target settings before you can connect to the Debug Probe. If you do not get this error, you can skip to Step 4.

Reset Procedure

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

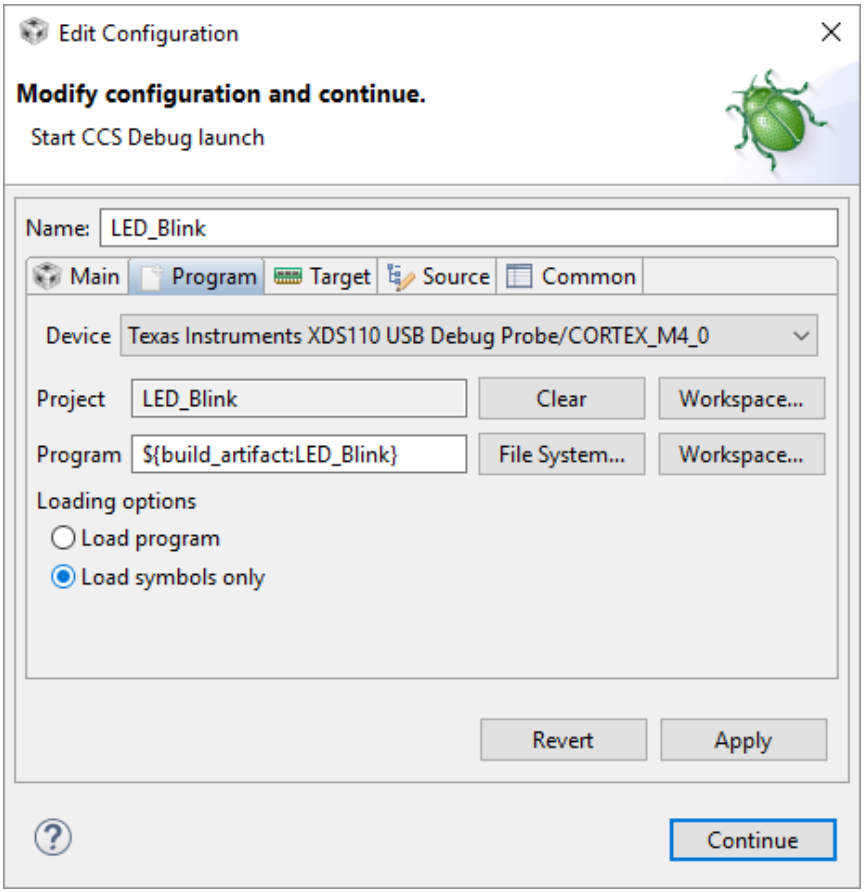
Step 2. From the debug perspective, right click on your **CCS – Device Debugging** and select Edit *project_name* …



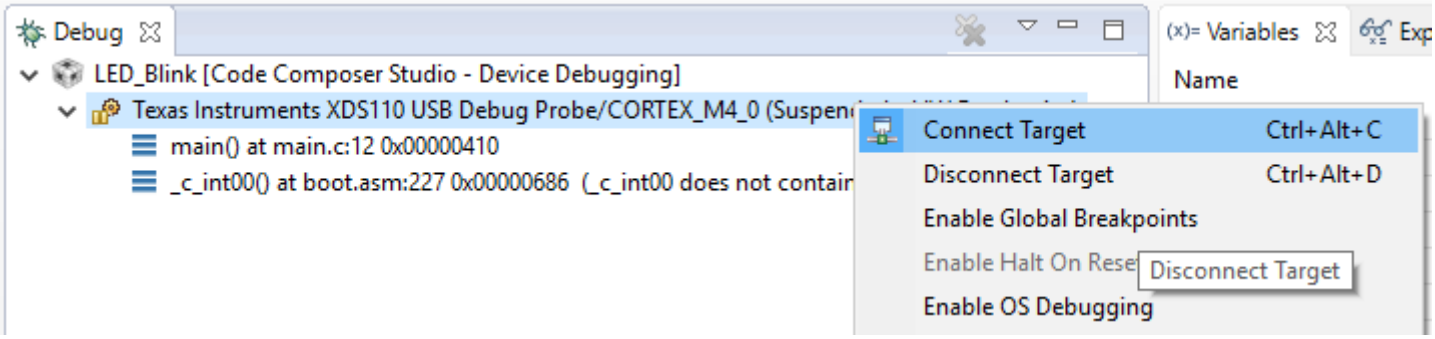
Step 3. Select the **Program** Tab and choose **Load symbols only**.

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

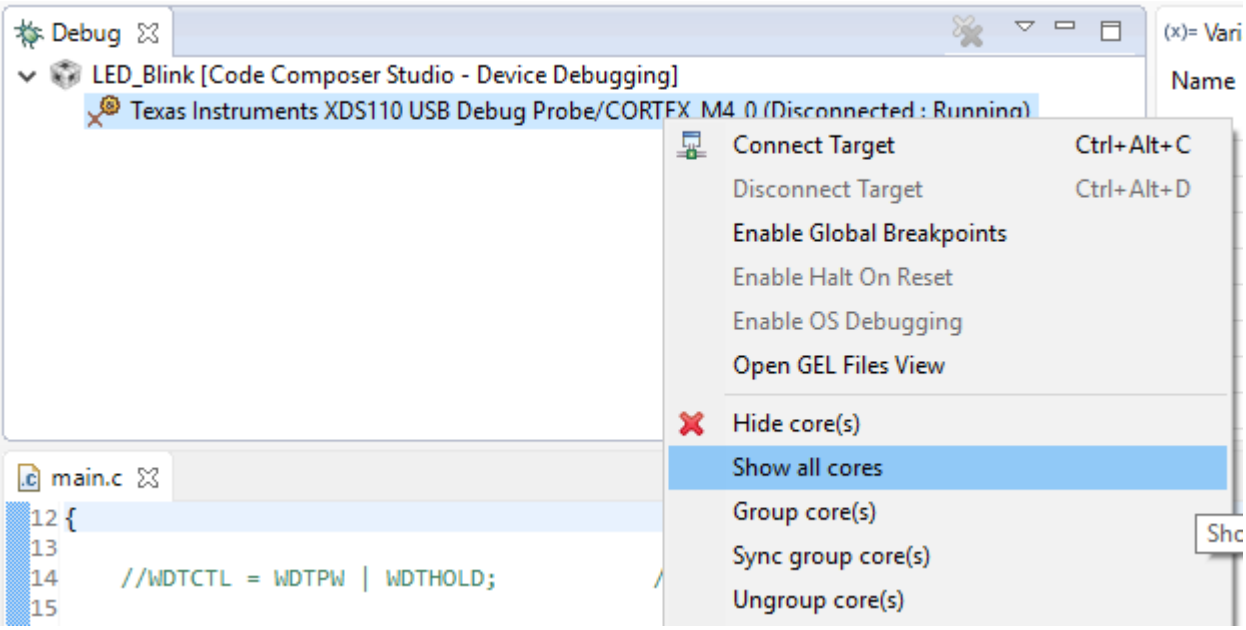
Try to program your MSP432 again by clicking on the Debug button like normal. Programming should complete with no error popup dialog.



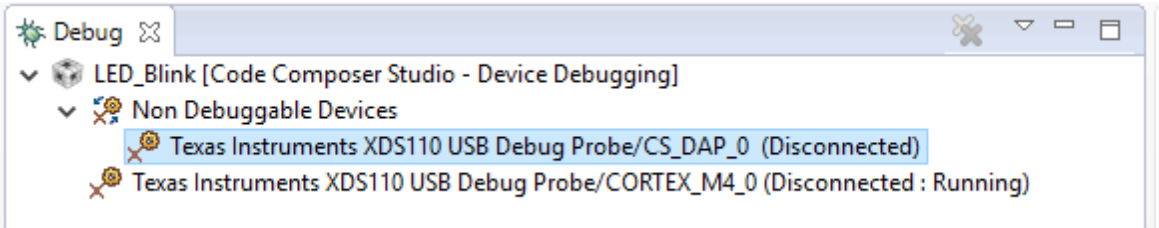
Step 4. Select the **XDS110 USB Debug Probe** from the Debug window, right click and select **Connect Target**



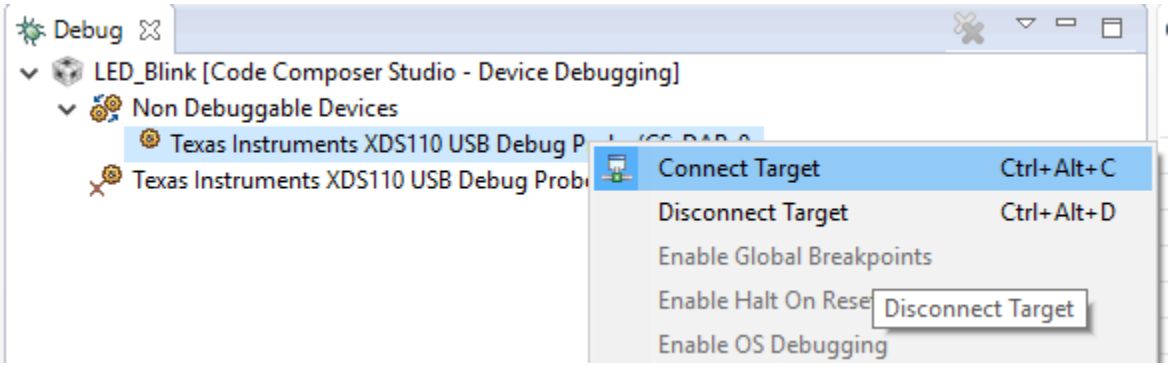
Step 5. After connecting, select the same **XDS110 USB Debug Probe**, right click and select **Show all cores**



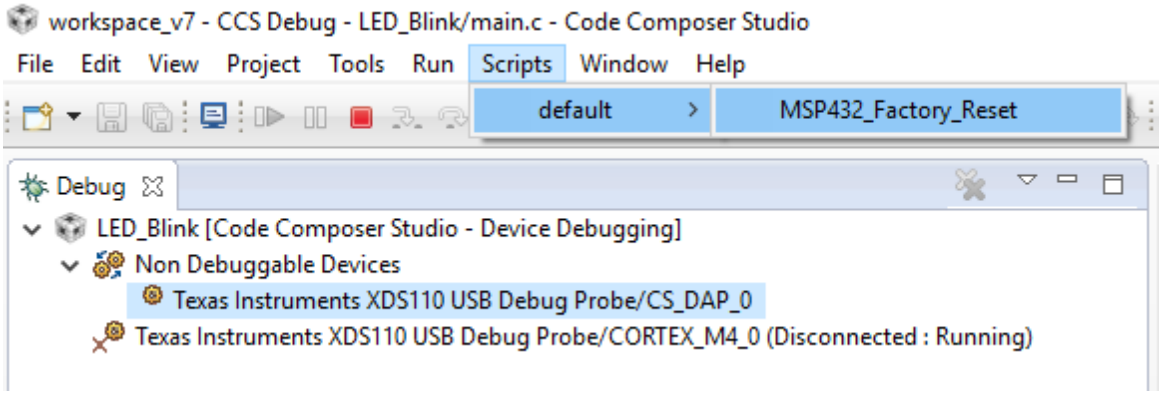
Step 6. A new device should show up on your debug window. **XDS110 USB Debug Probe / CS_DAP_0**



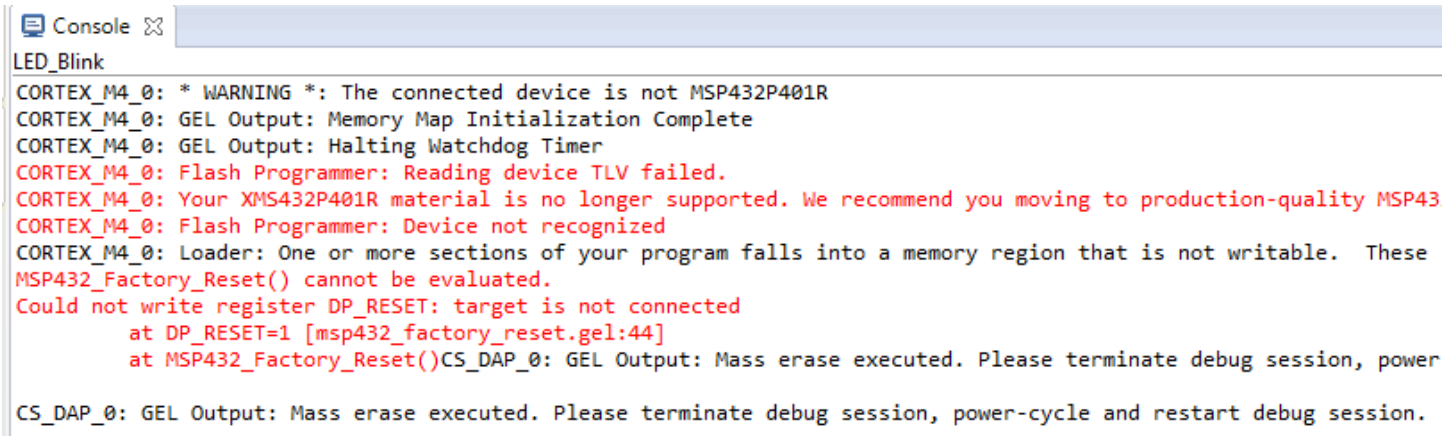
Step 7. Select **XDS110 USB Debug Probe / CS_DAP_0**, right click and select **Connect Target**



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.

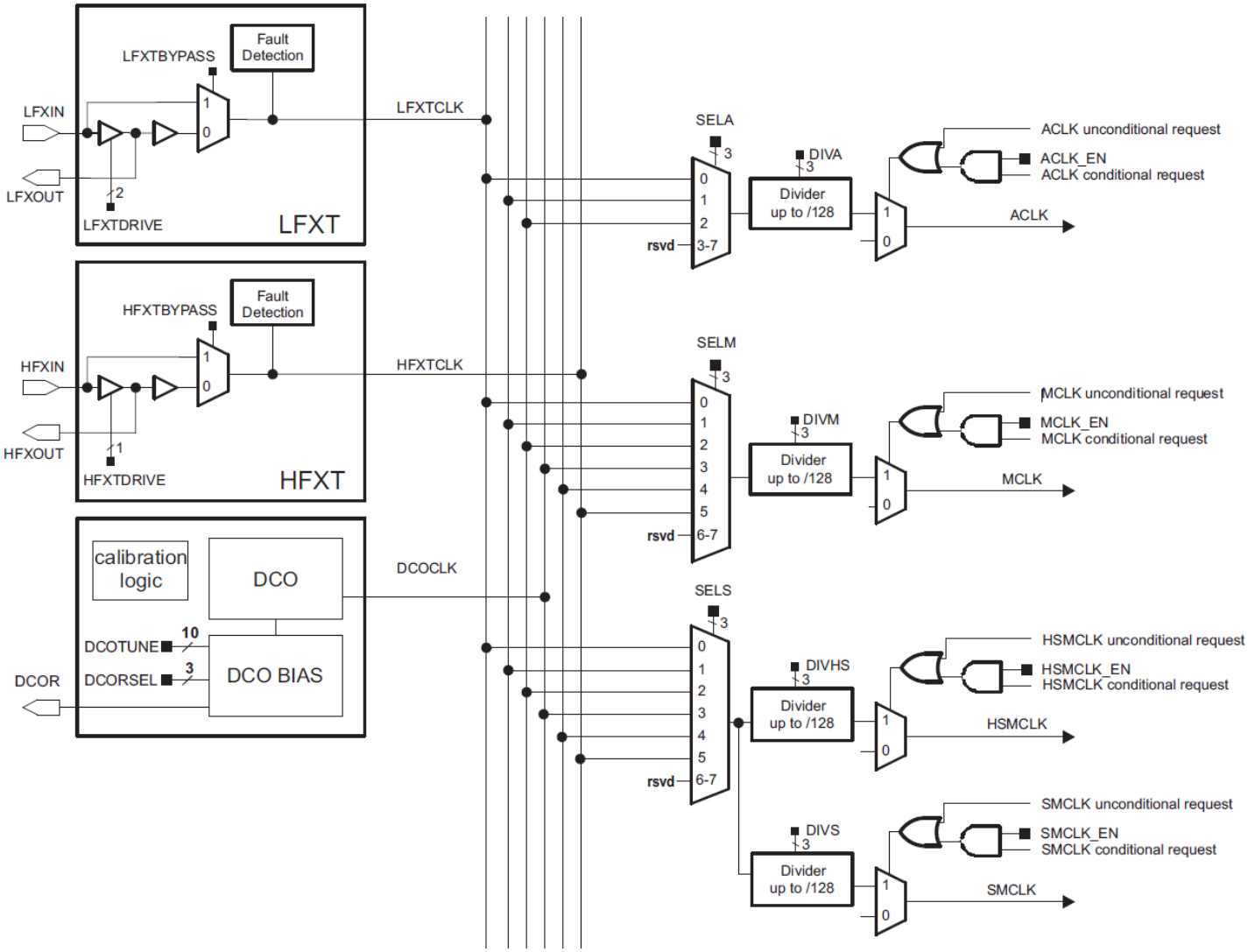


Diagram 1: MSP432 Clock System

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.

Higher operating frequencies use more power to accommodate the faster transistor switching. Operating at the maximum frequency of 48 MHz requires a higher core voltage to meet the timing requirements. The power control manager (PCM) is used to adjust this higher Vcore voltage. If the master clock signal (MCLK) is changed before ensuring the Vcore voltage 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 PCM takes time to change Vcore voltages. No changes to MCLK should occur until after the Vcore voltage has settled after being changed.

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
```

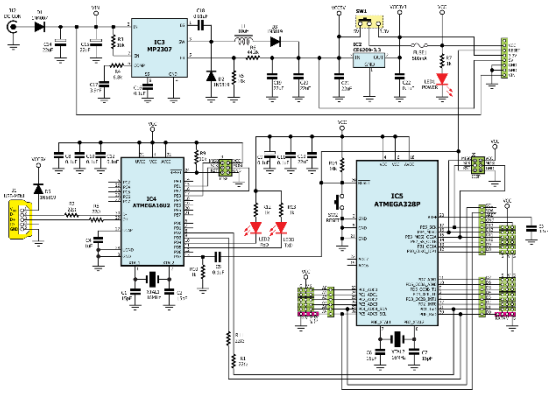
Using external crystal oscillators

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 access
```

TN4 – Schematic Best Practices



Schematic Standard Formatting

Creating schematics that are clear and follow standard formatting practices is necessary for all technical documentation of electronic devices. Schematics are critical for diagnosing and repairing malfunctioning electronics, so providing all of the necessary information in a clear and concise format is important.

Component Labels

All components in the schematic require a unique identifying label, typically in the form of a letter and number combination. The letter is used to identify what type of component is being referenced. Common identifiers include:

- R – Resistor
- C – Capacitor
- L – Inductor
- D – Diode
- Q – Transistor
- U / IC – Integrated Circuit
- Y – Crystal
- S – Switch
- J / JP – Junction, Connector, Jumper Pin
- T – Transformer
- K – Relay
- M – Motor
- F – Fuse

The letter identifier is followed by a number with each component of the same type incrementing the value. The numbering will typically start at one corner of the finished schematic and increment across the schematic. This creates that result of numerical values reflecting proximity, so R2 should be close to R1 and R3, and farther away from R20. The numbering between different types of components do not necessarily relate proximity, so R20 does not have to be close to C20.

Passive components should also be labeled with their component size or value using appropriate units. For example 10 kΩ rather than 10000 Ω. Units should use appropriate symbols and prefixes. Use 10 uΩ instead of 10 micro-ohms. This value label should be located above or below the component identifier.

IC and Chip Labeling

ICs should include their specific model number with the IC identifier. The outline of the IC should contain lines for every connected pin. Smaller ICs will typically have the pin locations correspond to their physical location on the chip. As the number of pins increases, this can create messy and hard to follow wire traces. To make the schematic readable, the pin locations can be moved to different sides on the schematic. Schematics can

connect pins to all 4 sides of the IC or just 2 (left and right). A schematic should let the reader's eye follow the flow of signals when possible. Pin locations should be equally spaced apart on the schematic IC.

Each connected pin of the IC requires 2 identifying labels. Inside the IC the signal or port name connected to the pin should be labeled like P1.0, VCC, SCLK, or MOSI. This label should be centered where the pin connects to the chip. Immediately outside the IC, the pin number should be printed above the pin wire. The pin number corresponds to the pin location based on the specific package of the IC used in building the circuit. Refer to the example in Figure 1 below.

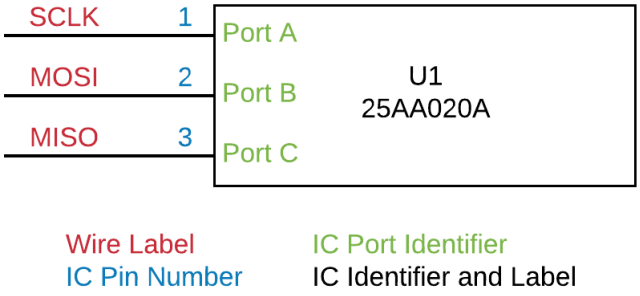


Figure 1: Example IC Schematic

Drawing and Labeling Wires

Wires should be spaced uniformly on a schematic. Groups of wires can be joined to form a bus for ease of reading, but the individual wires must be labeled to show how their order in the bus makeup. This is necessary to allow individual wires to be traced through the bus. While most wires require no label, some signal names can be labeled above their specific wires to help in ease of reading the schematic. These labels are required when wires connect off the current page to another schematic.

When drawing wires, crossing wires in a schematic do not signify they are connected. Jumping wires to show they are unconnected are valid, but not used in professional schematics. The jumps make the wiring appear busy and harder to follow. To show that wires are connected, a dot should be added at the connection point. For T junctions of 3 wires connecting, adding a dot is valid, but unnecessary. Most professional schematics will not include the dot in this situation. See Figure 2 below for reference.

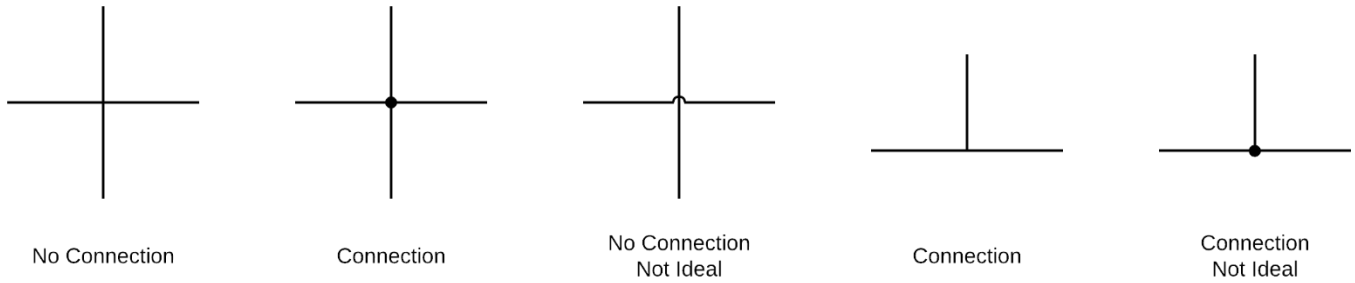


Figure 2: Example Wire Connections

General Layout Considerations

- Power connections should generally point upward on the sheet while ground connections should point downward
- IC shapes should have a consistent size. While a 14 pin chip should be larger than an 8 pin chip, all 14 pin chips should have the same size rectangle.
- Be mindful of color use. All schematics must also be understandable in black and white.

Additional Resources

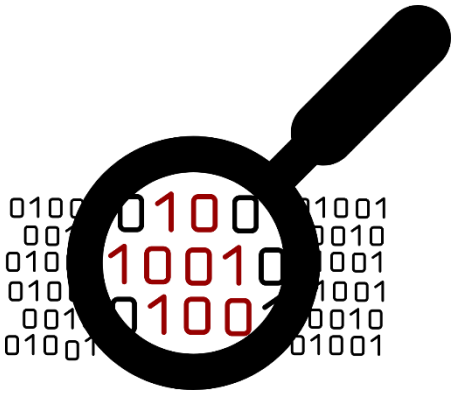
https://learn.sparkfun.com/tutorials/how-to-read-a-schematic

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

Software Tools for Creating Schematics

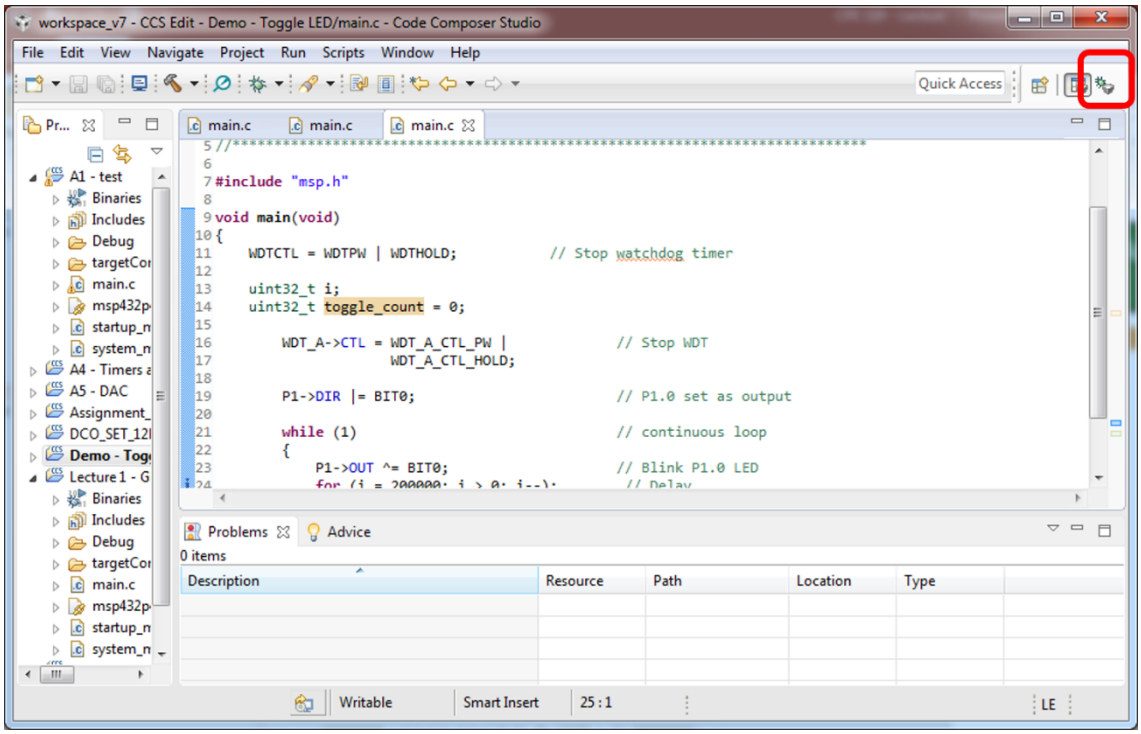
- CircuitLab https://www.circuitlab.com
- Lucidchart https://www.lucidchart.com
- Draw.io https://www.draw.io
- Microsoft Visio https://products.office.com/en-us/visio/flowchart-software

TN5 – Watching Expressions in CCS

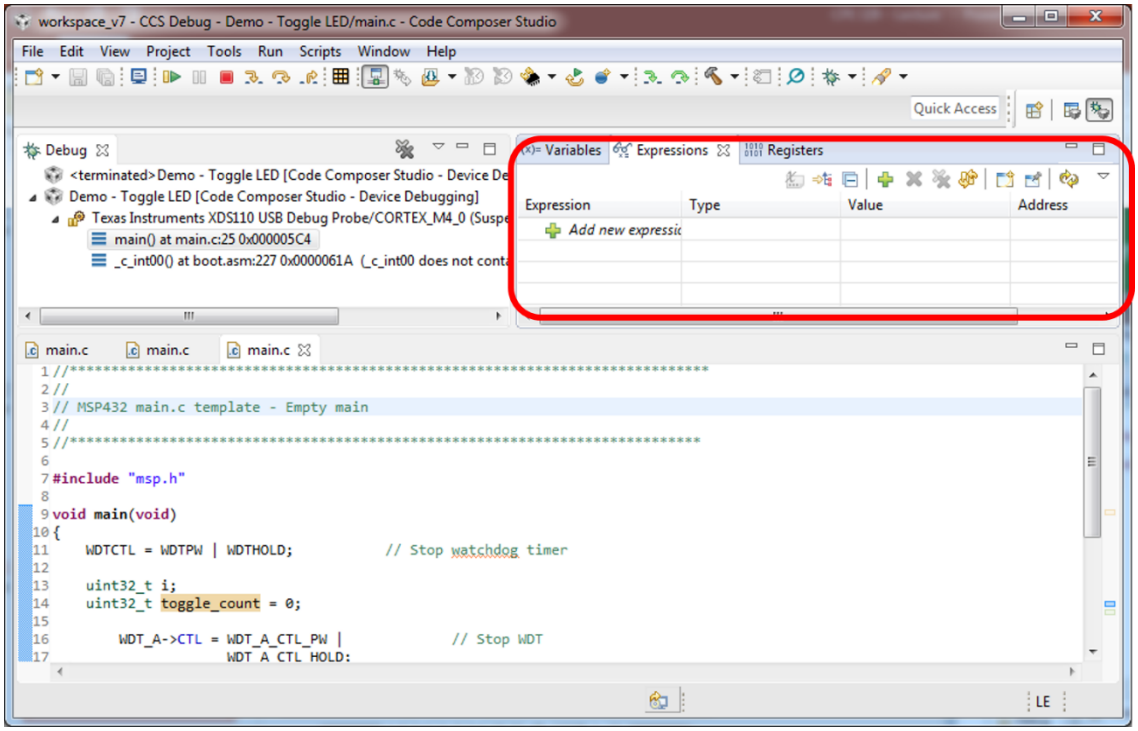


Instructions

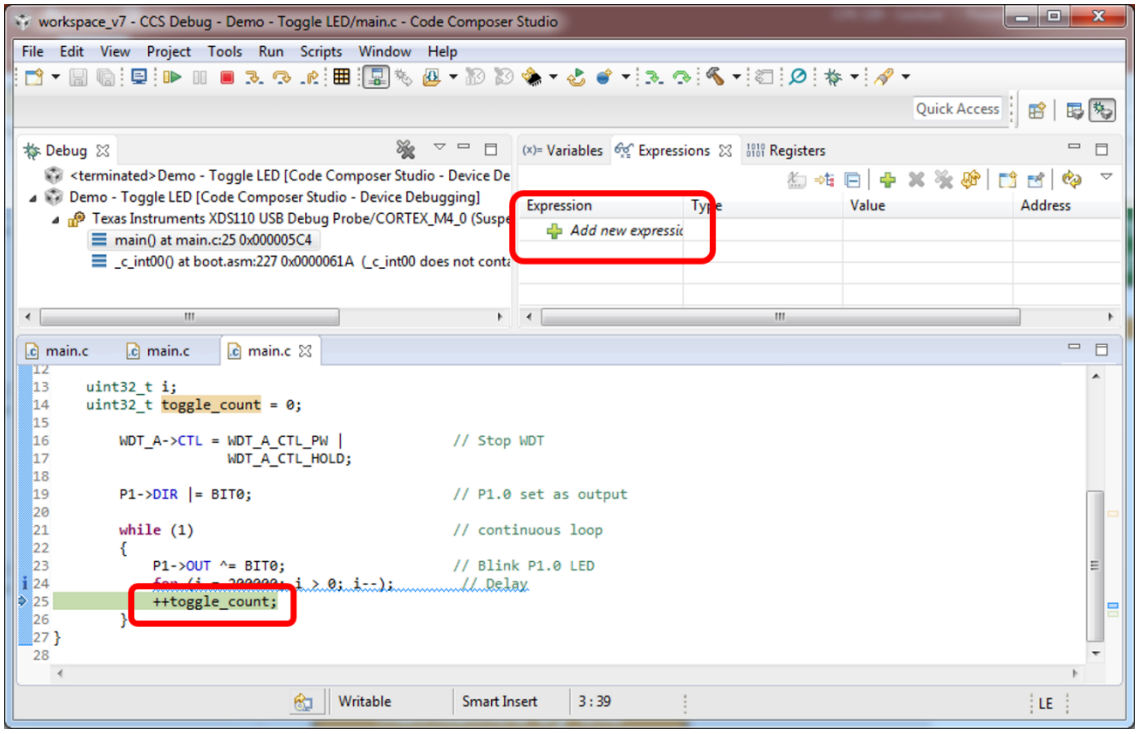
1. Enter Debug Mode by selecting the Debug Perspective



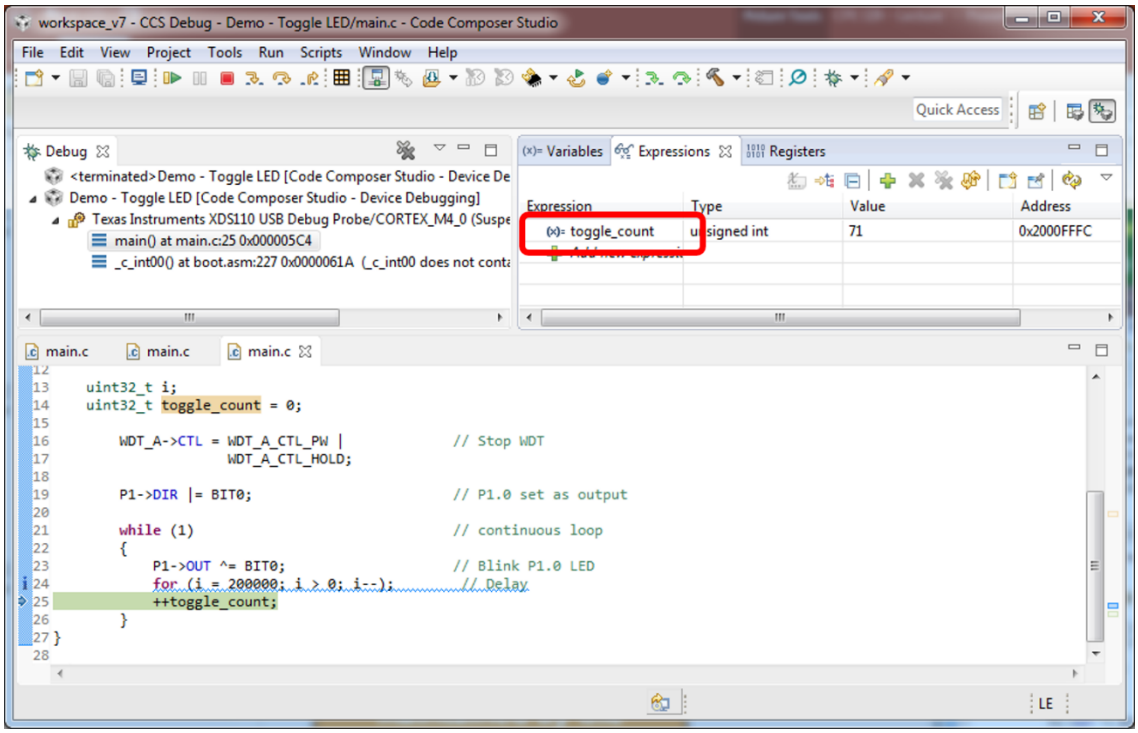
2. Select the Expression Tab. It may be necessary reset the perspective to show the debug pane. Select Window->Perspective->Reset Perspective



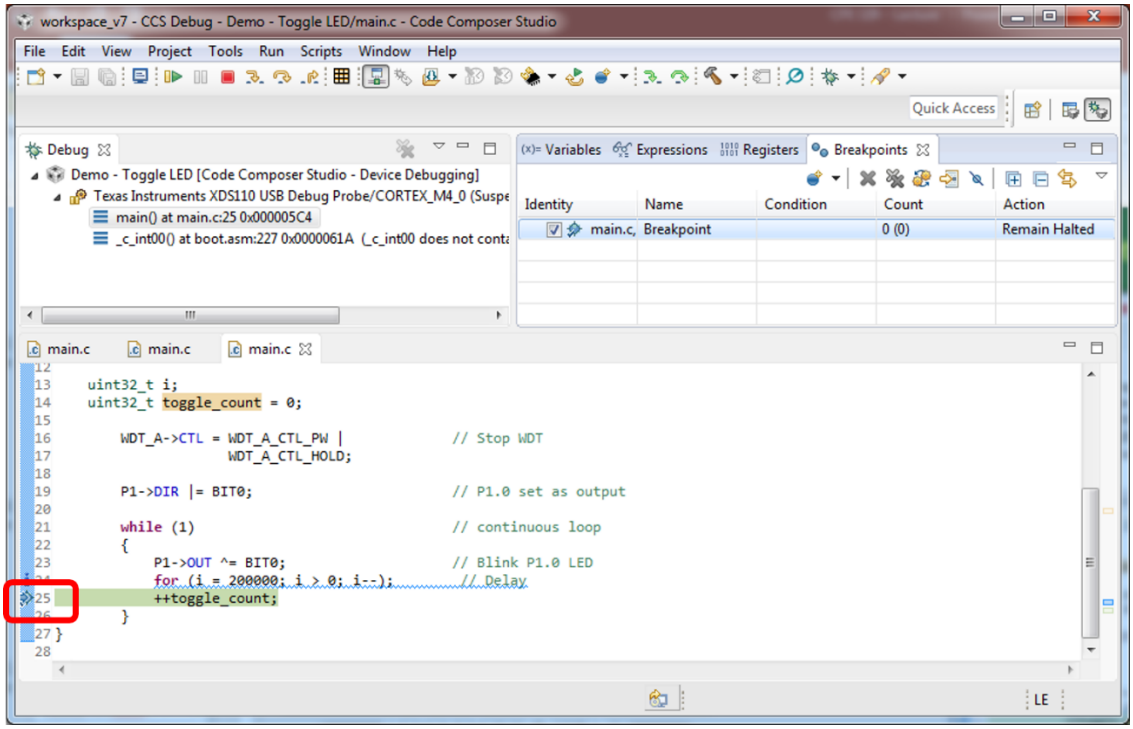
3. Click Add new expression and enter expression to watch (`toggle_count`)



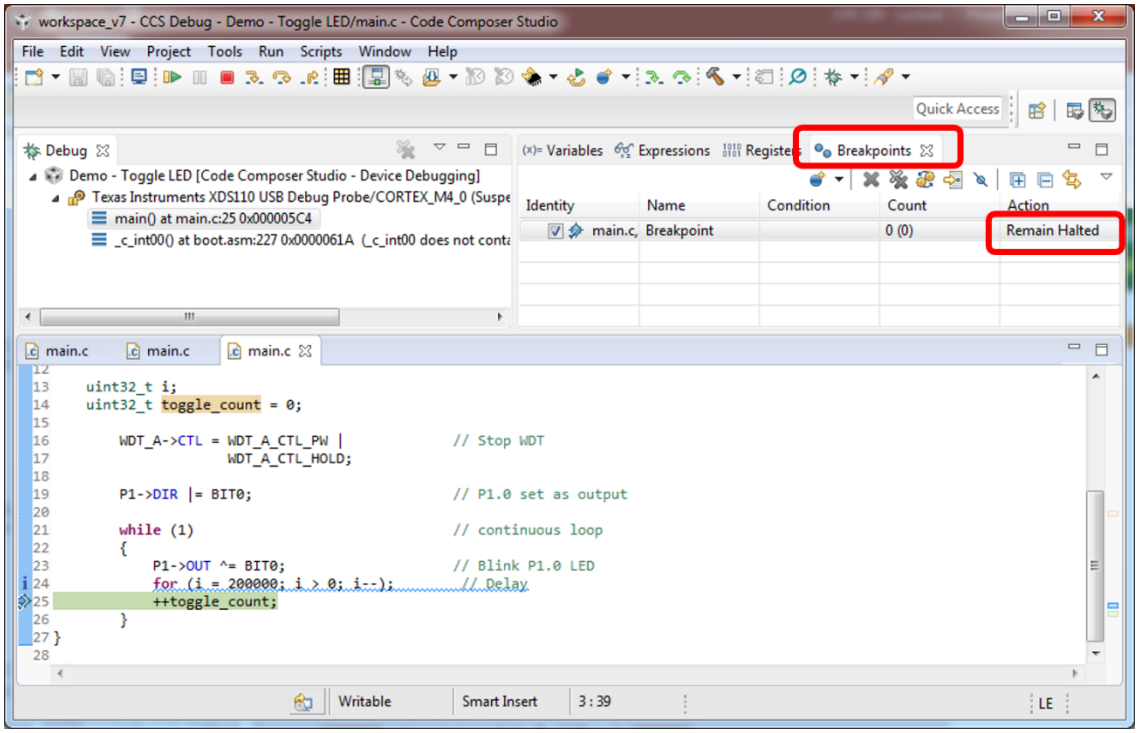
4. The expression `toggle_count` will now be watched. By default the value will only be updated between “steps” or when the program execution is paused or halted.



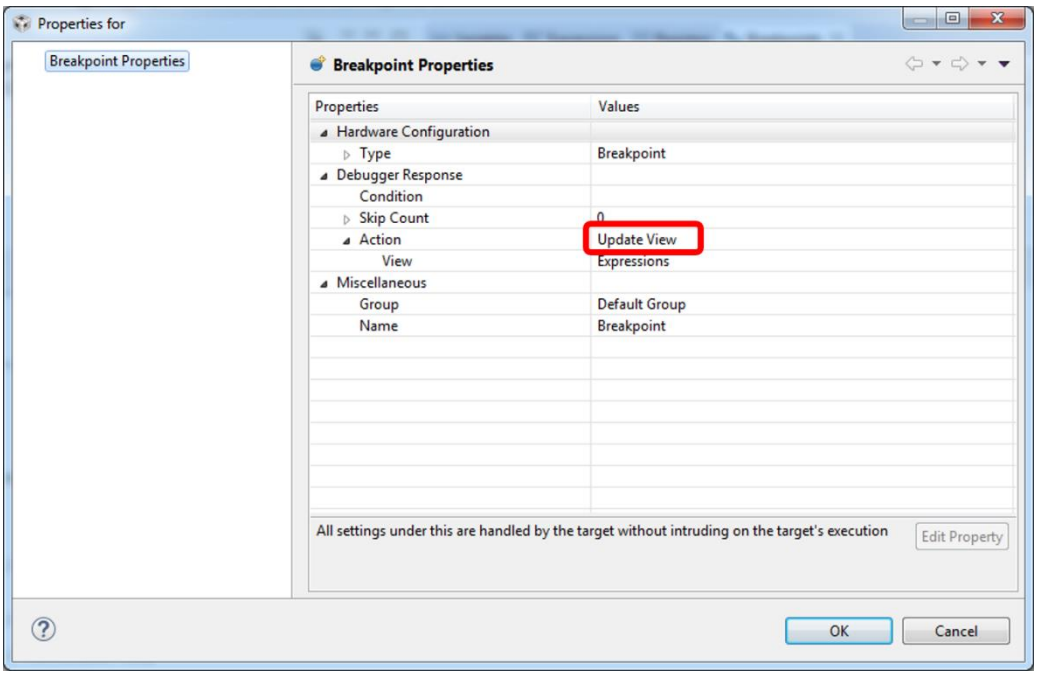
5. The expression can be forced to update at specific times during the program execution while running “live”. The location in the program when the expression value should be updated is determined by creating a breakpoint. Create a breakpoint by double clicking in the left hand margin on a line of code or right clicking and select Breakpoint (code composer studio) -> Breakpoint



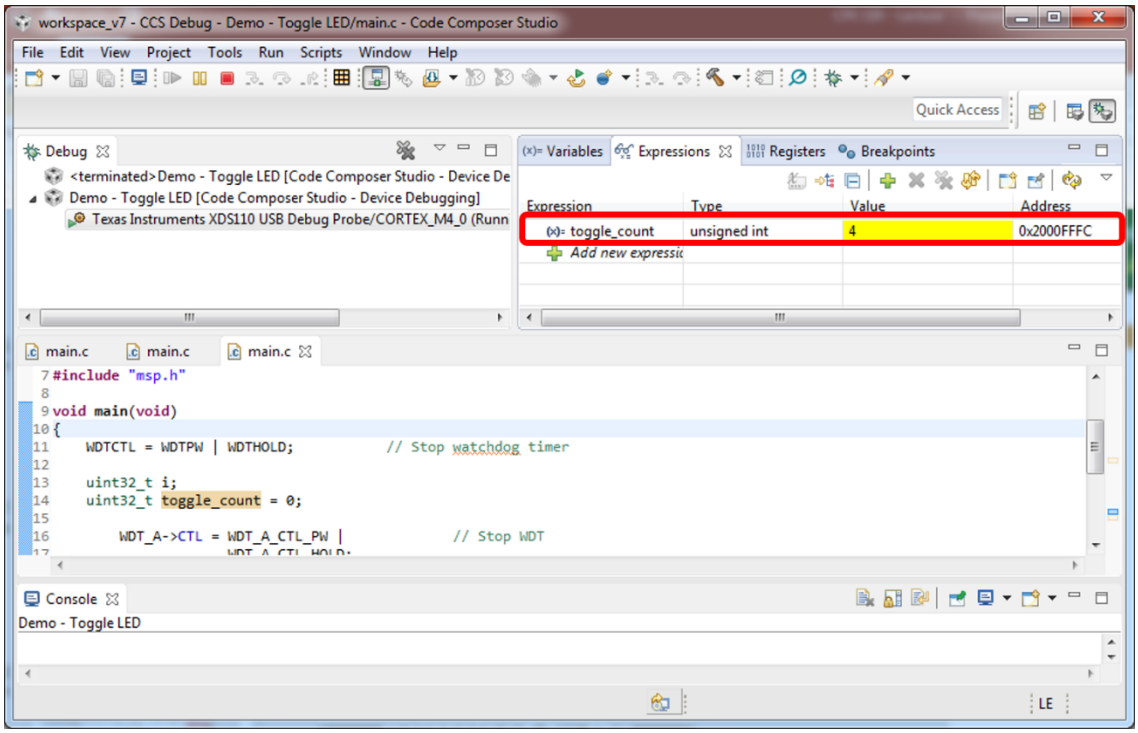
6. The Breakpoints tab will be added to the debug pane. By default execution will halt when reaching a breakpoint. This behavior can be changed to continue execution without pausing. Right click on Remain Halted and select Breakpoint Properties



7. Change the action to Update View. This will allow the code to continue to execute while updating the value of the expression in the debug pane.



8. Restart debug mode. The expression will update as it changes in software. While the program is executing and watching the expression in “real time”, the execution timing is still delayed and operates slower than when executing without debugging.



TN6 – VT100 Terminals

Background Information:

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.

Terminal Applications

Windows

- RealTerm https://realterm.i2cchip.com/

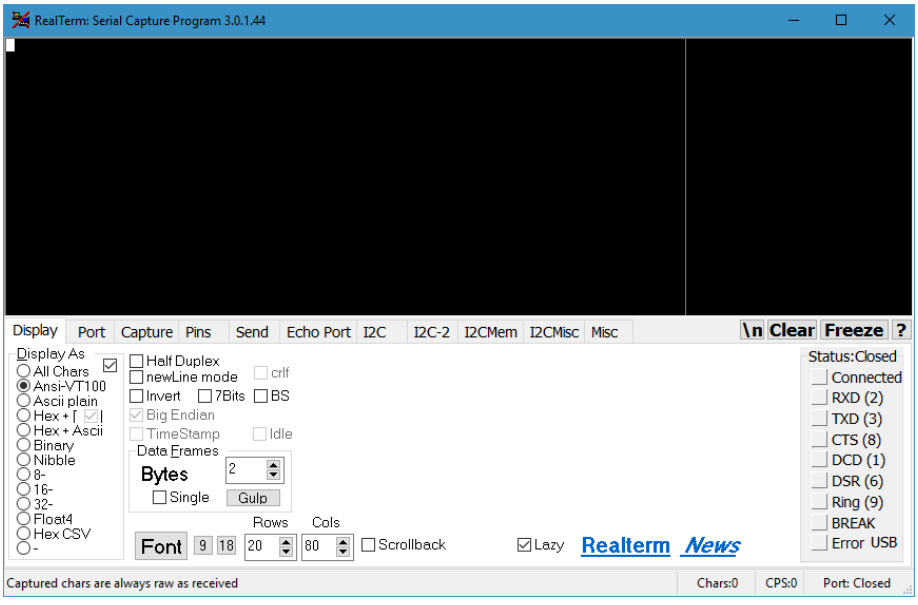


Figure 1: RealTerm VT100 supported with option Ansi-VT100

- Tera Term https://ttssh2.osdn.jp/index.html.en

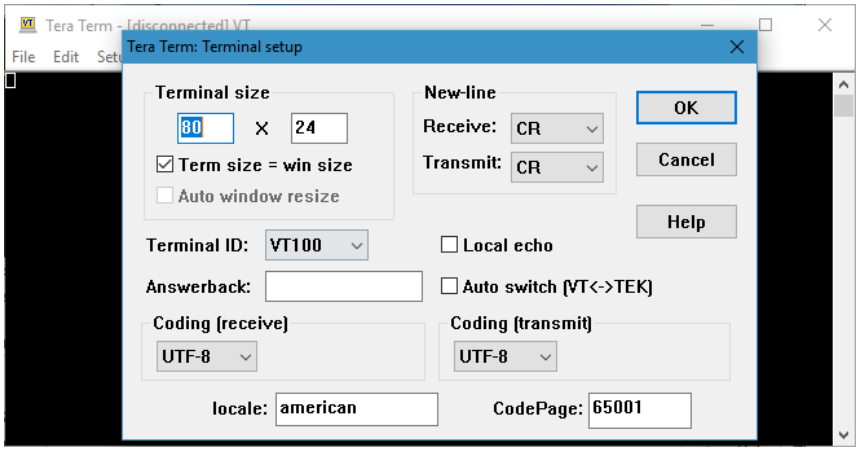


Figure 2: Tera Term VT100 setting Terminal ID

Linux

Most shells support VT100 escape codes by default (bash, zsh, csh, sh, ksh, tcsh)

Escape Codes

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 below 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 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 hex value 0x1B.

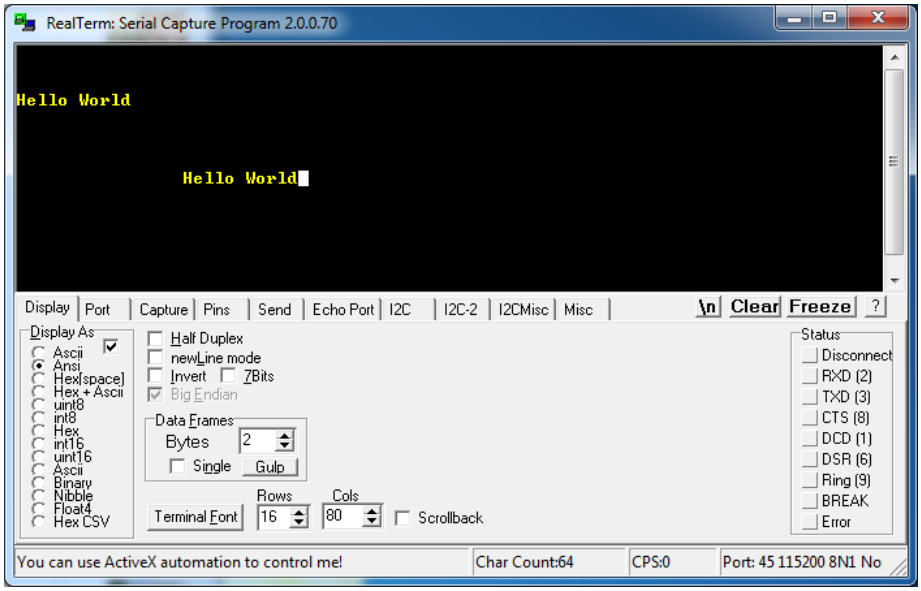
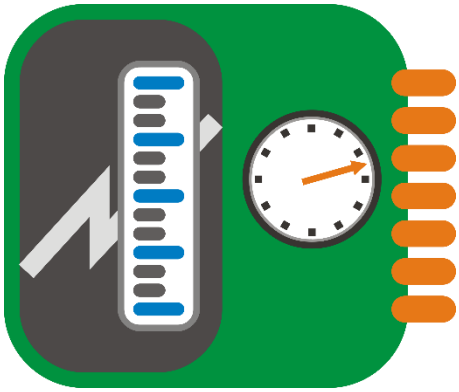


Figure 3: Example output from command sequence above

A table listing of VT100 escape sequences: 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

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

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

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

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.

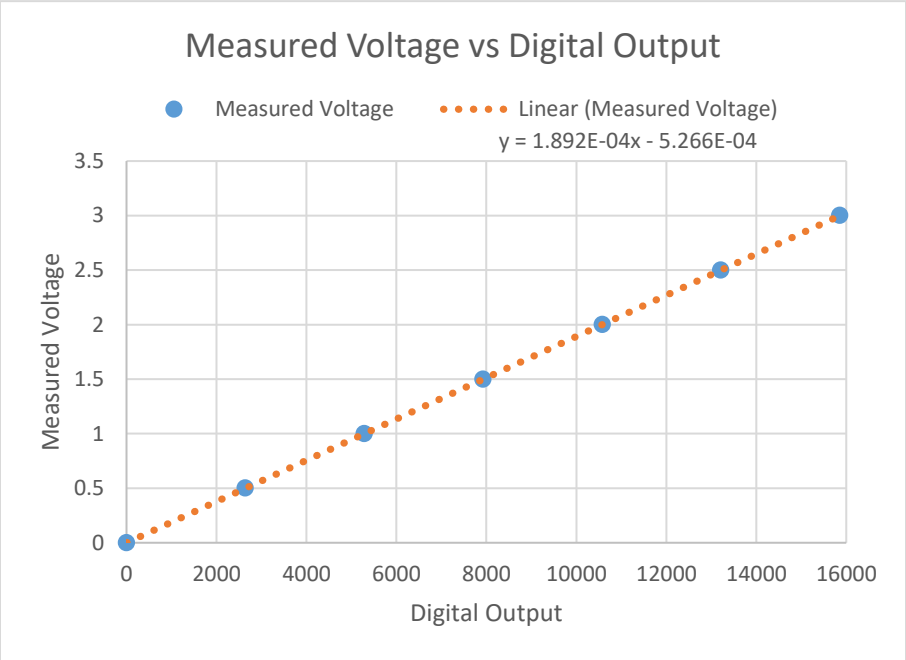
Linear Approximation Calibration Methodology

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

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

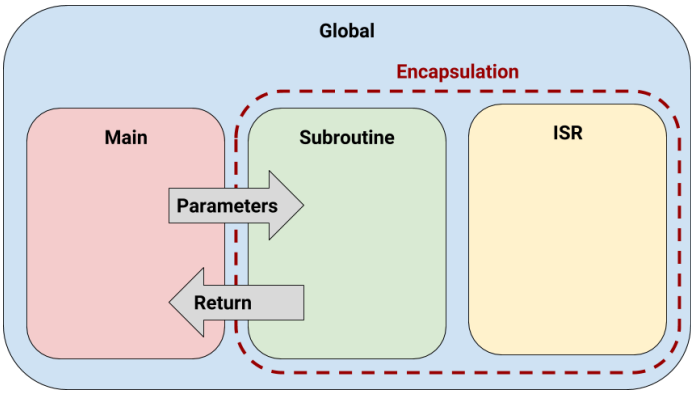
Digital Output	Measured Voltage (V)	Calibrated Value (µV)
6	0	607
2642	0.5	498811
5288	1	998905
7930	1.5	1498243
10579	2	1998904
13215	2.5	2497108
15861	3	2997202

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
   }
}
```

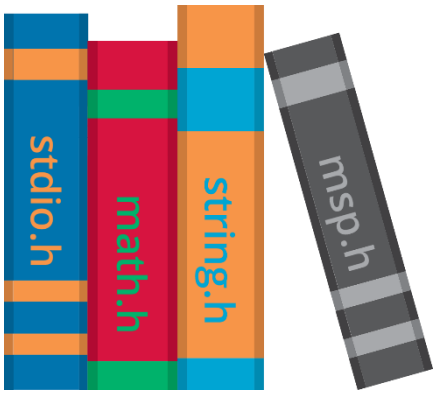
mydriver.c

```
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 – Create Custom Libraries with #include Header Files



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 Implementation

When writing code for a specific device or peripheral, all of the functions that would typically be written to utilize it can be grouped together in a library of code (.c) and definitions and declarations (.h). For example, if working with a GPS sensor, functions to use the device could include the following: GPS_Init, GPS_Config, GPS_ReadStatus, and GPS_ReadLocation. These functions can be separated into a set of files that can be easily included in future projects to reuse the code and easily implement the same GPS module in a new project.

Header file

Create a header file to include all defined constants and function declarations. This allows for easy editing of constants

GPS.h

```
#define GPS_FREQ 120000
#define GPS_MISO_PIN 4
#define GPS_MOSI_PIN 7

void GPS_Init(void);
uint8_t GPS_Config(uint32_t GPS_options);
uint32_t GPS_ReadStatus(void);
uint32_t GPS_ReadLocation(void);
```

Code File

Create a code file that implements all of the functions.

GPS.c

```
void GPS_Init(void) {
   // code implementing initialization function
}

uint8_t GPS_Config(uint32_t GPS_options) {
   // code implementing configuration of GPS module
}

uint32_t GPS_ReadStatus(void) {
   // code to read GPS status
}

uint32_t GPS_ReadLocation(void) {
   // code to read GPS location
}
```

Adding Library

Both of the created GPS.h and GPS.c files can be added to any project folder and then inside the main.c file, add `#include GPS.h`. This will tell the compiler where to find the functions when called. Any of the defined constants or functions can be called and used from the main program.

Homework Questions

HW1 – 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 – Square Wave Duty Cycle

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.

Problem 4 – 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.

```
uint32_t TimeMs;
uint32_t main(void)
{
      WDTCTL = WDTPW + WDTHOLD; // Stop WDT
      SMCLK_12MHZ();            // SET SMCLK AT 12MHZ

}


void PORT1_IRQHandler(void)
{

}
```

HW2 – 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). Only use Port 1 for connections.

Problem 2 – LCD Control Code

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 – 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.

HW3 – TimerA

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 1 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.

Problem 2 – Timer Control with Interrupts 2

Please complete the following source code to utilize interrupts and Timer A in continuous 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.

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 – TimerA 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

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.

Problem 6 – Bit Toggling with ACLK

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.

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.

HW4 – SPI and DAC

Problem 1 – Timer Control with Interrupts 1

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.

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?

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

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.

HW5 – Clock System

Problem 1 – DCO vs Crystal Oscillator

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

a. The DCO can run at different frequencies.
b. DCO frequency can be changed in software.
c. DCO stabilizes more quickly.

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 Crystals

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

Problem 4 – ACLK Speeds

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:

a. ACLK to be 4096 Hz
b. MCLK to be 32768 Hz
c. SMCLK to be 2 MHz

HW6 – UART

Problem 1 – UART Frequency Error

Please describe 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"

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.

HW7 – I2C and EEPROM

Problem 1 – EEPROM Interfacing

Draw a ***detailed*** schematic diagram (pin numbers, signal names, etc.) showing how the MSP 432 would be interfaced to the Microchip 24xx256 serial EEPROM.

Problem 2 – EEPROM I2C Data Transmission

Identify what bits 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 need to show the Start or Stop conditions.

Problem 3 – I2C Bus

Modify the schematic from Problem 1 to add a 2nd EEPROM.

Problem 4 – I2C Bus

How would the bits from the data line from Problem 2 change if the EEROM was absent from the system

HW8 – General

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 MSP430?