LABORATORY MANUAL

DEPARTMENT OF
ELECTRICAL & COMPUTER ENGINEERING

UNIVERSITY OF CENTRAL FLORIDA

EEL 4742
Embedded Systems

Revised
August 2015

Table of Contents

Safety Rules and Operating Procedures

1. Note the location of the Emergency Disconnect (red button near the door) to shut off power in an emergency. Note the location of the nearest telephone (map on bulletin board).

2. Students are allowed in the laboratory only when the instructor is present.

3. Open drinks and food are not allowed near the lab benches.

4. Report any broken equipment or defective parts to the lab instructor. Do not open, remove the cover, or attempt to repair any equipment.

5. When the lab exercise is over, all instruments, except computers, must be turned off. Return all equipment to the designated location. Your lab grade will be affected if your laboratory station is not tidy when you leave.

6. University property must not be taken from the laboratory.

7. Do not move instruments from one lab station to another lab station.

8. Do not tamper with or remove security straps, locks, or other security devices. Do not disable or attempt to defeat the security camera.

9. When touching the Experimenter Development boards please do not touch the solid-state parts on the board but handle the board from its edge.

10. **ANYONE VIOLATING ANY RULES OR REGULATIONS MAY BE DENIED ACCESS TO THESE FACILITIES**.

I have read and understand these rules and procedures. I agree to abide by these rules and procedures at all times while using these facilities. I understand that failure to follow these rules and procedures will result in my immediate dismissal from the laboratory and additional disciplinary action may be taken.

_____ _____

Signature Date Lab #

Laboratory Safety Information

Introduction

The danger of injury or death from electrical shock, fire, or explosion is present while conducting experiments in this laboratory. To work safely, it is important that you understand the prudent practices necessary to minimize the risks and what to do if there is an accident.

Electrical Shock

Avoid contact with conductors in energized electrical circuits. The typical can not let-go (the current in which a person can not let go) current is about 6-30 ma (OSHA). Muscle contractions can prevent the person from moving away the energized circuit. Possible death can occur as low 50 ma. For a person that is wet the body resistance can be as low as 1000 ohms. A voltage of 50 volts can result in death.

Do not touch someone who is being shocked while still in contact with the electrical conductor or you may also be electrocuted. Instead, press the Emergency Disconnect (red button located near the door to the laboratory). This shuts off all power, except the lights.

Make sure your hands are dry. The resistance of dry, unbroken skin is relatively high and thus reduces the risk of shock. Skin that is broken, wet, or damp with sweat has a low resistance.

When working with an energized circuit, work with only your right hand, keeping your left hand away from all conductive material. This reduces the likelihood of an accident that results in current passing through your heart.

Be cautious of rings, watches, and necklaces. Skin beneath a ring or watch is damp, lowering the skin resistance. Shoes covering the feet are much safer than sandals.

If the victim isn't breathing, find someone certified in CPR. Be quick! Some of the staff in the Department Office are certified in CPR. If the victim is unconscious or needs an ambulance, contact the Department Office for help or call 911. If able, the victim should go to the Student Health Services for examination and treatment.

Fire

Transistors and other components can become extremely hot and cause severe burns if touched. If resistors or other components on your proto-board catch fire, turn off the power supply and notify the instructor. If electronic instruments catch fire, press the Emergency Disconnect (red button). These small electrical fires extinguish quickly after the power is shut off. Avoid using fire extinguishers on electronic instruments.

Explosions

When using electrolytic / tantalum capacitors, be careful to observe proper polarity and do not exceed the voltage rating. Electrolytic and tantalum capacitors can explode and cause injury. A first aid kit is located on the wall near the door. Proceed to Student Health Services, if needed.

INTRODUCTION

When in Doubt, Read This

Laboratory experiments supplement class lectures by providing exercises in analysis, design and realization. The objective of the laboratory is to present concepts and techniques in designing, realizing, debugging, and documenting embedded system applications. The laboratory begins with a review of the Code Composer Studio (CCS) development system, which will be use extensively during the laboratory. Experiment #1 introduces the student to the fundamentals of CCS and its tool set such as the source, application and the debugger windows. In Experiment #2, the basic operations of the MSP430FG4618 Universal Asynchronous Receiver and Transmitter (UART) are discussed. Experiments #3 uses the UART develop basic input output routines such as ASCII to hexadecimal and ASCII to binary. Experiment #4 introduces the student to the LCD display used on the experimenter board while Experiment #5 uses the LCD display to display the output from hexadecimal addition, subtraction and multiplication. Experiment #6 discusses the use of interrupt on the MSP403FG4618 and how to write an interrupt service routine. Experiment #7 illustrates the use of the analog-to-digital converter as a method of reading external analog voltages applied to the MSP430FG4618.

This laboratory requires that each student obtain a copy of this manual and a downloaded version of CCS. The student can use CCS on the laboratory computers or the student can go to http://processors.wiki.ti.com/index.php/Download_CCS to download the CCS development tools.

The student is to prepare for each laboratory by reading the assigned topics. The student is expected to right a draft version of the programs required as directed by the pre-laboratory preparation assignments or by the laboratory instructor. Depending on the laboratory assignment, the pre-laboratory preparation may be due at the beginning of the laboratory period or may be completed during the assigned laboratory period. Be informed that during each laboratory period the instructor will grade your pre-laboratory preparation.

During the laboratory period you are expected to construct and complete each laboratory assignment, fully comment each program written and discuss in detail the steps involved in debugging the required programs. Also the expected results, equipment used, laboratory partners, and design changes should be recorded in the laboratory notebook. A laboratory performance grade will be assigned by the laboratory instructor upon successful completion of the above-described tasks for each experiment. Writing computer programs involves three important phases. The first phase is to enter into the development environment the source code. This is the easiest of the three phases. The next phase is to be able to compile or assemble the program with no syntax errors. The

hardest of the three phases is the last phase debugging a program so that it works logically as expected and meets the design requirements. As such, it is expected that each student will write their own programs, remove all syntax errors, and develop the skills (also become familiar with the various debugging tools) required to properly debug a program.

Each student will be assigned to a laboratory station which has a computer and an MSP430FG4618 experimenter board. Each student is responsible for his or her own work including software program development and documentation. A Laboratory Report, following the guidelines presented in this handout is due the laboratory period following the completion of the in-laboratory work or when the instructor designates. A numeric grade will be assigned using the grading policy given below. Laboratory reports will be due before the start of each laboratory. No credit will be given for laboratory reports that are late. However, a student must complete each assigned experiment in order to complete the laboratory. By not turning in a laboratory report, a student will receive an incomplete for that report, which could result in an incomplete for the laboratory grade.

Students who miss the laboratory lecture due to a pre-approved university activity should make arrangements with the laboratory instructor to make up the laboratory at a later time. Students who are late to their laboratory section will not receive credit for the pre-laboratory portion of the laboratory experiment.

EEL 4742 Lab Grading Policy Page

1. Students must will follow the LABORATORY FORMAT for submission of the laboratory report.
2. Each student must submit one REPORT per laboratory experiment.
3. Each laboratory will be completed in 1, 2, or 3 weeks. The EXACT DURATION of a lab will be decided by the laboratory instructor prior to the laboratory experiment.
4. Students can NOT skip the laboratory periods. SPECIAL PERMISSION must be obtained from the LABORATORY INSTRUCTOR to do so. Otherwise, grade points will be DEDUCTED from the student's total grade earned for the scheduled laboratory experiment.
5. Each lab will be graded out of 100.
6. The deadline for submission of lab reports will be decided by the lab instructor.
7. Each lab report must be submitted before or on the scheduled deadline date. No late laboratory reports will be accepted.

If required, Lab policy will be modified and will depend solely on the lab instructor.

LABORATORY REPORT FORMAT:

- **Cover Page:** The laboratory report should include a cover page giving the experiment number, your name, and date should be at the top right hand side of each page. It should also include the following: **Heading:** The experiment number, your name, and date should be at the top right of each page.

EEL 4742 LABORATORY EXPERIMENT #

Section: _____
Points: _____

Name: _____

- **Heading:** The experiment number, your name, and date should be at the top right hand side of each page.
- **Objective:** A brief but complete statement of what you intend to design or verify in the experiment should be at the beginning of each experiment.
- **Apparatus List:** List the items of equipment, with identification numbers using the UCF label tag, make, and model of the equipment. It may be necessary later to locate specific items of equipment for rechecks if discrepancies develop in the results. Also include the computer used and the version number of any software used.
- **Procedure and/or Design Methodology:** In general, lengthy explanations are unnecessary. Be brief. Keep in mind the fact that the experiment must be reproducible from the information given in your report. Include the steps taken in the design of the computer programs written to implement the various laboratory experiments.
- **Design Specification Plan:** A detailed discussion on how your design approach meets the requirements of the laboratory experiment should be presented. Given a set of requirements there are many ways to write a computer program that meets these requirements. The Design Specification Plan describes the methodology chosen and the reason for the selection (why). The Design Specification Plan is also used to verify that all the requirements of the project have been implemented as described by the requirements.

- **Test Plan:** A test plan describes how to test the implemented software against the given requirement specifications. This plan gives detailed steps during the test process defining which inputs should be tested and verifying for these inputs that the correct outputs appear from the software program.

- **Source Code:** Fully commented source code for both C language and assembly language programs must be included in the laboratory report. The comments used in these programs should be clear and easy to reader and explain how the program works.

- **Screen shot of output:** Where possible screen shot outputs (ALT-PRINT SCREEN) from the desktop computer display should be included in the laboratory report.

- **Conclusion:** This is your interpretation of the objectives, procedures, and results of the experiment, which will be used as a statement of what you learned in performing the experiment. This is not a summary. Be brief and specific but complete. Identify the advantages and/or disadvantages of your solution in design-oriented experiments.

EXPERIMENT #1

INTRODUCTION TO CODE COMPOSER STUDIO

Goals: To introduce the Code Composer Studio tool set that is used to write C language and assembly language programs for the MSP430. In particular, the experimenter board which contains the MSP430FG4618 will be used.

References: MSP430x4xx Family User's Guide, MSP430 Assembly Language Tools v 4.0 User's Guide, MSP430FG4618/F2013 Experimenter's Board, the help menu in code composer studio, and MSP430 Microcontroller Basics by John H. Davies.

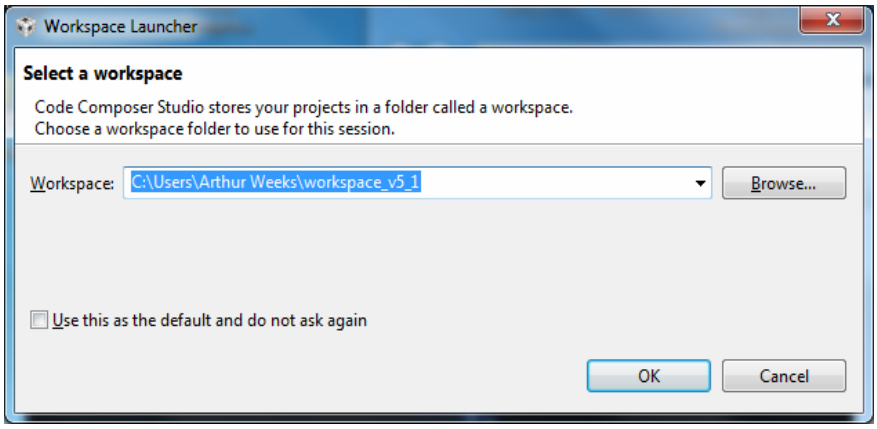
Equipment: A desktop computer system running code composer, the MSP430FG4618 experimenter board and the MSP430 USB-Debug interface (MSP-FET430UIF).

Pre-laboratory: Read this experiment carefully to become familiar with the procedural steps in this experiment.

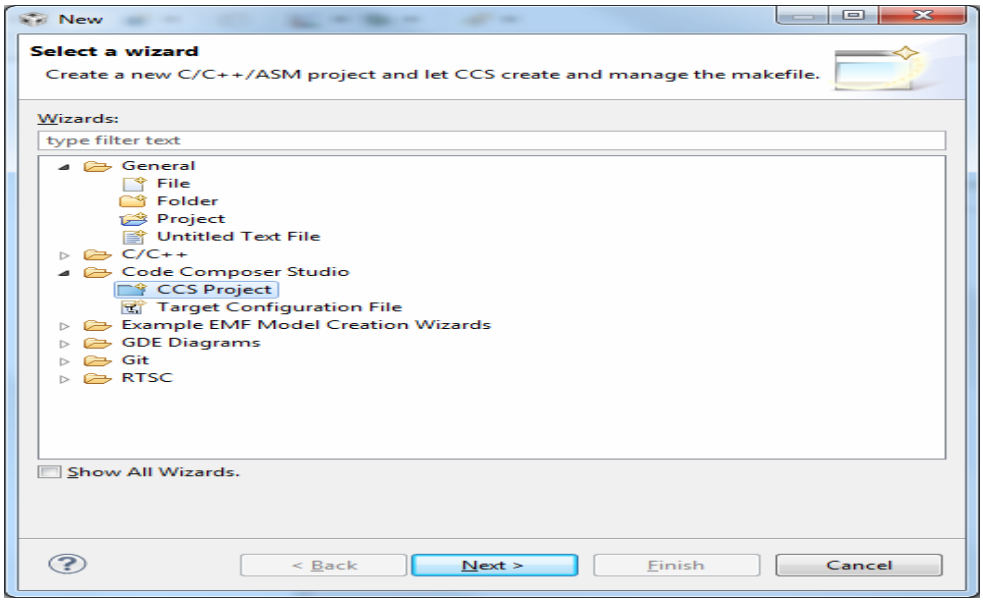
Discussion: This laboratory experiment will give the step by step procedures on creating C language and assembly language programs in Code Composer Studio that are executed on the MSP430 experimenter board.

Procedure: Part 1. Introduction to Code Composer Studio in generating a C language program

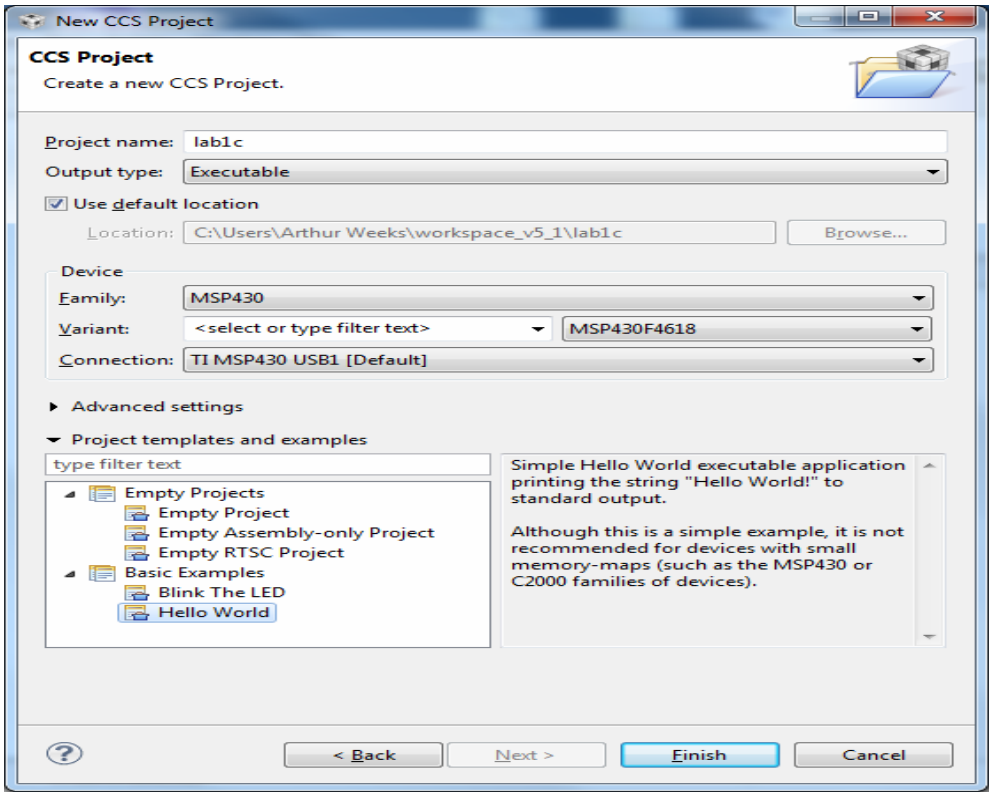
1. Double click on the CCS icon and start execution of code composer studio. The default location of where the projects are stored is given and can be changed to another location. Use the default location given.



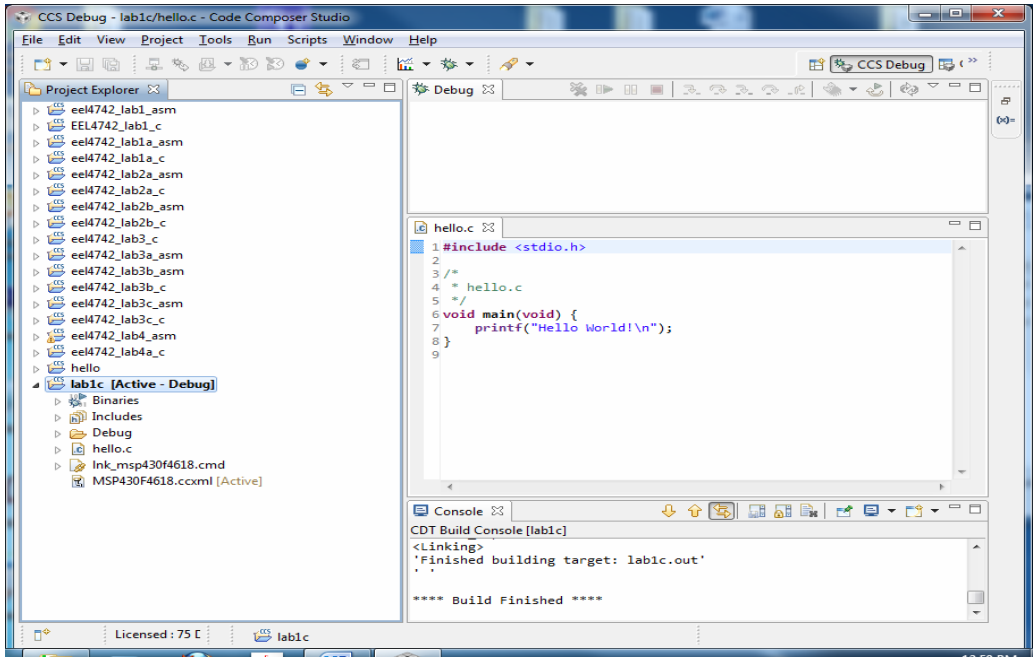
2. To create a new project, select from the File menu, Project and then Other (File -> project -> other). Next, select CCS Project.



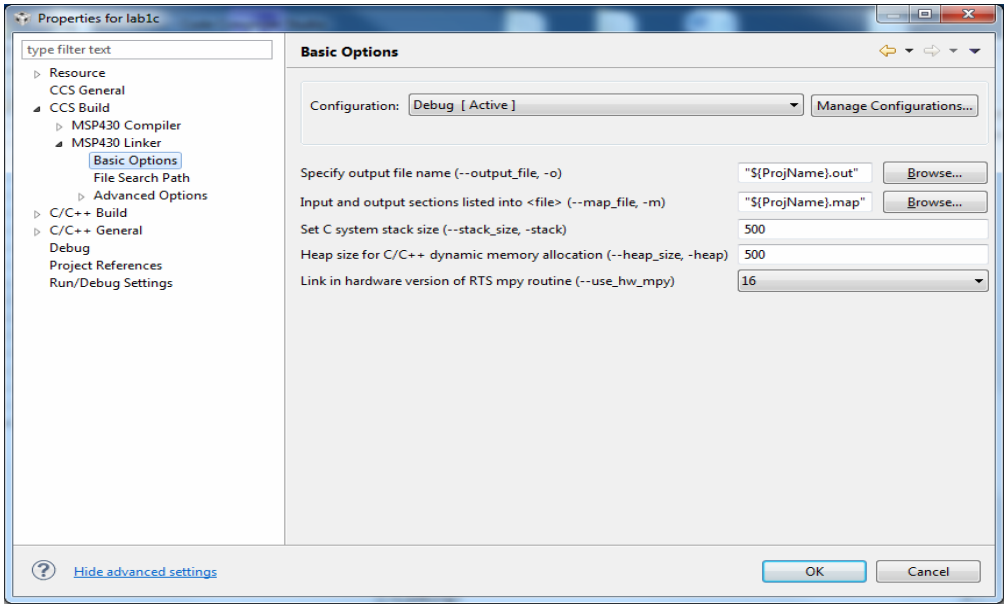
3. Select the desired project name and the desired version of the MSP430. For the experimenter board it's the MSP430FG4618. Also select the type of project: Empty C program, a Hello World C program, or an Assembly Language program. For this step, choose the Hello World program



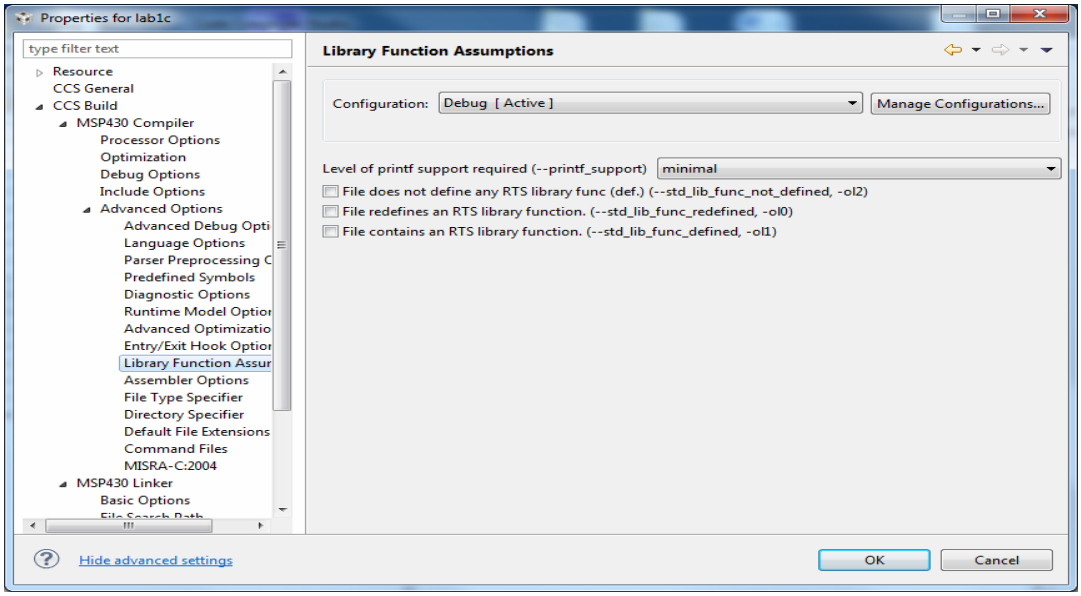
4. Highlighting the project makes it the active – debug project. Also, clicking on the left arrow opens all of the subdirectories and files associated with the project. Selecting main.c will open the C language source file.



5. To use the printf function in C, the reserved number of bytes for the stack and the heap must be increased from 80 to a minimum of 500 bytes. Open under the Project menu the Project Properties and under Basic Linker Options change the heap and stack memory size to 500.



The default printf option is set to minimum which only allows strings to be printed. To print integer or floating point variables this option must be changed and it is located under MSP430 Compiler Advanced options. The options are minimum support, no floating variable support, or full support. Full support uses the most program memory while minimum uses the least program memory.



5. To build and compiler a project, select the Project menu and then the Build Project option. CCS will then build the project and indicate if there are any errors that need to be corrected.

6. To debug a project, the debug option must be selected under the Run menu and the experimenter board must be connected to the desktop. The Build Project and Debug options should be selected every time the program source code changes.

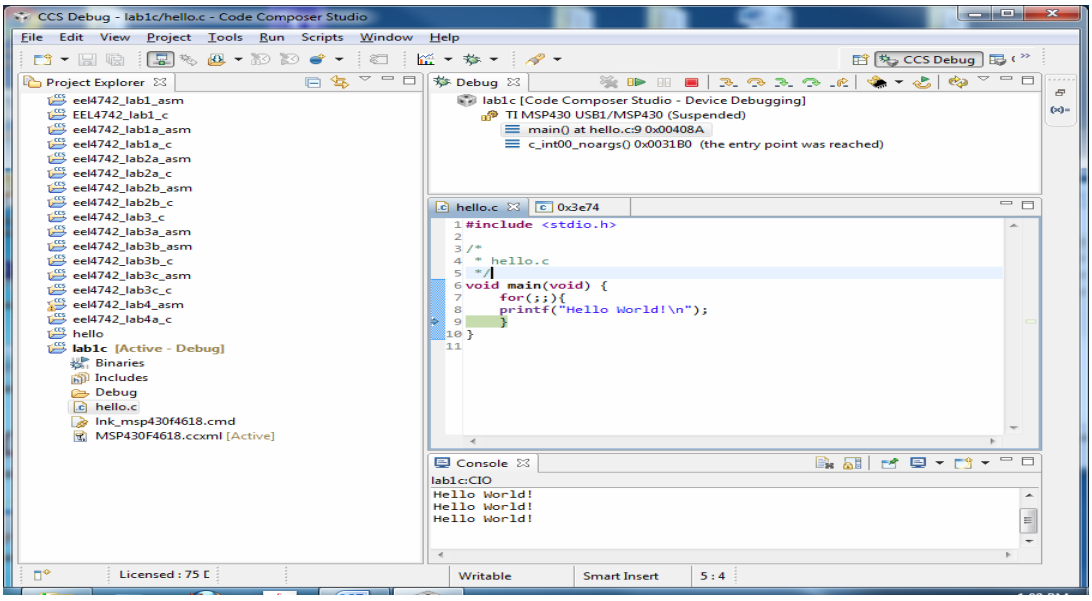
7. To execute the program, the Resume option under the Run menu is selected. The console window should now display the "Hello World" message.

8. To quit the program, the Terminate option under the Run menu is selected.

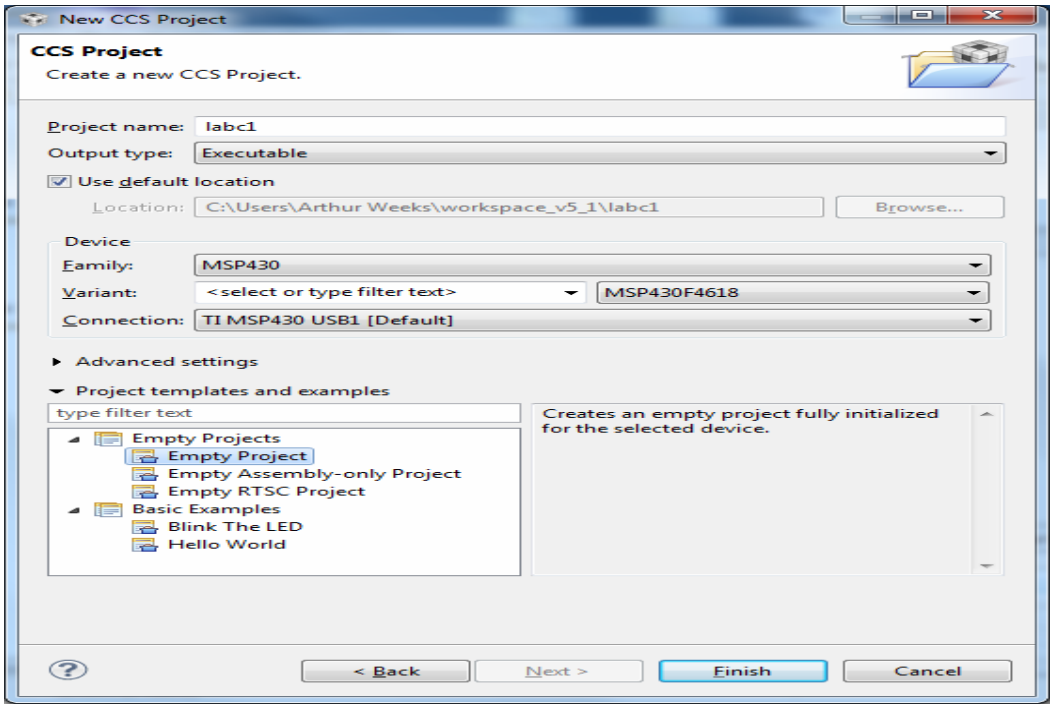
9. Also available for debugging is the ability of stepping through the program one line at a time. F5 (Step Into) executes the program one line at a time. F6 (Step Over) executes the various functions and subroutines but does not trace them one line at a time. The F5 option traces even subroutines and functions.

10. Add the following for(;;) to allow the program to run indefinitely. Build the project, debug the project, and then trace then program one line at a time using the Step Into

option (F5). Note how the program continuously loops printing the "Hello World Message"



11. Next, create a new project that has an empty file. This creates an empty C program template. Make sure to click on this project name in the Project Explorer windows so this project is the Active-Debug project.



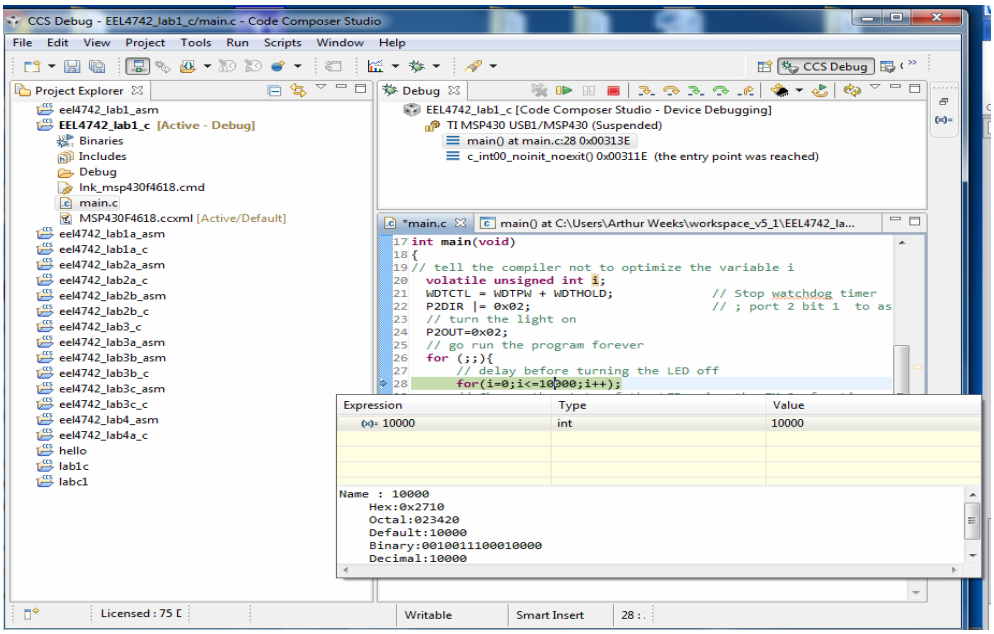
12. Next, type the following C program into the blank main.c file under the desired the Active-Debug project. Read the comments contained in this program, as these comments cover in detail the purpose of each line of code. The definitions for each variable is defined in the include file for the MSP430FG4618 #include "msp430fg4618.h". The notation for the MSP430 for a port is PX.Y where X is the port number and Y is the bit. Port 2 bit 2 is the green LED and Port 2 bit 1 is the yellow LED. All ports also have three variables associated with the port: the direction control PXDIR, the input PXIN, and the output PXOUT. For a port bit to be an output, its value must be 1 in the direction control variable. If it's a zero, then the bit is an input pin.

To set a bit to a one on a port, the program sets this bit in PXOUT to a one. To read an input bit on a port, the PXIN variable is used. For example, to read the status of the two push buttons on the experimenter board, (input buttons on port 1 bits SW1 = bit 0 and SW2 = bit 1), the variable P1IN is used. A value of 1 indicates that the appropriate switch is not pressed and a value of zero indicates the switch is pressed.

```
//*******************************************************************
//    LED turn on to blink the LED on Port 2 of the MSP430FG4618
//    experimenter board RAM at 0x1100 - 0x30ff, FLASH at 0x3100 -
//    0xfbff
//    Port 2 is used for the LED's Port 2 bit 2 is the green LED,
//    Port 2 bit 1 is the yellow LED
//    input buttons on port 1 bits SW1 = bit 0 and SW2 = Bit2
//    1 = SW not pressed 0 = pressed
//*******************************************************************
//-------------------------------------------------------------------
//    must include the C header to get the predefined variable names
//    used by the MSP430FG4618 on the experimenter board
//-------------------------------------------------------------------
#include "msp430fg4618.h"

int main(void)
{
//  tell the compiler not to optimize the variable I, otherwise the
//  compiler may change how the variable is used
  volatile unsigned int i;
  WDTCTL = WDTPW + WDTHOLD;          // Stop watchdog timer so the
       // program
                                     // runs indefinitely
  P2DIR |= 0x02;                     // Set port 2 bit 1 to as an
                                     // output 1 = output 0 = input
  // turn the light on
  P2OUT=0x02;
  // go run the program forever
  for (;;){
       // delay before turning changing the state of the LED
       for(i=0;i<=10000;i++);
       // Change the state of the LED using the EX-OR function
       P2OUT = P2OUT ^ 0x02;
       }
}
```

Execute the blinking LED program using the Resume menu option under the Run menu. Take note that the LED is blinking on the experimenter board. Terminate the program and rerun the program using the step into menu option (F5) executing one line at a time. Once the program is at the line that the for loop is on, highlight the variable i with the

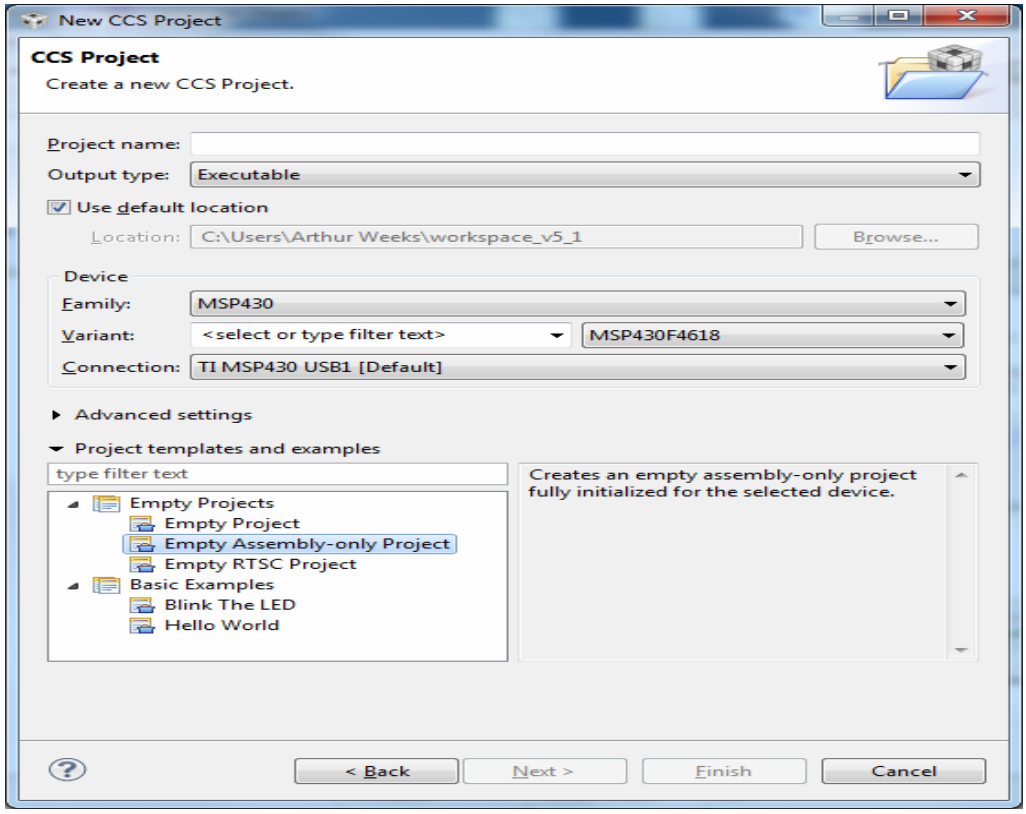
mouse and observe the value of i indicated. Every time the program executes this line the value of i is incremented.

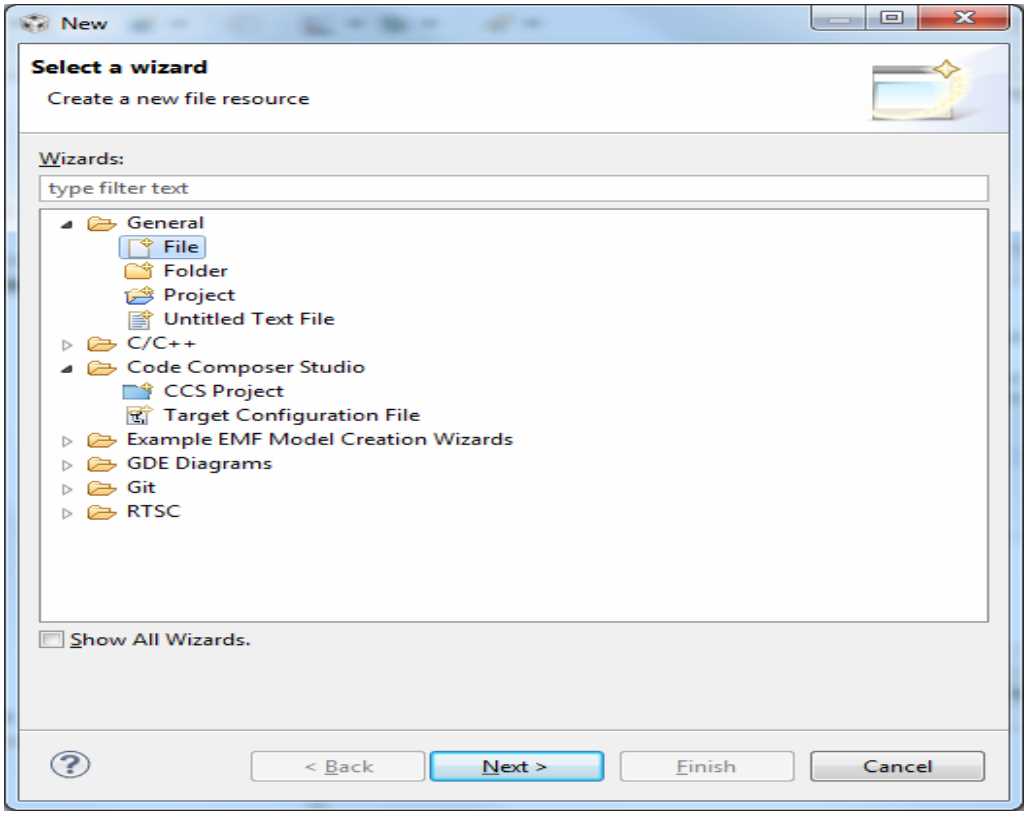


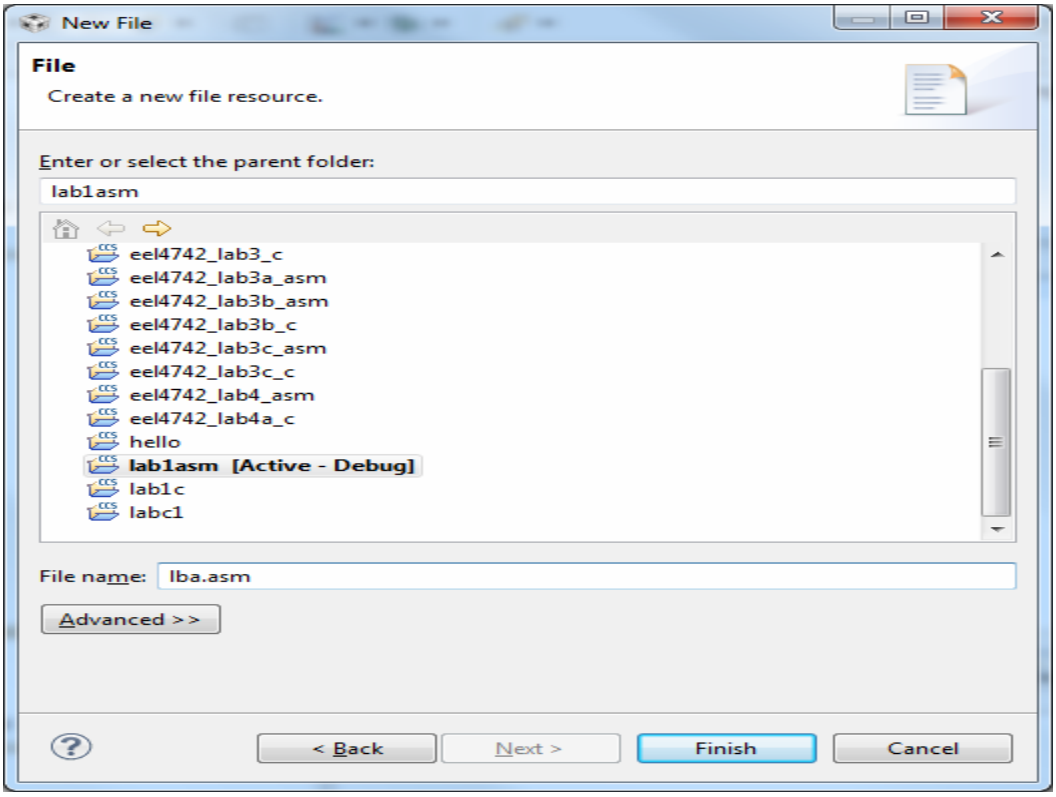
Procedure: Part 2. Introduction to Code Composer Studio in generating an assembly language program.

1. To create an assembly language program, the same steps are used as creating a C-language program. But for an assembly language program, an empty assembly-only-project is selected.

2. Next, under the File menu, select New and then Other. Under the General option, select the File option, and then the Next button. Choose a name for the blank assemble language project. Also make sure the desired project name for the assembly language file is selected as the parent folder. Make sure that the file extension is *.asm so the CCS knows that this is an assembly language file. Finally, click the finished button.


New CCS Project
CCS Project
Create a new CCS Project.
Project name:
Output type:
Executable
Use default location
Location:
C:\Users\Arthur Weeks\workspace_v5_1
Browse...
Device
Family:
MSP430
Variant:
<select or type filter text>
MSP430F4618
Connection:
TI MSP430 USB1 [Default]
Advanced settings
Project templates and examples
type filter text
Empty Projects
Empty Project
Empty Assembly-only Project
Empty RTSC Project
Basic Examples
Blink The LED
Hello World
Creates an empty assembly-only project fully initialized for the selected device.
< Back
Next >
Finish
Cancel


New
Select a wizard
Create a new file resource
Wizards:
type filter text
General
File
Folder
Project
Untitled Text File
C/C++
Code Composer Studio
CCS Project
Target Configuration File
Example EMF Model Creation Wizards
GDE Diagrams
Git
RTSC
Show All Wizards.
< Back
Next >
Finish
Cancel

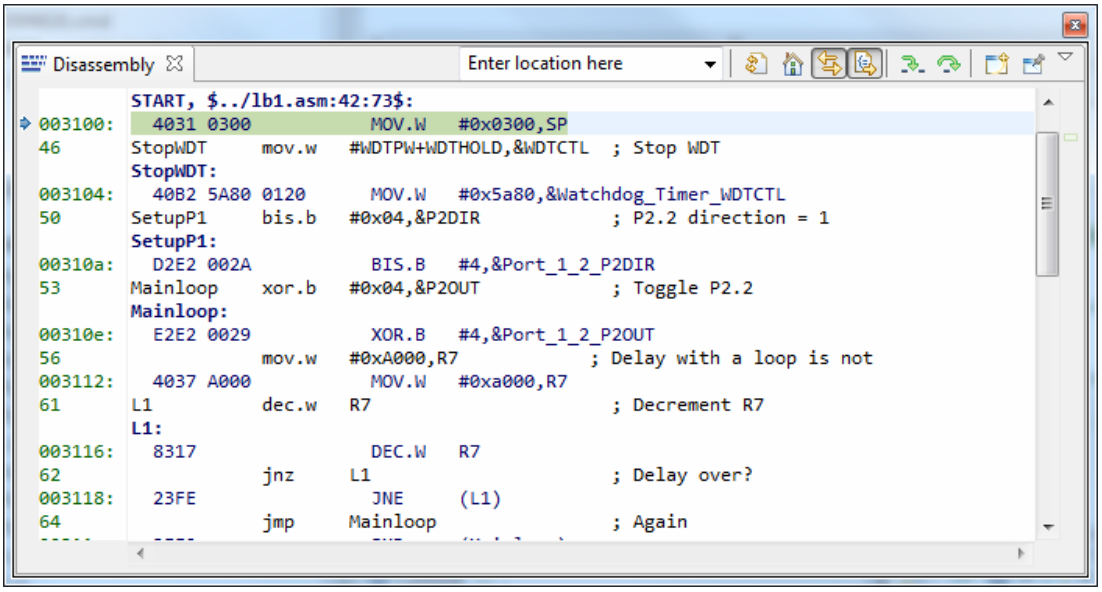


3. Type in the folowing assembly language program to make the LED lights flash. Build the project as done for a C-language project by selecting under the Project menu the Build Project option (PROJECT-> Build project). Read the comments contained in this program as these comments cover in detail the purpose of each line of code. The definitions for each variable is defined in the include file for the MSP430FG4618 #include "msp430fg4618.h". The notation for the MSP430 for a port is PX.Y where X is the port number and Y is the bit. Port 2 bit 2 is the green LED and Port 2 bit 1 is the yellow LED. All ports also have three variables associated with the port: the direction control PXDIR, the input PXIN, and the output PXOUT. For a port bit to be an output its value must be 1 in the direction control variable. If it's a zero, then the bit is an input pin.

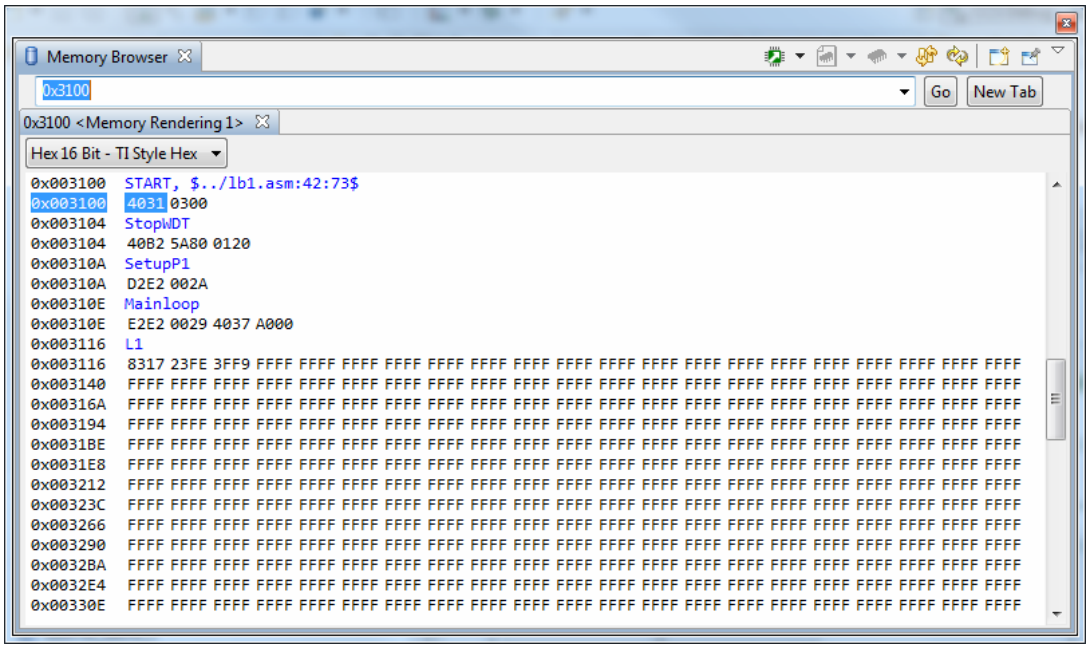
```
;*-------------------------------------------------------------------
;      semi-colon = comments
;      LED BLINK program in Assembly
;      experimenter board RAM at 0x1100 - 0x30ff, FLASH at 0x3100 - ;
       0xfbff
;      Port 2 is used for the LED's Port 2 bit 2 is the green LED,
;      Port 2 bit 1 is the yellow LED
;
;*-------------------------------------------------------------------
;-------------------------------------------------------------------
;  must include the C header to get the predefined variable names
; .cdecls assembler directive tell the assembler to include C file
;  headers
;-------------------------------------------------------------------

            .cdecls C,LIST,"msp430fg4618.h"  ; cdecls tells assembler ;
                                             ; to allow
                                             ; the c header file
```

```
;-------------------------------------------------------------------------
;   Main Code
;-------------------------------------------------------------------------
;  Need to tell the assembler where to put the assembly code  Since
;  there is FLASH, and RAM on the MSP430 need to tell the assembler
;  where constants, variable and data goes.  The .text directive tell
;  the assembler what follows is assembly instructions and to be placed
;  in program flash.  The .sect directive can also be used to put
;  variables in system memory or RAM (.sect ".sysmem").  The
;  definitions of the various sections of memory is given in *.CMD
;  file included with the project.  RAM in the MSP430FG4618 begins at
;  0x1100 and program flash begins at 0x3100.  The system RAM is also
;  mirrored starting 0x200.  For example, 0x300 is the same as 0x1200.


            .text                           ; program start
;  Tell the assembler that the label name START is a global label.
;  Please note the underscore in front of the laber START.  This label
;  name must match the label name of the first line of code.

            .global _START                  ; define entry point

;-------------------------------------------------------------------------
;  Must initialize the stack pointer to RAM
START       mov.w   #0x300,SP               ; Initialize '0x1200 or
        ; 0x300 stackpointer

;  Turn off the watchdog time so the program can run indefinitely
StopWDT     mov.w   #WDTPW+WDTHOLD,&WDTCTL  ; Stop WDT

;  Set port 2 bit 2 direction so thatP2.2 is an output 1 = output
;  0 = input
SetupP1     bis.b   #0x04,&P2DIR            ; P2.2 direction = 1

;  Set port P2.2 to a one to turn on the LED
Mainloop    xor.b   #0x04,&P2OUT            ; Toggle P2.2

;  Move the value of 0xA000 into register 7 so to create a delay
              mov.w   #0xA000,R7            ; Delay with a loop is not
                                            ; the best way; interrupts
                                            ; are better Given in Lab 6
;  Decrement register 7 until it's zero.  Stay in this loop until
;  register 7 is zero
L1          dec.w   R7                      ; Decrement R7
            jnz     L1                      ; Delay over?
;  Let's run the program forever
            jmp     Mainloop                ; Again


                                            ;
;-------------------------------------------------------------------------
;           Interrupt Vectors
;-------------------------------------------------------------------------
;  Need to load the MSP430fg4618 reset vector with the address of the
;  location of the first line of assembly instructions to be executed.
;  The .sect assembler directive does this

            .sect   ".reset"                ; MSP430 RESET Vector
            .short  START
                ;
;  The end assembler directive tell the assembler end of source code to
;  be assembled
            .end
```

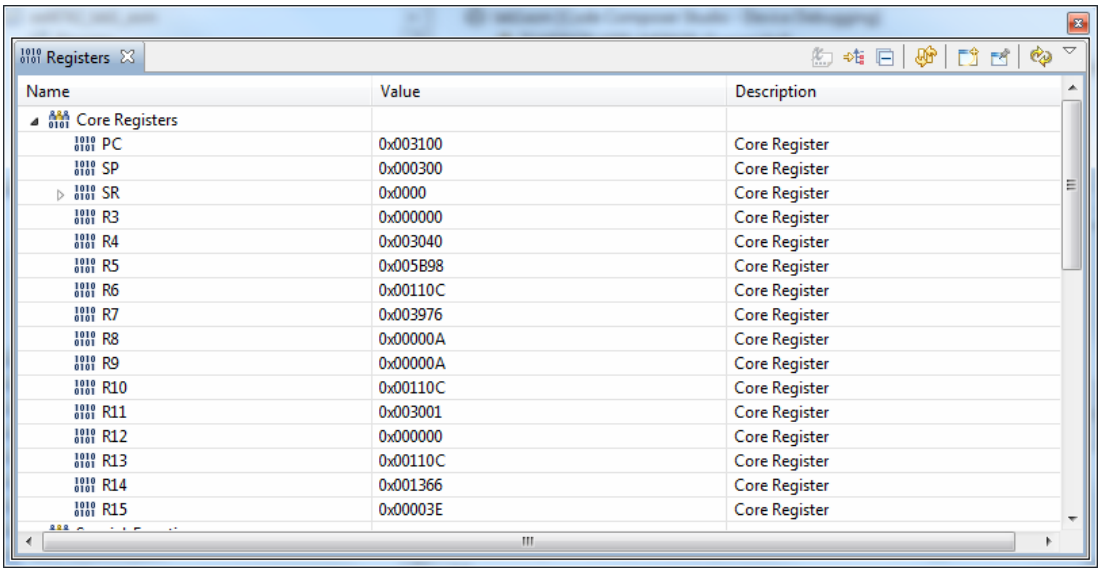
4. Next, debug this project using the Debug option under the Run menu. Finally, execute this project using Resume option under the RUN menu. As with the C language program the Step-Into (F5) and the Step-Over (F6) options can be used to execute one line of instructions at a time. To help in the debugging of an assembly language program, there are several windows available in CCS. These windows are available under the VIEW menu. The first is the Disassembly window. This window gives the assembled code listing file. It gives the assembly language instruction along with the machine code generated for each instruction and the location in program flash memory where the instructions are located.



The next window that is commonly used in debugging an assembly language program is the Memory Browser window. The user must enter a memory address to view. For example, if 0x3100 is chosen, the memory browser window displays the machine code of the assembly instructions.



The last window that is used for debugging a program is the Register window. This window allows the user to view the MSP430 registers and the various states of the MSP430 I/O devices. Clicking on the Core Register arrow gives the MSP430 registers.



5. Modify the assembly language program to load a value of 0x0005 into register R7. Build the project and then debug this project. Next, trace through each line of the code using the Step Into option observing the register values as each line of code is executed. In particular, take note how R7 starts with a value 0x0005 and decreases in value until it reaches zero.

Procedure: Part 3. Additional Programs to be written and executed.

1. Modify and execute the C-language program so that both the green and yellow LEDs blink alternatively on and off about once every second. For one second, the green LED is on and the yellow LED is off. For the next second, the green LED is off and the yellow LED is on. Adjust the value used in the delay loop to obtain the correct timing. Make sure that this program is properly commented. An additional for loop may be needed to obtain the desired timing. Make sure to have the lab assistant verify the functionality of this program.

2. Repeat step 1, but this time modify and execute the assembly language program. An additional loop may be needed to obtain the desired timing for second on and one second off.

3. Write a C-language program that turns on the green LED when SW1 on the experimenter board is pressed and the yellow LED when SW2 is pressed. Execute this program and verify that it is running correctly. Make sure that this program is properly commented. Make sure to have the lab assistant verify the functionality of this program.

4. Repeat step 3, but this time write an assembly language program to turn on and off the green and yellow LED's.

Report

1. Include in the laboratory report the objective of the laboratory experiment.

2. The procedure used to generate and execute the C-language and an assembly language programs (To be written in your own words not cut paste from the project).

3. The source code for the C-language and assembly language programs in Part 3 (4 programs).

4. A summary of the laboratory experiment and what you have learned.

EXPERIMENT #2

SERIAL COMMUNICATION USING THE MSP430 UART

Goals: To develop C language and assembly language programs that provides serial communications. In particular, the experimenter board which contains the MSP430FG4618 and the on board Universal Asynchronous Receiver and Transmitter (UART) will be used.

References: MSP430x4xx Family User's Guide (Chapter 19: pages 19-1 through 19-36), MSP430 Assembly Language Tools v 4.0 User's Guide, MSP430FG4618/F2013 Experimenter's Board, the help menu in code composer studio, and MSP430 Microcontroller Basics by John H. Davies pages (Chapter 10: 574-590).

Equipment: A desktop computer system running code composer, the MSP430FG4618 experimenter board and the MSP430 USB-Debug interface (MSP-FET430UIF).

Pre-laboratory: Read this experiment carefully to become familiar with the procedural steps in this experiment. Study in detail and become familiar with the C program and the assembly program provided with this lab. Develop an approach to implement the C language and assembly language programs required in Parts 1 and 2 of the procedures.

Discussion: The laboratory experiment will give the step by step procedures on creating C language and assembly language programs that provide serial communications using the MSP430 on board UART.

Part 1. Generating a C language program that provides serial communications.

1. Serial communications (also known as RS232C) is an asynchronous communication protocol that transmits 1 bit at a time. The C in the RS232C standard refers to this communication protocol as character mode. Typically, RS232C is used to transmit and receive ASCII data. Associated with each data set is one start bit and one or two stop bits. The number of bits transmitted can be configured to 7 or 8 bits. In addition, parity checking can be included and can be set either to even, odd parity, or no parity. Also, the order of the data being sent is such that the least significant bit is transmitted first.

2. The rate in which the serial stream of bits are transmitted is known as the baud rate with the following being the most common baud rates: 1200, 2400, 4800, 9600, 19200, 38400, and 115K.

3. The requirement for RS232C serial communications is that the transmitting and receiving computer be configured with the same setting / protocol. As such, there are nine registers in the MSP430fg4618 that must be configured. These registers are UCA0CTL0 Control Register 0), UCA0CTL1 (Control Register 1), UCA0BR0 (Baud Rate Register 0), UCA0BR1 (Baud Rate Register 1), UCA0MCTL (Modulation Control Register), UCA0STAT (Status Register) UCA0RXBUF (Receive data Buffer), UCATXBUF (Transmit Data Buffer) and PE2SEL (Select the appropriate pin for transmit and receive). Each of these registers is discussed in Chapter 19 of the MSP430x4xx Family User Guide and is highlighted below.

4. UCA0CTL0 register: The configuration for this laboratory experiment requires no parity, least significant bit first (LSB), 8 data, 1 stop, UART mode and asynchronous mode. The value that must be loaded into the UCA0CTL0 register is 0x00.

UCAxCTL0, USCI_Ax Control Register 0

7	6	5	4	3	2	1	0
UCPEN	UCPAR	UCMSB	UC7BIT	UCSPB	UCMODEx		UCSYNC=0
rw–0	rw–0	rw–0	rw–0	rw–0	rw–0	rw–0	rw–0

UCPEN	Bit 7	Parity enable 0 Parity disabled. 1 Parity enabled. Parity bit is generated (UCAxTXD) and expected (UCAxRXD). In address-bit multiprocessor mode, the address bit is included in the parity calculation.
UCPAR	Bit 6	Parity select. UCPAR is not used when parity is disabled. 0 Odd parity 1 Even parity
UCMSB	Bit 5	MSB first select. Controls the direction of the receive and transmit shift register. 0 LSB first 1 MSB first
UC7BIT	Bit 4	Character length. Selects 7-bit or 8-bit character length. 0 8-bit data 1 7-bit data
UCSPB	Bit 3	Stop bit select. Number of stop bits. 0 One stop bit 1 Two stop bits
UCMODEx	Bits 2–1	USCI mode. The UCMODEx bits select the asynchronous mode when UCSYNC = 0. 00 UART Mode. 01 Idle-Line Multiprocessor Mode. 10 Address-Bit Multiprocessor Mode. 11 UART Mode with automatic baud rate detection.
UCSYNC	Bit 0	Synchronous mode enable 0 Asynchronous mode 1 Synchronous Mode

5. UCA0CTL1 register: The UCA0CTL1 register sets the clock source to determine the baud rate for the UART. For this project the baud rate clock source will be set to the external Aclock (ACLK) which is set to 32768 Hz by an external crystal. The receiver erroneous character interrupt flag is to be rejected and the special received break characters interrupt flag is to be set to zero. The dormant state is cleared so that all characters are received and the next frame is data not address field. UCT0BRK is cleared so that the next frame transmitted is not a break. Finally, the UART is put into a reset mode until the baud rate is set. Finally the UART is temporarily placed into reset state (UCSWRST = 1) until all of the UART registers have been set. This configuration requires that 0x41 be written into UCA0CTL1.

UCAxCTL1, USCI_Ax Control Register 1

7	6	5	4	3	2	1	0
UCSSELx		UCRXEIE	UCBRKIE	UCDORM	UCTXADDR	UCTXBRK	UCSWRST
rw–0	rw–0	rw–0	rw–0	rw–0	rw–0	rw–0	rw–1

UCSSELx	Bits 7-6	USCI clock source select. These bits select the BRCLK source clock. 00 UCLK 01 ACLK 10 SMCLK 11 SMCLK
UCRXEIE	Bit 5	Receive erroneous-character interrupt-enable 0 Erroneous characters rejected and UCAxRXIFG is not set 1 Erroneous characters received will set UCAxRXIFG
UCBRKIE	Bit 4	Receive break character interrupt-enable 0 Received break characters do not set UCAxRXIFG. 1 Received break characters set UCAxRXIFG.
UCDORM	Bit 3	Dormant. Puts USCI into sleep mode. 0 Not dormant. All received characters will set UCAxRXIFG. 1 Dormant. Only characters that are preceded by an idle-line or with address bit set will set UCAxRXIFG. In UART mode with automatic baud rate detection only the combination of a break and synch field will set UCAxRXIFG.
UCTXADDR	Bit 2	Transmit address. Next frame to be transmitted will be marked as address depending on the selected multiprocessor mode. 0 Next frame transmitted is data 1 Next frame transmitted is an address
UCTXBRK	Bit 1	Transmit break. Transmits a break with the next write to the transmit buffer. In UART mode with automatic baud rate detection 055h must be written into UCAxTXBUF to generate the required break/synch fields. Otherwise 0h must be written into the transmit buffer. 0 Next frame transmitted is not a break 1 Next frame transmitted is a break or a break/synch
UCSWRST	Bit 0	Software reset enable 0 Disabled. USCI reset released for operation. 1 Enabled. USCI logic held in reset state.

6. UCA0BR0, UCA0BR1 and UCA0MCTL registers: UCA0BR0 and UCA0BR1 set the baud rate by dividing the selected clock source's frequency by

Baud rate = (Selected Clock Source's Frequency) / (UCABR0 + UCA0BR1 * 256).

Since the clock source is the ACLK and its frequency is 32768 Hz, for a desired baud rate of 9600 yields a baud rate divisor of 3.413. The closest integer value is 3. This requirement sets UCA0BR0 to 0x03 and UCA0BR1 to 0x00. For this baud rate, the MSP430's UART requires a modulation pattern set to 0x06 (UCA0MCTL = 0x06). The MSP430x4xx Family User's Guide gives more information on how to set the modulation register.

UCAxBR0, USCI_Ax Baud Rate Control Register 0

7	6	5	4	3	2	1	0
UCBRx – low byte							
rw	rw	rw	rw	rw	rw	rw	rw

UCAxBR1, USCI_Ax Baud Rate Control Register 1

7	6	5	4	3	2	1	0
UCBRx – high byte							
rw	rw	rw	rw	rw	rw	rw	rw

UCBRx	Clock prescaler setting of the Baud rate generator. The 16-bit value of (UCAxBR0 + UCAxBR1 × 256) forms the prescaler value UCBRx.

UCAxMCTL, USCI_Ax Modulation Control Register

7	6	5	4	3	2	1	0
UCBRFx				UCBRSx			UCOS16
rw–0	rw–0	rw–0	rw–0	rw–0	rw–0	rw–0	rw–0

7. UCA0STAT register: For the UCA0STAT register, it is desired to have listen enable set to a disabled mode. Also there is no requirement for error detection for framing, data overrun, or parity. The UART will not use idle line detection or no break conditions. Finally, the bits received by the UART are data not an address. To meet these requirements, a value of 0x00 is loaded in the UCA0STAT register.

UCAxSTAT, USCI_Ax Status Register

7	6	5	4	3	2	1	0
UCLISTEN	UCFE	UCOE	UCPE	UCBRK	UCRXERR	UCADDR UCIDLE	UCBUSY
rw–0	rw–0	rw–0	rw–0	rw–0	rw–0	rw–0	r–0

UCLISTEN	Bit 7	Listen enable. The UCLISTEN bit selects loopback mode. 0 Disabled 1 Enabled. UCAxTXD is internally fed back to the receiver.
UCFE	Bit 6	Framing error flag 0 No error 1 Character received with low stop bit
UCOE	Bit 5	Overrun error flag. This bit is set when a character is transferred into UCAxRXBUF before the previous character was read. UCOE is cleared automatically when UCxRXBUF is read, and must not be cleared by software. Otherwise, it will not function correctly. 0 No error 1 Overrun error occurred
UCPE	Bit 4	Parity error flag. When UCPEN = 0, UCPE is read as 0. 0 No error 1 Character received with parity error
UCBRK	Bit 3	Break detect flag 0 No break condition 1 Break condition occurred
UCRXERR	Bit 2	Receive error flag. This bit indicates a character was received with error(s). When UCRXERR = 1, on or more error flags (UCFE, UCPE, UCOE) is also set. UCRXERR is cleared when UCAxRXBUF is read. 0 No receive errors detected 1 Receive error detected
UCADDR	Bit 1	Address received in address-bit multiprocessor mode. 0 Received character is data 1 Received character is an address
UCIDLE		Idle line detected in idle-line multiprocessor mode. 0 No idle line detected 1 Idle line detected
UCBUSY	Bit 0	USCI busy. This bit indicates if a transmit or receive operation is in progress. 0 USCI inactive 1 USCI transmitting or receiving

8. UCA0RXBUF and UCA0RXBUF registers: UCA0RXBUF register holds the last serial data received by the UART. UCATXBUF register holds the data to be transmitted by the UART.

UCAxRXBUF, USCI_Ax Receive Buffer Register

7	6	5	4	3	2	1	0
UCRXBUFx							
r	r	r	r	r	r	r	r

UCRXBUFx	Bits 7–0	The receive-data buffer is user accessible and contains the last received character from the receive shift register. Reading UCAxRXBUF resets the receive-error bits, the UCADDR or UCIDLE bit, and UCAxRXIFG. In 7-bit data mode, UCAxRXBUF is LSB justified and the MSB is always reset.

UCAxTXBUF, USCI_Ax Transmit Buffer Register

7	6	5	4	3	2	1	0
UCTXBUFx							
rw	rw	rw	rw	rw	rw	rw	rw

UCTXBUFx	Bits 7–0	The transmit data buffer is user accessible and holds the data waiting to be moved into the transmit shift register and transmitted on UCAxTXD. Writing to the transmit data buffer clears UCAxTXIFG. The MSB of UCAxTXBUF is not used for 7-bit data and is reset.

9. There are two ways of receiving and transmitting data. One approach, is to use interrupts that tell the MSP430 when either the transmit buffer is empty and it's now OK to transmit a character or the receive buffer is full so it's OK for the MSP430 to acquire data from the receive buffer. The other method is to use the IFG2 receive and transmitter flags located in the IFG2 register along with disabling the interrupts in the IE2 register. With this approach, the transmit interrupt (UCA0TXIFG) and the receiver interrupt (UCA0RXIFG) flags are continuously polled until either it's OK to transmit data to the transmit buffer or it's OK to receive data from the receive buffer. For the method of the polling, the approach used for this laboratory experiment, a value of zero must be written into the IE2 register.

IE2, Interrupt Enable Register 2

7	6	5	4	3	2	1	0
						UCA0TXIE	UCA0RXIE
						rw–0	rw–0

	Bits 7-2	These bits may be used by other modules. See device-specific data sheet.
UCA0TXIE	Bit 1	USCI_A0 transmit interrupt enable 0 Interrupt disabled 1 Interrupt enabled
UCA0RXIE	Bit 0	USCI_A0 receive interrupt enable 0 Interrupt disabled 1 Interrupt enabled

IFG2, Interrupt Flag Register 2

7	6	5	4	3	2	1	0
						UCA0 TXIFG	UCA0 RXIFG
						rw–1	rw–0

	Bits 7-2	These bits may be used by other modules (see the device-specific data sheet).
UCA0 TXIFG	Bit 1	USCI_A0 transmit interrupt flag. UCA0TXIFG is set when UCA0TXBUF is empty. 0 No interrupt pending 1 Interrupt pending
UCA0 RXIFG	Bit 0	USCI_A0 receive interrupt flag. UCA0RXIFG is set when UCA0RXBUF has received a complete character. 0 No interrupt pending 1 Interrupt pending

To disable the interrupts so that polling of the UCA0TXIFG and UCA0RXIFG flags can be used a value of 0x00 is written into the IE2 register (IE2 = 0x00). The UCA0TXIFG flag is set to a value of 1 when the transmit buffer (UCA0TXBUF) is empty. This flag can be used to indicate when it is OK to send new data to be transmitted to the transmit buffer. The UCA0RXIFG flag is set to a value of 1 when the receive buffer (UCA0RXBUF) is full. The receive buffer is full when a full set of bits of serial data has been received. At this point it is OK to read the receive buffer.

10. The C language program included with this laboratory experiment has three functions associated with the UART. The first function Init UART initializes the various register values so to achieve the desired UART setting. The first item is to set the bits P2.4 to the UART transmit pin and P2.5 to the UART receive pin. Also until all of the setting are set for the UART, the UART is held in a reset mode. Upon completion of setting the UART values, the UART is then taken out of the reset mode.

The function INCHAR_UART function continuously reads the UCA0RXIFG flag until this flag is one. At this point, the function INCHAR_UART reads the UCA0RXBUF

register to obtain the received data (an ASCII character). This function returns the data received as an 8 bit unsigned character with a value in the range of 0 - 255.

The function OUTA_UART transmits an ASCII character passed to this function as an unsigned character with values in the range of 0 to 255. Like the INCHAR_UART function, the OUTA_UART function continuously polls the UCA0TXIFG flag until it is 1. At this point, the UCA0TXBUF register is empty and the OUTA_UART function sends data to the UCA0TXBUF register to be transmitted by the UART.

The main body of this program initializes the UART followed by waiting for user input and then transmits the received data back to the user for display. Finally, the program blinks the LED on the experimenter board to indicate that the program is running.

```
//-------------------------------------------------------------------
// Console I/O through the on board UART for MSP 430X4XXX
//-------------------------------------------------------------------

void  Init_UART(void);
void OUTA_UART(unsigned char A);
unsigned char INCHAR_UART(void);

#include "msp430fg4618.h"
#include "stdio.h"

int main(void){
volatile unsigned char a;
volatile unsigned int i;       // volatile to prevent optimization
WDTCTL = WDTPW + WDTHOLD;      // Stop watchdog timer
Init_UART();
a=INCHAR_UART();
OUTA_UART(a);
// go blink the light to indicate code is running

P2DIR |= 0x02;                 // Set P1.0 to output direction
                               // Use The LED as an indicator
for (;;){
  P2OUT ^= 0x02;               // Toggle P1.0 using exclusive-OR
  i = 10000;                   // SW Delay
  do i--;
  while (i != 0);
  }
}


void OUTA_UART(unsigned char A){
//-------------------------------------------------------------------
//*******************************************************************
//-------------------------------------------------------------------
// IFG2 register (1) = 1 transmit buffer is empty,
// UCA0TXBUF 8 bit transmit buffer
// wait for the transmit buffer to be empty before sending the
// data out
do{
  }while ((IFG2&0x02)==0);

// send the data to the transmit buffer
UCA0TXBUF =A;
}
```

```
unsigned char INCHAR_UART(void){
//-------------------------------------------------------------------
//*******************************************************************
//-------------------------------------------------------------------
// IFG2 register (0) = 1 receive buffer is full,
// UCA0RXBUF 8 bit receive buffer
// wait for the receive buffer is full before getting the data
do{
}while ((IFG2&0x01)==0);
// go get the char from the receive buffer
return (UCA0RXBUF);
}


void  Init_UART(void){
//-------------------------------------------------------------------
// Initialization code to set up the uart on the experimenter
// board to 8 data,
// 1 stop, no parity, and 9600 baud, polling operation
//-------------------------------------------------------------------
P2SEL=0x30;  // transmit and receive to port 2 b its 4 and 5
             // Bits p2.4 transmit and p2.5 receive
UCA0CTL0=0;  // 8 data, no parity 1 stop, uart, async
             // (7)=1 (parity), (6)=1 Even, (5)= 0 lsb first,
             // (4)= 0 8 data / 1 7 data,
             // (3) 0 1 stop 1 / 2 stop, (2-1)  -- UART mode,
             // (0) 0 = async
UCA0CTL1= 0x41;
             // select ALK 32768 and put in
             // software reset the UART
             // (7-6) 00 UCLK, 01 ACLK (32768 hz), 10 SMCLK,
             // 11 SMCLK
             // (0) = 1 reset
UCA0BR1=0;   // upper byte of divider clock word
UCA0BR0=3;   // clock divide from a clock to bit clock 32768/9600
             // = 3.413
             // UCA0BR1:UCA0BR0 two 8 bit reg to from 16 bit
             // clock divider
             // for the baud rate
UCA0MCTL=0x06;
             // low frequency mode module 3 modulation pater
             // used for the bit clock
UCA0STAT=0;  // do not loop the transmitter back to the
             // receiver for echoing
             // (7) = 1 echo back trans to rec
             // (6) = 1 framing, (5) = 1 overrun, (4) =1 Parity,
             // (3) = 1 break
             // (0) = 2 transmitting or receiving data
UCA0CTL1=0x40;
             // take UART out of reset
IE2=0;       // turn transmit interrupts off

//-------------------------------------------------------------------
//*******************************************************************
//-------------------------------------------------------------------
            // IFG2 register (0) = 1 receiver buffer is full,
            // UCA0RXIFG
            // IFG2 register (1) = 1 transmit buffer is empty,
            // UCA0RXIFG
            // UCA0RXBUF 8 bit receiver buffer
            // UCA0TXBUF 8 bit transmit buffer
}
```

11. The steps required for an assembly language program is very similar to those used to produce a C language program. The assembly language program given below initializes the UART and then waits for user input and then transmits this received character. Finally, this program flashes the LED on the experimenter board to indicate that the program is running. Also included with this assembly language program, are examples on defining variables or constants. Variables can be defined as bytes, 16-bit words or strings. Since variables can change in value, these variables are stored in RAM using the .sect ".sysmem" assembler directive. Constants are stored in ROM similar to program code. The .sect ".const" assembler directive allocates constants in ROM instead of RAM.

```
;*****************************************************************
;   Console I/O through the on board UART for MSP 430X4XXX
;   experimenter board RAM at 0x1100 - 0x30ff, FLASH at 0x3100;
;   - 0xfbff
;*****************************************************************

            .cdecls C,LIST,"msp430fg4618.h"
; cdecls tells assembler to allow the device header file
;-----------------------------------------------------------------
;   Main Code
;-----------------------------------------------------------------
; This is the stack and variable area of RAM and begins at
; address 0x1100 can be used for program code or constants

;           .sect ".stack"      ; data ram for the stack  ;
      .sect ".const"      ; data rom for initialized
                                ; data constants
;           .sect ".text"       ; program rom for code
;           .sect ".cinit"      ; program rom for global inits
;           .sect ".reset"      ; MSP430 RESET Vector
;           .sect ".sysmem"     ; data ram for initialized
                                ; variables. Use this .sect to
                                ; put data in RAM
;dat        .byte 0x34          ; example of defining a byte
            .bss label, 4       ; allocates 4 bytes of
                                ; uninitialized memory with the
                                ; name label
            .word 0x1234        ; example of defining a 16 bit
                                ; word
;strg2      .string "Hello World" ; example of a string store in
                                ; RAM
            .byte 0x0d,0x0a     ; add a CR and a LF to the string
            .byte 0x00          ; null terminate the string
; This is the constant area flash begins at address 0x3100 can be
; used for program code or constants
            .sect ".const"      ; initialized data rom for
                                ; constants. Use this .sect to
                                ; put data in ROM
strg1       .string "This is a test" ; example of a string stored
                                ; in ROM
            .byte 0x0d,0x0a     ; add a CR and a LF
            .byte 0x00          ; null terminate the string with

; This is the code area flash begins at address 0x3100 can be
; used for program code or constants

            .text                       ; program start
            .global _START              ; define entry point
```

```
;-------------------------------------------------------------------
START       mov.w   #300h,SP            ; Initialize 'x1121
                                        ; stackpointer
StopWDT     mov.w   #WDTPW+WDTHOLD,&WDTCTL  ; Stop WDT
SetupP1     bis.b   #02h,&P2DIR         ; P2.2 output

; go initialize the uart
            call    #Init_UART

Mainloop    xor.b   #02h,&P2OUT         ; Toggle P2.2
Wait        mov.w   #0A000h,R15         ; Delay to R15
L1          dec.w   R15                 ; Decrement R15
            jnz     L1                  ; Delay over?
; go print a character to the screen from the keyboard
             call    #INCHAR_UART
             call    #OUTA_UART
             jmp     Mainloop                ; Again
                                             ;

OUTA_UART
;-------------------------------------------------------------------
; prints to the screen the ASCII value stored in register 4 and
; uses register 5 as a temp value
;-------------------------------------------------------------------
; IFG2 register (1) = 1 transmit buffer is empty,
; UCA0TXBUF 8 bit transmit buffer
; wait for the transmit buffer to be empty before sending the
; data out
             push R5
lpa          mov.b &IFG2,R5
             and.b #0x02,R5
             cmp.b #0x00,R5
             jz lpa
; send the data to the transmit buffer UCA0TXBUF = A;
             mov.b R4,&UCA0TXBUF
             pop R5
             ret


INCHAR_UART
;-------------------------------------------------------------------
; returns the ASCII value in register 4
;-------------------------------------------------------------------
; IFG2 register (0) = 1 receive buffer is full,
; UCA0RXBUF 8 bit receive buffer
; wait for the receive buffer is full before getting the data

             push R5
lpb          mov.b &IFG2,R5
             and.b #0x01,R5
             cmp.b #0x00,R5
             jz lpb
             mov.b &UCA0RXBUF,R4
             pop R5
; go get the char from the receive buffer
             ret


Init_UART
;-------------------------------------------------------------------
; Initialization code to set up the uart on the experimenter board
to 8 data,
;     1 stop, no parity, and 9600 baud, polling operation
;-------------------------------------------------------------------
```

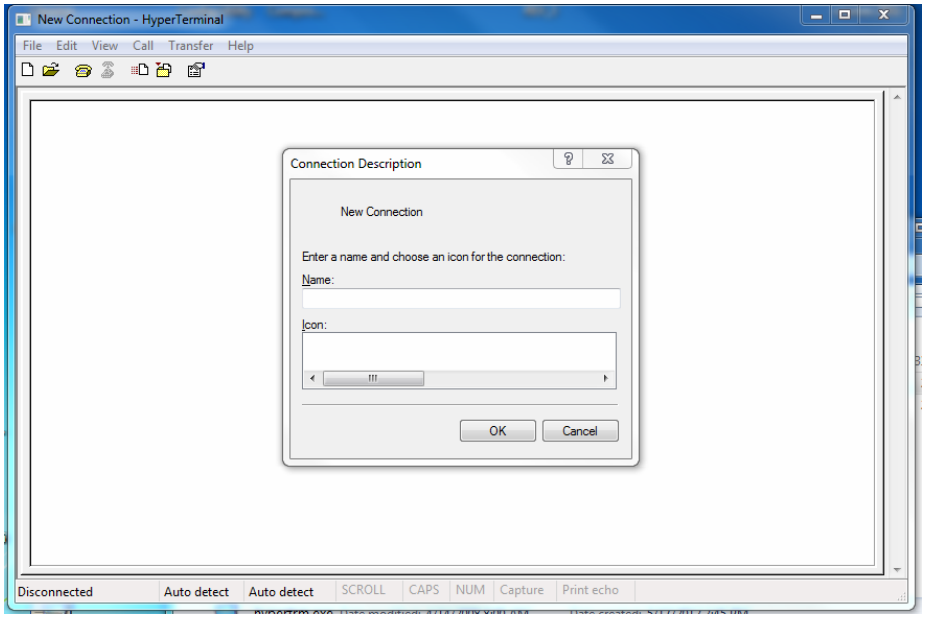
```
;P2SEL=0x30;
; transmit and receive to port 2 b its 4 and 5
            mov.b #0x30,&P2SEL
; Bits p2.4 transmit and p2.5 receive UCA0CTL0=0
; 8 data, no parity 1 stop, uart, async
            mov.b #0x00,&UCA0CTL0
; (7)=1 (parity), (6)=1 Even, (5)= 0 lsb first,
; (4)= 0 8 data / 1 7 data, (3) 0 1 stop 1 / 2 stop, (2-1)  --
; UART mode, (0) 0 = async
; UCA0CTL1= 0x41;
            mov.b #0x41,&UCA0CTL1
; select ALK 32768 and put in software reset the UART
; (7-6) 00 UCLK, 01 ACLK (32768 hz), 10 SMCLK, 11 SMCLK
; (0) = 1 reset
;UCA0BR1=0;
; upper byte of divider clock word
            mov.b #0x00,&UCA0BR1
;UCA0BR0=3;              ;
; clock divide from a clock to bit clock 32768/9600 = 3.413
            mov.b #0x03,&UCA0BR0
; UCA0BR1:UCA0BR0 two 8 bit reg to from 16 bit clock divider
; for the baud rate
;UCA0MCTL=0x06;
; low frequency mode module 3  modulation pater used for the bit
; clock
            mov.b #0x06,&UCA0MCTL
;UCA0STAT=0;
; do not loop the transmitter back to the receiver for echoing
            mov.b #0x00,&UCA0STAT
; (7) = 1 echo back trans to rec
; (6) = 1 framing, (5) = 1 overrun, (4) =1 Parity, (3) = 1 break
; (0) = 2 transmitting or receiving data
;UCA0CTL1=0x40;
; take UART out of reset
            mov.b #0x40,&UCA0CTL1
;IE2=0;
; turn transmit interrupts off
            mov.b #0x00,&IE2
; (0) = 1 receiver buffer Interrupts enabled
; (1) = 1 transmit buffer Interrupts enabled
;-----------------------------------------------------------------
;*****************************************************************
;-----------------------------------------------------------------
; IFG2 register (0) = 1 receiver buffer is full, UCA0RXIFG
; IFG2 register (1) = 1 transmit buffer is empty, UCA0RXIFG
; UCA0RXBUF 8 bit receiver buffer, UCA0TXBUF 8 bit transmit
; buffer
            ret

;-----------------------------------------------------------------
;           Interrupt Vectors
;-----------------------------------------------------------------
            .sect   ".reset"                ; MSP430 RESET Vector
            .short  START                   ;
            .end
```

Procedure: Part 1. C language program implementation of the MSP430FG4618 UART

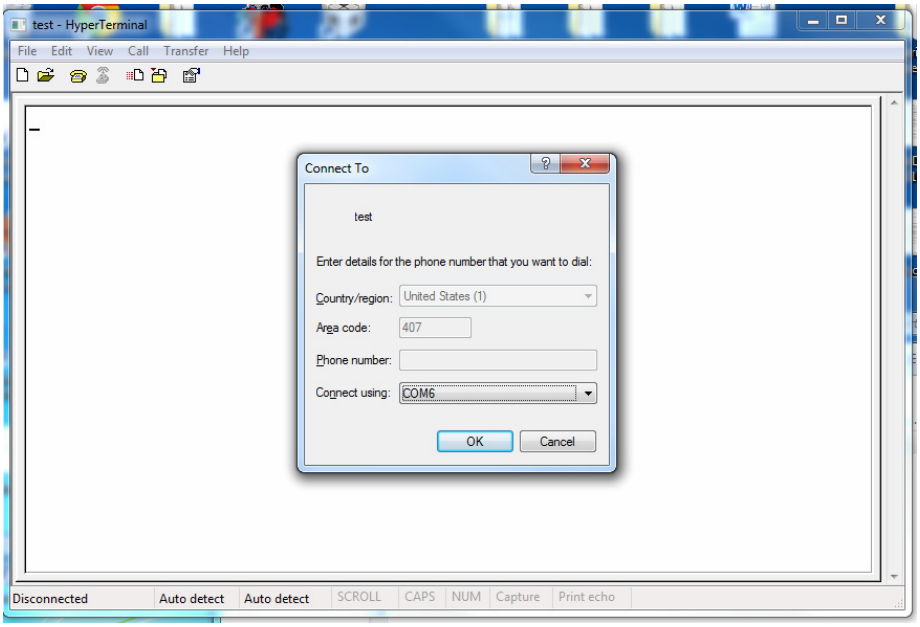
1. Read the C language program included with this laboratory experiment. Become familiar with the setting of the various registers used by the MSP430FG4818 to configure the UART.

2. Build a CCS project using the steps described in experiment #1 using the C language program included with this laboratory.

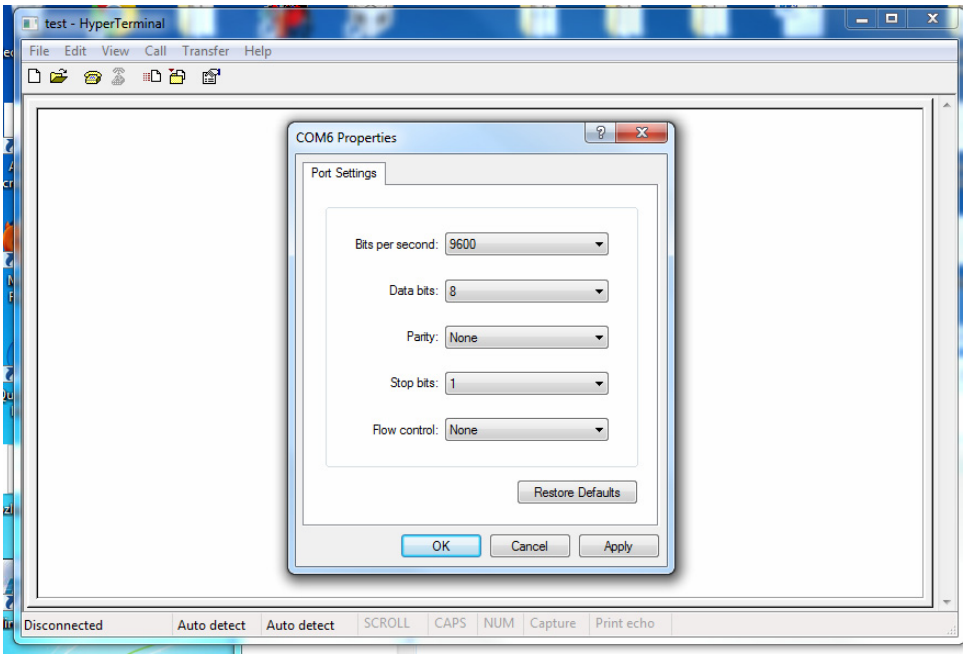
3. Open Hyperterminal on the Computer and create a new connection entering any name that is desired for this new connection.



4. Select the appropraite communication port used by the RS232C hardware on the desktop computer.



5. Next, select the baud rate, the number of data bits, parity, number of stop bits to match the setting values for the MSP430 UART (9600 Baud, 1 Stop Bit and No Parity). Also select flow control to none.



6. Debug and run the program. The program should allow the user to enter in a single character from the keyboard and then display this character back in the hyperterminal screen. Finally, the program should blink the yellow LED on the experimenter board.

7. Modify the existing C language program (a new CCS project) by adding a new function that prints one character at a time via the UART from a string of characters to the hyperterminal display. This function should continue printing characters until the null character is reached (0x00) indicating an end of a string. Passed to this function should be the starting address of the string. Nothing is needed to be returned by this function.

8. Add the string "Laboratory #2 for EEL4742 Embedded Systems" to the C language program generated in Step 7. Using the C language program developed in Step 7 build the project, debug and then execute this project. Verify that the program displayed on the hyperterminal screen is "Laboratory #2 for EEL4742 Embedded Systems".

9. Write a new C language program (a new CCS project) that inputs from the keyboard the uppercase ASCII characters "G" or "Y". Once the upper case letter "G" is entered, the green light on the experimenter board turns on and stays on until the upper case letter "G" is entered again. At this point, the green LED turns off. The same operation applies for the yellow LED.

10. Write a new C language program (a new CCS project) that reads the status of SW1 and SW2 on the experimenter board and displays a one on the hyperterminal display if the button is pressed, otherwise it displays a zero. For example, the display output on the hyperterminal display if SW1 is pressed and SW2 is not pressed is:

SW1 = 1, SW2 = 0

Procedure: Part 2. An assembly language program implementation of the MSP430FG4618 UART

1. Read the assembly language program included with this laboratory experiment. Become familiar with the setting of the various registers used by the MSP430FG4818 to configure the UART.

2. Build a CCS project using the steps described in experiment #1 using the assembly language program included with this laboratory.

3. Open hyperterminal on the desktop computer as described in Part 1 of the procedure for this laboratory experiment.

4. Debug and run the program. The program should allow the user to enter in a single character from the keyboard and then display this character back in the hyperterminal screen. Unlike the C program, this program allows the user to continuously enter in characters from the keyboard while changing the state of the yellow LED each time.

5. Modify the existing assembly language program (a new CCS project) by adding a new subroutine that prints one character at a time from a string of characters via the UART to the hyperterminal display. This subroutine should continue printing characters until the null character is reached (0x00) indicating an end of the string. Passed to this subroutine in a register should be the starting address of the string. Nothing is needed to be returned by this subroutine.

6. Add the string "Laboratory #2 for EEL4742 embedded Systems" to the assembly language program generated in Step 7. Using this assembly language program, build the project, debug and then execute this project. Verify that the program displayed on the hyperterminal screen is "Laboratory #2 for EEL4742 embedded Systems". Included in the assembly language program with this laboratory experiment are examples on how to reserve data using the MSP430 assembler. Study the example on how to reserve a string in assembly language.

7. Write a new assembly language program (a new CCS project) that inputs from the keyboard the uppercase ASCII characters "G" or "Y". Once the upper case letter "G" is entered, the green light on the experimenter board turns on and stays on until the upper case letter "G" is entered again. At this point, the green LED turns off. The same operation applies for the yellow LED.

8. Write a new assembly language program (a new CCS project) that reads the status of SW1 and SW2 on the experimenter board and display a one on the hyperterminal display if the button is pressed otherwise it displays a zero. For example, the display output on the hyperterminal display if neither SW1 nor SW2 are pressed is:

SW1 = 0, SW2 = 0

Report

1. Include in the laboratory report the objective of the laboratory experiment.

2. The procedure used to generate and execute a C-language and an assembly language program (To be written in your own words not cut paste from the project).

3. The source code for the C-language language programs in Part 1 of the procedures (4 commented programs).

4. The source code for the assembly language programs in Part 2 of the procedures (4 commented programs).

5. A summary of the laboratory experiment and what you have learned.

EXPERIMENT #3

BASIC INPUT AND OUTPUT USING THE MSP430 UART

Goals: To develop C language and assembly language programs that converts ASCII characters to binary and hexadecimal numbers. The UART on the MSP430FG4618 will be used via hyperterminal to enter in ASCII data and to display ASCII data.

References: MSP430x4xx Family User's Guide, MSP430 Assembly Language Tools v 4.0 User's Guide, MSP430FG4618/F2013 Experimenter's Board, the help menu in code composer studio, MSP430 Microcontroller Basics by John H. Davies and an ASCII table.

Equipment: A desktop computer system running code composer, the MSP430FG4618 experimenter board and the MSP430 USB-Debug interface (MSP-FET430UIF).

Pre-laboratory: Read this experiment carefully to become familiar with the procedural steps in this experiment. Develop a first draft of the various programs that will be required for this laboratory experiment.

Discussion: This laboratory experiment discusses the conversion of an ASCII hexadecimal number to a 4 bit binary number. The ASCII numbers "0" – "9" are given by the values of 0x30 to 0x39. To convert these digits to a four bit binary number, a value of 0x30 is subtracted from the number. For the hexadecimal digits of "A" – "F", a different conversion value is used. Since the ASCII letters "A" – "F" have values of 0x41 through 0x46 to convert these ASCII letters to hexadecimal values of 0x0A – 0x0F a value of 0x37 must be subtract from each of these ASCII letter.

Conversion from a hexadecimal number to an ASCII symbol is the same except for the hexadecimal number in the range of 0 – 9 then a value of 0x30 is added to the hexadecimal number. If the hexadecimal number is in the range of A – F then a value of 0x37 is added to these hexadecimal numbers.

To convert an ASCII uppercase letter to an ASCII lowercase letter a value of 0x20 is added to the uppercase letter. For example to convert the uppercase letter "A" which has a hexadecimal value of 0x41 to the lower case letter "a" a value of 0x20 is added to the 0x41 to produce a value of 0x61.

Procedure Part 1. Generating Basic input with C language programs.

1. Using the procedures from laboratory experiments 1 and 2 write a C language program that inputs two hexadecimal ASCII numbers. Next, the program should convert

these ASCII numbers to an 8 bit binary number. Finally, this program should display the equivalent ASCII character on a new line in the hyperterminal window. The program should then print a new line and start over waiting for user input. For example, if the input to the program is a "3F", the program should display on a new line "?" (0x3F is the ASCII code for ?).

2. Write a C language program that inputs one ASCII character and displays on a new line in the hyperterminal window its two digit hexadecimal value. The program should then print a new line and start over waiting for user input. For example, if the input to the program is a "T", the program should display the numeric value of 54.

3. Write a C language program that inputs one ASCII character ("A – Z" or "a – z"). Next, the program should convert the uppercase letters to a lowercase and convert the lowercase letters to uppercase. The program should display on a new line in the hyperterminal window the converted letter. The program should then print a new line and start over waiting for user input. For example, if the input is the ASCII capital letter "A" then on a new line the ASCII lower case "a" is displayed.

4. Write a C language program that inputs 8 ASCII "0" or "1" digits. Next, the program should convert this value to an eight bit binary number. Finally, the program should display on a new line in the hyperterminal window, the hexadecimal value of this eight bit binary number. The program should then print a new line and start over waiting for user input. For example, if the input is 11010110 the display output should be "D6".

5. Write a C language program that inputs ASCII characters and stores them in a character array until the enter key is hit (0x0D) or until 32 characters have been entered. Next, end the character array with the null character (0x00). Display this character (string) array on a new line in the hyperterminal window. Finally, have the program sort these ASCII characters from lowest to highest and display on a new line this sorted character array. The program should then print a new line and start over waiting for user input. For example, if the input is:

45F37X

The output display for this program would be:

45F37X
3457FX

A simple sorting algorithm can be used to sort the ASCII character array:

```
for(i = 0; i++; i < N-1){
    for(j = 0; j++; j < N-1){
            if( ARRAY[j] > ARRAY[j+1] ){
                temp =ARRAY[j];
                ARRAY[j]=ARRAY[j+1]
```

```
                    ARRAY[j+1]=ARRAY[j]
                    }
                }
            }
```

where N is the number of elements in the array ARRAY[i] that are to be sorted.

Procedure Part 2. Generating Basic input with assembly language programs.

1. Using the procedures from laboratory experiments 1 and 2, write an assembly language program that inputs two hexadecimal ASCII numbers. Next, the program should convert these ASCII numbers to an 8 bit binary number. Finally, this program should display the equivalent ASCII character on a new line in the hyperterminal window. The program should then print a new line and start over waiting for user input. For example, if the input to the program is a "2E", the program should display on a new line ".".

2. Write an assembly language program that inputs one ASCII character and displays on a new line in the hyperterminal window its two digit hexadecimal value. The program should then print a new line and start over waiting for user input. For example, if the input to the program is "S", the program should display the numeric value of 53.

3. Write an assembly language program that inputs one ASCII character ("A – Z" or "a – z"). Next, the program should convert the uppercase letters to a lowercase and convert the lowercase letters to uppercase. The program should display on a new line in the hyperterminal window, the converted letter. The program should then print a new line and start over waiting for user input. For example, if the input is the ASCII lower case letter "b", then on a new line the ASCII capital letter "B" is displayed.

4. Write an assembly language program that inputs 8 ASCII "0" or "1" digits. Next, the program should convert this value to an eight bit binary number. Finally, the program should display on a new line in the hyperterminal window, the hexadecimal value of this eight bit binary number. The program should then print a new line and start over waiting for user input. For example, if the input is 11010100 the display output should be "D4".

5. Write an assembly language program that inputs ASCII characters and stores them in a character array until the enter key is hit (0x0D) or until 32 characters have been entered. Next, end the character array with the null character (0x00). Display this character (string) array on a new line in the hyperterminal window. Finally, have the program sort these ASCII characters from lowest to highest and display on a new line this sorted character array. The program should then print a new line and start over waiting for user input. For example, if the input is:

DC12167

The output display for this program would be:

The same simple sorting algorithm given in Step 4 of Part 1 of the Procedures can be used to sort the ASCII character array but this time needs to be written in assembly language using two loops.

Report

1. Include in the laboratory report the objective of the laboratory experiment.

2. The procedure used to generate and execute the C-language and an assembly language programs.

3. The source code for the C-language language programs in Part 1 of the procedures (5 commented programs).

4. The source code for the assembly language programs in Part 2 of the procedures (5 commented programs).

5. A summary of the laboratory experiment and what you have learned.

EXPERIMENT #4

LCD DISPAY USING THE MSP430FG4618 EXPERIMENTER BOARD

Goals: To develop C language and assembly language programs that provide an interface to a standard matrix LCD module. In particular, the experimenter board which contains the MSP430FG4618 and the on board LCD display will be used.

References: MSP430x4xx Family User's Guide (Chapter 26), MSP430 Assembly Language Tools v 4.0 User's Guide, MSP430FG4618/F2013 Experimenter's Board, the help menu in code composer studio, MSP430 Microcontroller Basics by John H. Davies, and laboratory experiments 3 and 4.

Equipment: A desktop computer system running code composer, the MSP430FG4618 experimenter board and the MSP430 USB-Debug interface (MSP-FET430UIF).

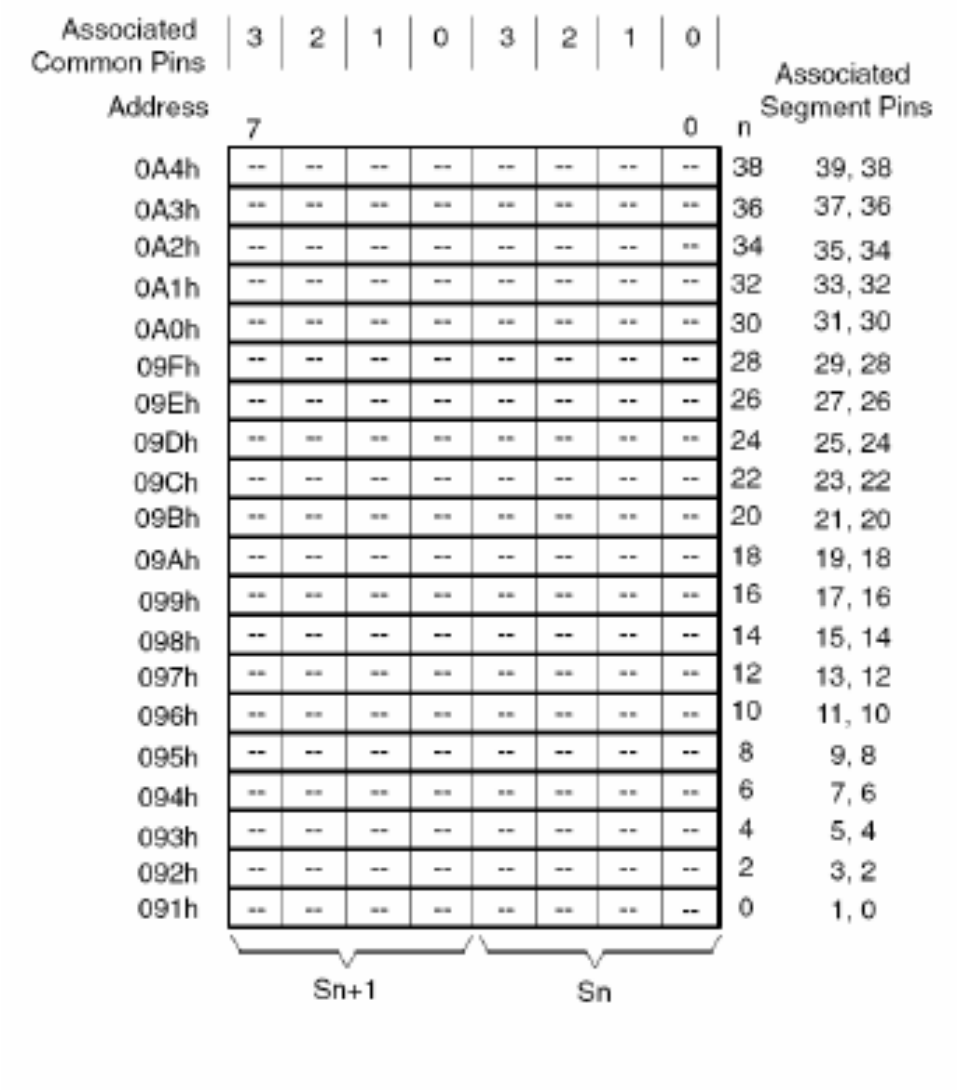
Pre-laboratory: Read this experiment carefully to become familiar with the procedural steps in this experiment. Study in detail and become familiar with the C language program and the assembly language program provided with this laboratory experiment. Develop an approach to implement the C language and assembly language programs required in Parts 1 and 2 of the procedures.

Discussion: The laboratory experiment will give the step by step procedures on creating a C language and an assembly language program that provides interfacing to the onboard LCD display.

Part 1. Generating a C language program that provides serial communications.

1. The LCD that is located on the experimenter board contains both seven segment elements as well as specialty characters. By writing to the appropriate address and memory and configuring the correct timing and voltage signals, the LCD display can be used to display hexadecimal numbers plus these specialty symbols.

2. LCD MEMORY MAP: Located in the memory space of the MSP430FG4618 are the allocated memory spaces that are reserved for the LCD display. The LCD contains both segment pins as well as common pins. In this manner, the number of pins required to interface to the LCD display can be reduced. This requires that the LCD display be time multiplexed with a set of segments / display elements turning on for a given instant of time. The LCD is scanned at a rate that is faster than the eye can see so to the user the LCD display appears to be continuously running and flicker free. A further discussion of how each element of the LCD display is time multiplexed is given below. This requires that a scan rate at which the LCD display is time multiplexed be specified during the initialization of the LCD interface. The particular version of the LCD display on the experimenter board uses memory addresses starting at the address location LCDM3 (the right most seven segment display) and uses a total of 11 consecutive memory locations. Writing a one to the appropriate bit in these locations turns on the associated LCD display element.

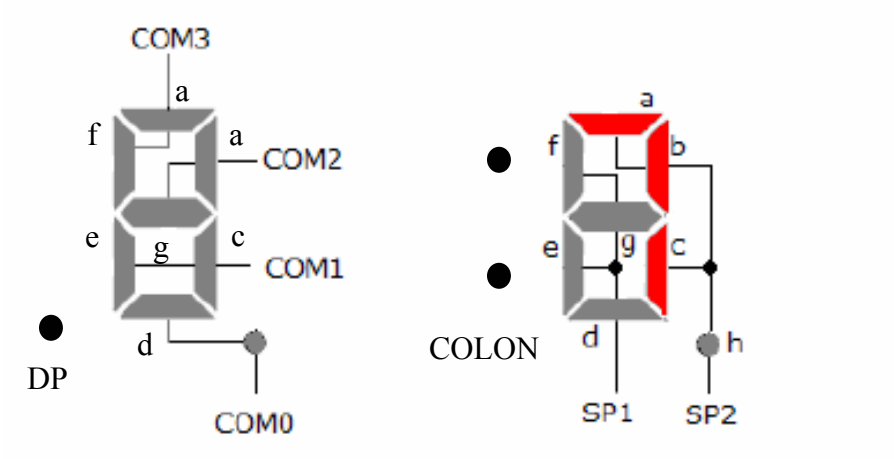


3. BIT ORDER: The MSB is used for either the decimal point or the colon between the seven segment displays. It depends on the actual display location. The colons are located such that the LCD display can be used for a time a day clock.

B7	B6	B5	B4	B3	B2	B1	B0
DP	e	g	f	d	c	b	a
COLON	e	g	f	d	c	b	a

To reduce the number of pins required by the LCD, each segment of the LCD is scanned at a rate faster than the persistence of the eye. Using this approach, allows the LCD display pins to be time shared. Each of COM0 – COM3 lines as well as the segments SP1 and SP2 lines are time multiplexed. In red, is the example of the segments that must be on to display the number "7". To display the number "7" SP2 control line along with COM1, COM2, and COM3 are used.

The multiplexing of COM0 – COM3 and SP1 – SP2 is done by the LCD_A controller. The programmer only needs to determine the scan rate of the LCD_ and which segments of the seven segment display needs to be turned on. Turning on an LCD segment is done by writing a one in the appropriate memory location in the memory map for the LCD display starting at LCDM3.



4. LCDAVCTLO register: Before the LCD display can be used, power and bias voltages must be applied to the LCD display. This is accomplished by setting the various configuration bits in the LCD display voltage control register. For this laboratory experiment, the LCD display will obtain its voltage VLCD internally and will use a 1/3 bias voltage. Since the LCD display will be using an internal voltage, all external voltages are ignored and disabled. This gives a value of 0x00 that needs to be written to the LCDAVCTL0 register.

LCDAVCTL0, LCD_A Voltage Control Register 0

7	6	5	4	3	2	1	0
Unused	R03EXT	REXT	VLCDEXT	LCDCPEN	VLCDREFx		LCD2B
rw–0	rw–0	rw–0	rw–0	rw–0	rw–0	rw–0	rw–0

Unused	Bit 7	Unused
R03EXT	Bit 6	V5 voltage select. This bit selects the external connection for the lowest LCD voltage. R03EXT is ignored if there is no R03 pin available. 0 V5 is AV_{SS} 1 V5 is sourced from the R03 pin
REXT	Bit 5	V2 – V4 voltage select. This bit selects the external connections for voltages V2 – V4. 0 V2 – V4 are generated internally 1 V2 – V4 are sourced externally and the internal bias generator is switched off
VLCDEXT	Bit 4	V_{LCD} source select 0 V_{LCD} is generated internally 1 V_{LCD} is sourced externally
LCDCPEN	Bit 3	Charge pump enable. 0 Charge pump disabled. 1 Charge pump enabled when V_{LCD} is generated internally (VLCDEXT = 0) and VLCDx > 0 or VLCDREFx > 0.
VLCDREFx	Bits 2–1	Charge pump reference select 00 Internal 01 External 10 Reserved 11 Reserved
LCD2B	Bit 0	Bias select. LCD2B is ignored when LCDMx = 00. 0 1/3 bias 1 1/2 bias

5. LCDAPCTL0 LCDAPCTL1 registers: This register is used to indicate to the MSP430 how many LCD memory addresses must be reserved by the MSP430. Writing a one to each of the selected bits reserves these memory locations for the LCD. The particular version of the LCD display used on the experimenter board uses on segments S4 through S24. As such LCDAPCTL1 is loaded a value of 0x00 (LCDS32 and LCDS36 are not used) which its default value and LCDAPCTL0 is loaded a value of 0x7E (LCDS0 is not used).

LCDAPCTL0, LCD_A Port Control Register 0

7	6	5	4	3	2	1	0
LCDS28	LCDS24	LCDS20	LCDS16	LCDS12	LCDS8	LCDS4	LCDS0†
rw–0	rw–0	rw–0	rw–0	rw–0	rw–0	rw–0	rw–0

† Segments S0–S3 on the MSP430FG461x devices are disabled from LCD functionality when charge pump is enabled.

LCDS28 Bit 7 LCD segment 28 to 31 enable
This bit only affects pins with multiplexed functions. Dedicated LCD pins are always LCD function.
0 Multiplexed pins are port functions.
1 Pins are LCD functions

LCDS24 Bit 6 LCD segment 24 to 27 enable
This bit only affects pins with multiplexed functions. Dedicated LCD pins are always LCD function.
0 Multiplexed pins are port functions.
1 Pins are LCD functions

LCDS20 Bit 5 LCD segment 20 to 23 enable
This bit only affects pins with multiplexed functions. Dedicated LCD pins are always LCD function.
0 Multiplexed pins are port functions.
1 Pins are LCD functions

LCDS16 Bit 4 LCD segment 16 to 19 enable
This bit only affects pins with multiplexed functions. Dedicated LCD pins are always LCD function.
0 Multiplexed pins are port functions.
1 Pins are LCD functions

LCDS12 Bit 3 LCD segment 12 to 15 enable
This bit only affects pins with multiplexed functions. Dedicated LCD pins are always LCD function.
0 Multiplexed pins are port functions.
1 Pins are LCD functions

LCDS8 Bit 2 LCD segment 8 to 11 enable
This bit only affects pins with multiplexed functions. Dedicated LCD pins are always LCD function.
0 Multiplexed pins are port functions.
1 Pins are LCD functions

LCDS4 Bit 1 LCD segment 4 to 7 enable
This bit only affects pins with multiplexed functions. Dedicated LCD pins are always LCD function.
0 Multiplexed pins are port functions.
1 Pins are LCD functions

LCDS0 Bit 0 LCD segment 0 to 3 enable
This bit only affects pins with multiplexed functions. Dedicated LCD pins are always LCD function.
0 Multiplexed pins are port functions.
1 Pins are LCD functions

LCDAPCTL1, LCD_A Port Control Register 1

7	6	5	4	3	2	1	0
Unused						LCDS36	LCDS32
rw–0	rw–0	rw–0	rw–0	rw–0	rw–0	rw–0	rw–0

Unused	Bits 7–2	Unused
LCDS36	Bit 1	LCD segment 36 to 39 enable This bit only affects pins with multiplexed functions. Dedicated LCD pins are always LCD function. 0 Multiplexed pins are port functions. 1 Pins are LCD functions
LCDS32	Bit 0	LCD segment 32 to 35 enable This bit only affects pins with multiplexed functions. Dedicated LCD pins are always LCD function. 0 Multiplexed pins are port functions. 1 Pins are LCD functions

6: LCDACTL register: This register configures the clock frequency derived from the ACLK as well as turns on the LCD display system. Since the ACLK is at 32768 Hz, for a LCD scan frequency of 256 Hz, a divisor of 128 is needed (LCDFREQ = 011). Also LCDSON and LCDON need to be set to one. LCDON turns on the LCD system and LCDSON is the master segment control that can be used to turn all of the LCD segments off. LCDSON is typically used to flash the LCD display on / off. Since the LCD display on the experimenter board uses four common lines COM0-COM3, a 4 to 1 multiplex is required by the MSP430FG4618. This requires that LCDMX be set to the four mux option (11). Combining all of these bits together requires that the LCDACTL register be set to 0x7D.

7. The C language program included with this laboratory experiment first initializes the LCD system by writing to LCDACTL, LCDAPCTL0, and LCDAVCTL0 registers. In addition, the initialization function writes zero to all the bits reserved for the LCD segments turning off all of the LCD display elements. Next, the program sets all of the bits allocated in the MSP430 memory reserved for the LCD segments. This results in all of the LCD display elements turning on. For example, to write the number 1 to the right most display a value of 0x06 is written to LCDM3 memory location turning on segment elements b and c. Since the data that must be written to the display must be an unsigned eight bit number, a pointer is assigned as an unsigned character and pointer to the address of LCDM3:

```
unsigned char  *LCDSeg = (unsigned char *) &LCDM3;
```

To access the various memory locations used by the LCD display (after the pointer to LCDMEM3 has been defined) either a pointer equation is used or the standard C language array access is used:

```
*(LCDSeg+n) = 0;   or     LCDSeg[n]=0;
```

Finally, once all of the segments of the LCD are turned on, the program blinks the yellow LED to indicate that the program is running.

LCDACTL, LCD_A Control Register

7	6	5	4	3	2	1	0
LCDFREQx			LCDMXx		LCDSON	Unused	LCDON
rw–0	rw–0	rw–0	rw–0	rw–0	rw–0	rw–0	rw–0

LCDFREQx	Bits 7-5	LCD frequency select. These bits select the ACLK divider for the LCD frequency. 000 Divide by 32 001 Divide by 64 010 Divide by 96 011 Divide by 128 100 Divide by 192 101 Divide by 256 110 Divide by 384 111 Divide by 512
LCDMXx	Bits 4-3	LCD mux rate. These bits select the LCD mode. 00 Static 01 2-mux 10 3-mux 11 4-mux
LCDSON	Bit 2	LCD segments on. This bit supports flashing LCD applications by turning off all segment lines, while leaving the LCD timing generator and R33 enabled. 0 All LCD segments are off 1 All LCD segments are enabled and on or off according to their corresponding memory location.
Unused	Bit 1	Unused
LCDON	Bit 0	LCD On. This bit turns on the LCD_A module. 0 LCD_A module off. 1 LCD_A module on.

```
//-------------------------------------------------------------------
//  LCD Driver for the for MSP 430X4XXX experimenter board using
//  Softbaugh LCD
//  Davies book pg. 259, 260
//-------------------------------------------------------------------

#include "msp430fg4618.h"
#include "stdio.h"

void  Init_LCD(void);
 // setup a pointer to the area of memory of the TMS430 that points to
 // the segments
 // of the softbaugh LCD LCDM3 = the starting address
 // each of the seven segments for each display is store in memory
 // starting at address LCDM3
 // which is the right most seven segment of the LCD
```

```
 // The bit order in each byte is
 // dp, E, G, F, D, C, B, A    or
 //  :, E, G, F, D, C, B, A
 // after the seven segments these memory locations are used to turn on
 // the special characters
 // such as battery status, antenna, f1-f4, etc.
 // there are 7 seven segment displays
unsigned char  *LCDSeg = (unsigned char *) &LCDM3;
 // there are 11 locations that are needed for the softbaugh LCD
 // ony 7 used for the seven segment displays
int LCD_SIZE=11;

int main(void){
 volatile unsigned char a;
 volatile unsigned int i;            // volatile to prevent optimization
 WDTCTL = WDTPW + WDTHOLD;           // Stop watchdog timer
 // setup pprt 3 as an output so to be able to turn on the LED
 P2DIR |= 0x02;                      // Set P1.0 to output direction
 //  go Initialize the LCD
 Init_LCD();
 //  Turn on all of the segments
 //  LCD_SIZE-4 only gives the 7 segment displays plus DP,
 //  and colons colons = dp
 //  Right most display is at LCDSeg[0];
 for (i=0;i<LCD_SIZE;i++){
 //  To turn on a segment of the LCD a one is written in the
 //  appropriate location in the LCD memory
 //  Setting all the bits to 1 for all memory locations turns on
 //  all of the display elements
 //  Including all of the special characters
     LCDSeg[i]=0xff;
    }
 //  now that all the LCD segments have been turned on just toggle
 //  the yellow LED on / off
  for (;;){
    P2OUT ^= 0x02;                   // Toggle P1.0 using exclusive-OR
    i = 10000;                       // SW Delay
    do i--;
    while (i != 0);
  }
}

//-------------------------------------------------------------------------
//                      Initialize the LCD system
//-------------------------------------------------------------------------
void  Init_LCD(void){
 //  Using the LCD A controller for the MSP430fg4618
 //  the pins of the LCD are memory mapped onto the mp430F4xxx
 //  memory bus and
 //  are accessed via LCDSeg[i] array
 //  See page 260 of Davie's text
 //  LCD_SIZE-4 only gives the 7 segment displays plus DP, and
 //  (colons are the same bit setting)
 //  LCD_SIZE-4 only gives the 7 segment displays plus DP, and
 //  colons (colons / dp)
 //  Right most seven segment display is at LCDSeg[0];
 //  Display format
 //                   AAA
 //                 F   B
 //            X    F   B
 //                  GGG
 //            X    E   C
 //                 E   C
 //            DP    DDD
```

```
//  bit order
//  dp, E, G, F, D, C, B, A    or
//   :, E, G, F, D, C, B, A
int n;
for (n=0;n<LCD_SIZE;n++){
     // initialize the segment memory to zero to clear the LCD
     // writing a zero in the LCD memory location clears turns
     // off the LCD segment
     //  Including all of the special characters
     // This way or
     *(LCDSeg+n) = 0;
     //    LCDSeg[n]=0;
     }
// Port 5 ports 5.2-5.4 are connected to com1, com2, com3 of LCD and
// com0 is fixed and already assigned
// Need to assign com1 - com3 to port5
P5SEL = 0x1C; // BIT4 | BIT3 |BIT2 = 1 P5.4, P.3, P5.2 = 1
//  Used the internal voltage for the LCD bit 4 = 0 (VLCDEXT=0)
//  internal bias voltage set to 1/3 of Vcc, charge pump disabled,
//  page 26-25 of MSP430x4xx user manual
LCDAVCTL0 = 0x00;
//  LCDS28-LCDS0 pins LCDS0 = lsb and LCDS28 = MSB need
//  LCDS4 through LCDS24
//  from the experimenter board schematic the LCD uses S4-S24,
//  S0-S3 are not used here
//  Only use up to S24 on the LCD 28-31 not needed.
//  Also LCDACTL1 not required since not using S32 - S39
//  Davie's book page 260
//  page 26-23 of MSP430x4xx user manual
LCDAPCTL0 = 0x7E;
//  The LCD uses the ACLK as the master clock as the scan rate for
//  the display segments
//  The ACLK has been set to 32768 Hz with the external
//  327768 Hz crystal
//  Let's use scan frequency of 256 Hz (This is fast enough not
//  to see the display flicker)
//  or a divisor of 128
//  LCDFREQ division(3 bits), LCDMUX (2 bits), LCDSON segments on,
//  Not used, LCDON LCD module on
//  011 = freq /128, 11 = 4 mux's needed since the display uses for
//  common inputs com0-com3
//  need to turn the LCD on LCDON = 1
//  LCDSON allows the segments to be blanked good for blinking but
//  needs to be on to
//  display the LCD segments LCDSON = 1
//  Bit pattern required =  0111 1101 = 0x7d
//  page 26-22 of MSP430x4xx user manual
LCDACTL = 0x7d;
}
```

8. The assembly language program included with this laboratory experiment first initializes the LCD system by writing to LCDACTL, LCDAPCTL0, and LCDAVCTL0 registers. In addition, the initialization function writes zero to all the bits reserved for the LCD segments turning off all of the LCD display elements. Next, the program sets all of the bits allocated in the MSP430 memory reserved for the LCD segments. This results in all of the LCD display elements turning on. For example, to write the number 1 to the right most display a value of 0x06 is written to LCDM3 memory location turning segment elements b and c.

Finally, once all of the segments of the LCD are turned on the program blinks the green LED to indicate that the program is running.

```
;-------------------------------------------------------------------
; LCD Driver for the for MSP 430X4XXX experimenter board using
; Softbaugh LCD
; Davies book pg 259, 260
; setup a pointer to the area of memory of the TMS430 that points to
; the segments
; of the softbaugh LCD LCDM3 = the starting address
;-------------------------------------------------------------------
            .cdecls C,LIST,"msp430fg4618.h"  ; cdecls tells assembler
                                             ; to allow
                                             ; the device header file
;-------------------------------------------------------------------

; #LCDM3 is the start of the area of memory of the TMS430 that points
; to the segments
; of the softbaugh LCD LCDM3 = the starting address
; each of the seven segments for each display is store in memory
; starting at address LCDM3
; which is the right most seven segment of the LCD
; The bit order in each byte is
; dp, E, G, F, D, C, B, A    or
;  :, E, G, F, D, C, B, A
; after the seven segments these memory locations are used to turn on
; the special characters
; such as battery status, antenna, f1-f4, etc.
; there are 7 seven segment displays

;  data area ram starts 0x1100
;-------------------------------------------------------------------
;  the .sect directives are defined in lnk_msp430f4618.cmd
;                 .sect ".stack"      ; data ram for the stack
;                 .sect ".const"      ; data rom for initialized data
                                      ; constants
;                 .sect ".text"       ; program rom for code
;                 .sect ".cinit"      ; program rom for global inits
;                 .sect ".reset"      ; MSP430 RESET Vector
                  .sect ".sysmem"     ; data ram for initialized
                                      ; variables

; there are 11 locations that are needed for the softbaugh LCD
; only 7 used for the seven segment displays
LCD_SIZE    .byte 11                ; eleven bytes needed by the LCD

; This is the code area
; flash begins at address 0x3100

;-------------------------------------------------------------------
;   Main Code

;-------------------------------------------------------------------
            .text                               ; program start
            .global _START                         ; define entry point
;-------------------------------------------------------------------
START       mov.w   #300h,SP                    ; Initialize 'x1121
                                                ; stackpointer
StopWDT     mov.w   #WDTPW+WDTHOLD,&WDTCTL      ; Stop WDT
SetupP1     bis.b   #04h,&P2DIR                 ; P2.2 output

      ;  go initialize the LCD Display
            call  #Init_LCD
```

```
      ;  LCD_SIZE-4 only gives the 7 segment displays plus DP, and
      ;  colons (colons = dp)
      ;  Right most display is at LCDSeg[0];
      ;  R6 is a loop counter to cover all of the segments.  This count
      ;  counts up from 0
            mov.b #0x00, R6
      ;  R5 points to the beginning memory for the LCD
      ;  Turn on all of the segments
      ;  LCD_SIZE-4 only gives the 7 segment displays plus DP, and
      ;  colons:  colons = dp
      ;  Right most display is at LCDSeg[0];
      ;  To turn on a segment of the LCD a one is written in the
      ;  the appropriate location in the LCD memory
      ;  Setting all the bits to 1 for all memory locations turns on
      ;  all of the display elements
      ;  including all special characters
            mov.w #LCDM3, R5
      ;  move 0xff into R7 to turn on all LCD segments the LCD memory
            Mov.b #0xFF, R7
lpt1        mov.b R7, 0(R5)
   ;  Increment R5 to point to the next seven segment display
   ;  Increment R6 for the next count inthe loop
            inc.w R5
            inc.b R6
   ;  See if the loop is finished / finish writng to the  last display
   ;  element
            cmp.b LCD_SIZE, R6
            jnz lpt1
   ;  Blink the green LED to make sure the code is running
;
Mainloop    xor.b   #04h,&P2OUT             ; Toggle P2.2
Wait        mov.w   #0A000h,R15             ; Delay to R15
L1          dec.w   R15                     ; Decrement R15
            jnz     L1                      ; Delay over?
            jmp     Mainloop                ; Again

;-------------------------------------------------------------------
;                       Initialize the LCD system
;-------------------------------------------------------------------
Init_LCD
      ;   Using the LCD A controller for the MSP430fg4618
      ;   the pins of the LCD are memory mapped onto the mp430F4xxx
      ;   memory bus and
      ;   are accessed via LCDSeg[i] array
      ;   See page 260 of Davie's text
      ;   LCD_SIZE-4 only gives the 7 segment displays plus DP, and
      ;   (colons are the same bit setting)
      ;   LCD_SIZE-4 only gives the 7 segment displays plus DP, and
      ;   colons: colons / dp
      ;   Right most seven segment display is at LCDSeg[0];

      ;   Display format
      ;                    AAA
      ;                   F   B
      ;             X     F   B
      ;                    GGG
      ;             X     E   C
      ;                   E   C
      ;             DP     DDD
    ;   bit order
      ;   dp, E, G, F, D, C, B, A    or
      ;    :, E, G, F, D, C, B, A
      ;   initialize the segment memory to zero to clear the LCD
      ;   writing a zero in the LCD memory location clears turns off
```

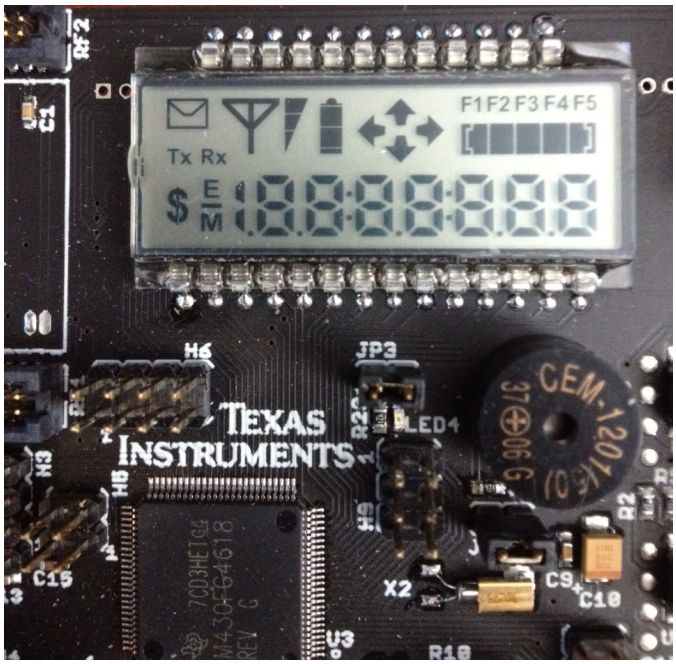
```
        ;   the LCD segment
        ;   R6 is a loop counter to cover all of the segments
        ;   including all special characters
              mov.b #0x00, R6
        ;   R5 points to the beginning memory for the LCD
              mov.w #LCDM3, R5
        ;   move 0 into R7 to clear the LCD memory
              mov.b #0x00, R7
lpt           mov.b R7, 0(R5)
        ;   Increment R5 to point to the next seven segment display
        ;   Increment R6 for the next count in the loop
              inc.w R5
              inc.b R6
        ;   See if the loop is finished
              cmp.b LCD_SIZE, R6
              jnz lpt
        ;   Port 5 ports 5.2-5.4 are connected to com1,com2,com3 of LCD
        ;   com0 fixed and already assigned
        ;   Need to assign com1 - com3 to port5
        ;   BIT4 | BIT3 |BIT2 = 1 P5.4, P.3, P5.2 = 1
              mov.b #0x1C, &P5SEL
        ;   Used the internal voltage for the LCD bit 4 = 0 (VLCDEXT=0)
        ;   internal bias voltage set to 1/3 of Vcc, charge pump
        ;   disabled,
        ;   page 26-25 of MSP430x4xx user manual
              mov.b #0x00, &LCDAVCTL0
        ;   LCDS28-LCDS0 pins LCDS0 = lsb and LCDS28 = MSB need
        ;   LCDS4 through LCDS24
        ;   from the experimenter board schematic the LCD uses S4-S24,
        ;   S0-S3 are not used here
        ;   Only use up to S24 on the LCD 28-31 not needed.
        ;   Also LCDACTL1 not required since not using S32 - S39
        ;   Davie's book page 260
        ;   page 26-23 of MSP430x4xx user manual
              mov.b #0x7E, &LCDAPCTL0
        ;   The LCD uses the ACLK as the master clock as the scan
        ;   rate for the display segments
        ;   The ACLK has been set to 32768 Hz with the external 327768 Hz
        ;   crystal
        ;   Let's use scan frequency of 256 Hz (This is fast enough not
        ;   to see the display flicker)
        ;   or a divisor of 128
        ;   LCDFREQ division(3 bits), LCDMUX (2 bits), LCDSON segments
        ;   on, Not used, LCDON LCD module on
        ;   011 = freq /128, 11 = 4 mux's needed since the display uses
        ;   for common inputs com0-com3
        ;   need to turn the LCD on LCDON = 1
        ;   LCDSON allows the segments to be blanked good for blinking
        ;   but needs to be on to
        ;   display the LCD segments LCDSON = 1
        ;   Bit pattern required =  0111 1101 = 0x7d
        ;   page 26-22 of MSP430x4xx user manual
              mov.b #0x7d, &LCDACTL
              ret

;-------------------------------------------------------------------------
;           Interrupt Vectors
;-------------------------------------------------------------------------
          .sect   ".reset"                ; MSP430 RESET Vector
          .short  START                   ;
         .end
```

Procedure: Part 1. C language program implementation of the MSP430FG4618 LCD Display

1. Read the C language program included with this laboratory experiment. Become familiar with the setting of the various registers used by the MSP430FG4618 to configure the LCD.

2. Build a CCS project using the steps described in experiment #1 using the C language program included with this laboratory.

3. Run the program and verify that all of the LED segments have turned on and that the yellow LCD is blinking.



4. Write a C language program to display the number 2 (segments a,b,d,e,g) on the <u>right</u> most seven segment display.

5. Write a C language program to display the number 2 (segments a,b,d,e,g) on the <u>left</u> most seven segment display.

6. Write a C language program that can display the numbers 0 – 9 and the letter A, b, c, d, E, and F. Please take note the number 6 and the letter b. The number six should have a top hat where the letter b does not. Have this program count from 0 to F and then repeat indefinitely.

7. Write a C language program that counts up from 0 to 999 in decimal and displays this count on the LCD display approximately every tenth of a second if SW1 is pressed on the experimenter board otherwise the counter counts down is SW2 is pressed and displays the count on the LCD. If neither of the switches are pressed then the count displayed on the LCD display is the last count.

Procedure: Part 2. An assembly language program implementation of the MSP430FG4618 LCD Display

1. Read the assembly language program included with this laboratory experiment. Become familiar with the setting of the various registers used by the MSP430FG4618 to configure the LCD.

2. Build a CCS project using the steps described in experiment #1 using the assembly language program included with this laboratory.

3. Run the program and verify that all of the LED segments have turned on and that the green LCD is blinking.

4. Write an assembly language program to display the number 2 (segments a,b,d,e,g) on the right most seven segment display.

5. Write an assembly language program to display the number 2 (segments a,b,d,e,g) on the left most seven segment display.

6. Write an assembly language program that can display the numbers 0 – 9 and the letter A, b, c, d, E, and F. Please take note the number 6 and the letter b. The number six should have a top hat where the letter b does not. Have this program count from 0 to F and then repeat indefinitely. Using a lookup table of the form is one method of implementing a binary to seven segment decoder:

```
.byte 0x5F,0x00,0x00,0x00,0x00
.byte 0x00,0x00,0x00,0x00,0x00
.byte 0x00,0x00,0x00,0x00,0x00,0x00
```

There are sixteen entries in the table one for each number to be displayed. In the table above, the first element is the bit pattern required to display the number 0 on the LCD display. The other entries in this table must be calculated and substituted for the zero values given.

7. Write an assembly language program that counts from up from 0 to 999 in decimal and displays this count on the LCD display approximately every tenth of a second if SW1 is pressed on the experimenter board otherwise the counter counts down is SW2 is pressed and displays the count on the. If neither of the switches are pressed then the count displayed on the LCD display is the last count.

Report

1. Include in the laboratory report the objective of the laboratory experiment.

2. The procedure used to generate and execute the C-language and assembly language programs (To be written in your own words not cut paste from the project).

3. The source code for the C-language language programs in Part 1 of the procedures (4 commented programs).

4. The source code for the assembly language programs in Part 2 of the procedures (4 commented programs).

5. A summary of the laboratory experiment and what you have learned.

EXPERIMENT #5

HEXADECIMAL CALCULATOR USING THE MSP430 EXPERIMENTER BOARD

Goals: To develop a C language and an assembly language program that adds, subtracts and multiplies two hexadecimal digits. The resulting answer is then displayed on the LCD display of the MSP430FG4618 experimenter board.

References: MSP430x4xx Family User's Guide, MSP430 Assembly Language Tools v 4.0 User's Guide, MSP430FG4618/F2013 Experimenter's Board, the help menu in code composer studio, MSP430 Microcontroller Basics by John H. Davies, and laboratory experiments #3 and #4.

Equipment: A desktop computer system running code composer, the MSP430FG4618 experimenter board and the MSP430 USB-Debug interface (MSP-FET430UIF).

Pre-laboratory: Read this experiment carefully to become familiar with the procedural steps in this experiment. Develop a first draft of the various programs that will add, subtract, and multiply two hexadecimal digits.

Discussion: Please refer to discussion content for both laboratory experiments #3 and #4. Review the programs required in experiment #3 that converts an ASCII hexadecimal number to a 4 bit binary number.

Procedure Part 1. C language programs.

1. Write a C language program that input from the keyboard two hexadecimal numbers via hyperterminal. Next these two numbers are to be displayed in the hyperterminal window and on the right most two digits of the LCD display on the experimenter board. To convert an ASCII "0"- "9" to a four bit number, 0x30 is subtracted from the ASCII number. To convert the ASCII capital letters "A"-"F" to a four bit number, 0x37 is subtracted from these capital letters. To help write this program, please use as reference the C language programs written for laboratory experiments #3 and #4.

2. Write a C language program that inputs two ASCII hexadecimal digits followed by the "+" symbol to indicate addition. Next, the user enters a second set of two hexadecimal digits that are to be added to the first two hexadecimal digits. The addition result is shown on both the hyperterminal display as well as the LCD display. Three digits are required for this resulting addition. For example, what should be displayed is:

FF + 02 = 101 (0xFF + 0x01 = 0x101)

This program should run indefinitely and start on a new line each time waiting for user input.

3. Write a C language program that inputs two ASCII hexadecimal digits followed by the "–" symbol to indicate subtraction. Next, the user enters a second set of two hexadecimal digits that are to be subtracted from the first two hexadecimal digits. The subtraction result is shown on both the hyperterminal display as well as the LCD display. Care must be taken when the second hexadecimal number is greater than the first so the proper sign is given. The best way to perform the subtraction is to subtract the smaller of the two numbers from the larger of the two numbers and display a negative sign if the second number is larger than the first number. For example, what should be displayed is:

01 – FF = –FE (0x01 – 0xFF = –0xFE)

This program should run indefinitely and start on a new line each time waiting for user input.

4. Write a C language program that inputs two ASCII hexadecimal digits followed by the "*" symbol to indicate multiplication. Next, the user enters a second set of two hexadecimal digits that are to be multiplied with the first two hexadecimal digits. The multiplication result is shown on both the hyperterminal display as well as the LCD display. Multiplication of two eight bit hexadecimal numbers can produce a result as large as sixteen bits. As such four hexadecimal digits should be used to display the result. For example what should be displayed is:

4E * A1 = 310E (0x4E * 0xA1 = 0x310E)

Care must be taken when defining a sixteen bit integer number in C. As default an integer number is defined as a sign number. The multiplication of two unsigned eight bit hexadecimal numbers produces a result that is a sixteen bit unsigned integer. The resulting variable should be defined using the unsigned descriptor:

unsigned int result;

The variable result is now in the range of 0x0000 to 0xFFFF. Finally, this program should run indefinitely and start on a new line each time waiting for user input.

5. Write a program that merges steps 2, 3, and 4 together and uses the ASCII symbols "+", "-", and "*" to determine which arithmetic operation to perform. This program should run indefinitely and start on a new line each time waiting for user input.

Procedure Part 2. Assembly language programs.

1. Write an assembly language program that inputs from the keyboard two hexadecimal numbers via hyperterminal. Next, these two numbers are displayed in the hyperterminal window and on the right most two digits of the LCD display on the experimenter board. To convert an ASCII "0"- "9" to a four bit number, 0x30 is subtracted from the ASCII number. To convert the ASCII capital letters "A"-"F" to a four bit number, 0x37 is subtracted from these capital letters. To help write this program, please use as reference the assembly language programs written for laboratory experiments #3 and #4.

2. Write an assembly language program that inputs two ASCII hexadecimal digits followed by the "+" symbol to indicate addition. Next, the user enters a second set of two hexadecimal digits that are to be added to the first two hexadecimal digits. The addition result is shown on both the hyperterminal display as well as the LCD display. Three digits are required for this resulting addition. For example, what should be displayed is:

FF + 01 = 100 (0xFF + 0x01 = 0x100)

This program should run indefinitely and start on a new line each time waiting for user input.

3. Write an assembly language program that inputs two ASCII hexadecimal digits followed by the "-" symbol to indicate subtraction. Next, the user enters a second set of two hexadecimal digits that are to be subtracted from the first two hexadecimal digits. The subtraction result is shown on both the hyperterminal display as well as the LCD display. Care must be taken when the second hexadecimal number is greater than the first so the proper sign is given. The best way to perform the subtraction is to subtract the smaller of the two numbers from the larger of the two numbers and display a negative sign if the second number is larger than the first number. For example, what should be displayed is:

01 – FE = –FD (0x01 – 0xFF = –0xFD)

This program should run indefinitely and start on a new line each time waiting for user input.

4. Write an assembly language program that inputs two ASCII hexadecimal digits followed by the "*" symbol to indicate multiplication. Next, the user enters a second set of two hexadecimal digits that are to be multiplied with the first two hexadecimal digits. The multiplication result is shown on both the hyperterminal display as well as the LCD display. The MSP430 does not have an assembly language multiplication instruction. For this project, the program needs to implement the shift and add multiplication

algorithm. For example to multiply 0x4E with 0xA1 the following shift and add algorithm is used:

```
0x4E                0100 1110
0xA1              1010 0001
                  0100 1110
                  0000 0000
                 0 0000 0000
                00 0000 0000
               000 0000 0000
              0 1001 1100 0000
             00 0000 0000 0000
            010 0111 0000 0000
            011 0001 0000 1110  = 0x310E
```

For example, what should be displayed is:

4E * A1 = 310E (0x4E * 0xA1 = 0x310E)

Finally, this program should run indefinitely and start on a new line each time waiting for user input.

5. Write a program that merges steps 2, 3, and 4 together and uses the ASCII symbols "+", "-", and "*" to determine which arithmetic operation to perform. This program should run indefinitely and start on a new line each time waiting for user input.

Report

1. Include in the laboratory report the objective of the laboratory experiment.

2. The procedures used to generate and execute the C-language and assembly language programs.

3. The source code for the C-language language programs in Part 1 of the procedures (5 commented programs).

4. The source code for the assembly language programs in Part 2 of the procedures (5 commented programs).

5. A summary of the laboratory experiment and what you have learned.

EXPERIMENT #6

INTERRUPTS USING THE MSP430FG4618 EXPERIMENTER BOARD

Goals: To develop C language and assembly language programs that uses the timer and interrupts to control the timed execution of an event. An interrupt service routine will be written to flash the LED's on the experimenter board at a given rate. Also a real time clock and a stop watch will be implemented.

References: MSP430x4xx Family User's Guide (Chapter 15), MSP430 Assembly Language Tools v 4.0 User's Guide, MSP430FG4618/F2013 Experimenter's Board, the help menu in code composer studio, MSP430 Microcontroller Basics by John H. Davies, and laboratory experiments #4 and #5.

Equipment: A desktop computer system running code composer, the MSP430FG4618 experimenter board and the MSP430 USB-Debug interface (MSP-FET430UIF).

Pre-laboratory: Read this experiment carefully to become familiar with the procedural steps in this experiment. Study in detail and become familiar with the C program and the assembly program provided with this lab. Develop an approach to implement the C language and assembly language programs required in Steps 1 - 5 of the procedures.

Discussion: The laboratory experiment will give the step by step procedures on creating a C language and an assembly language program that provide interfacing to the MSP430FG4618 timer functions used to generate a time controlled interrupt.

1. The timer (A) module in the MSP430FG4618 can be used to generate controlled timed events via the TAR timer (A) counter register. Depending on timer (A) mode of operation, the TAR can count up from 0x0000 to 0xFFFF. It can also be configured to count up from 0x0000 to a set value determined by the value stored in the TACCR0 register. The TACCR0 register is used to store a 16 bit number that is compared with the 16 bit number in the TAR counter register. When the TAR register equals the TACCR0 register, the count is then reset to zero and an interrupt from the timer (A) system can occur. In addition, the TAR register can be configured to count up to the value in the TACCR0 register and immediately count back down to zero. The timer (A) system can be synchronous to the processer clock or it can be configured as asynchronous using an external clock source such as the ACLK. Finally, the value in the TAR register can be cleared to zero by setting the TACLR bit in the TA0CTL / TACTL register. Chapter 15 of MSP430x4xx Family User's Guide gives a detailed discussion of the timer (A) system in the MSP430FG4618.

2. TA0CTL / TACTL Register: This register configures the mode and the clock input for the timer (A) system located in the MSP430FG4618. Bits 8 - 9 determine the clock source for the timer. For this laboratory experiment, the clock source will be derived from the ACLK (01) which is at 32768 Hz. Also for this project, a divide by 8 option (bits 6 - 7 = 11) will be used so as to reduce the input to the timer A system by a factor of 8. Since the ACLK is at 32768 Hz, this sets the clock input to the timer (A) system to 4096 Hz. There are four possible modes that timer (A) can be used with. The goal of this project is to generate an interrupt whenever the TAR register reaches a selected value store in TACCR0. As such, the timer (A) mode that must be selected is the count up mode to the value stored in TACCR0 register (bits 5 - 4 = 10). TACLR bit will be set to zero so as not to clear the TAR register and the TAIE the timer (A) interrupt enable bit will be set to one so an interrupt will be generated every time the TAR register reaches the value stored in TACCR0. For the options selected for this laboratory experiment, a value of 0x01D4 must be stored in the TA0CTL register.

TACTL, Timer_A Control Register

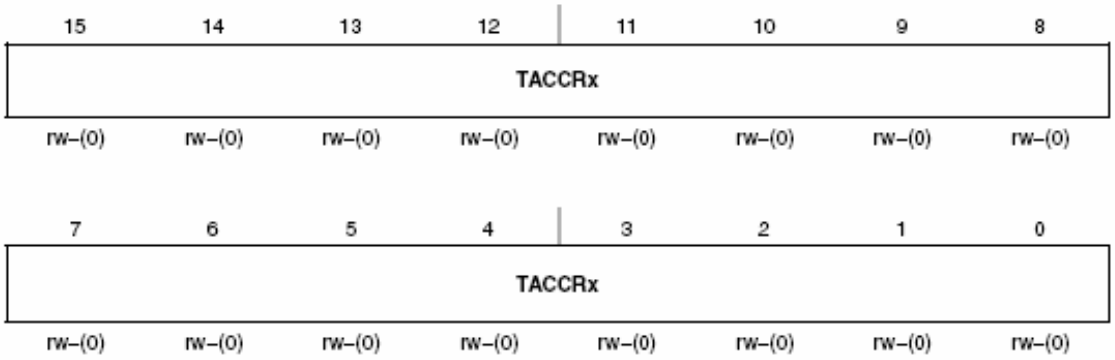
15	14	13	12	11	10	9	8
Unused						TASSELx	
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)

7	6	5	4	3	2	1	0
IDx		MCx		Unused	TACLR	TAIE	TAIFG
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	w-(0)	rw-(0)	rw-(0)

Unused	Bits 15-10	Unused
TASSELx	Bits 9-8	Timer_A clock source select 00 TACLK 01 ACLK 10 SMCLK 11 Inverted TACLK
IDx	Bits 7-6	Input divider. These bits select the divider for the input clock. 00 /1 01 /2 10 /4 11 /8
MCx	Bits 5-4	Mode control. Setting MCx = 00h when Timer_A is not in use conserves power. 00 Stop mode: the timer is halted 01 Up mode: the timer counts up to TACCR0 10 Continuous mode: the timer counts up to 0FFFFh 11 Up/down mode: the timer counts up to TACCR0 then down to 0000h
Unused	Bit 3	Unused
TACLR	Bit 2	Timer_A clear. Setting this bit resets TAR, the clock divider, and the count direction. The TACLR bit is automatically reset and is always read as zero.
TAIE	Bit 1	Timer_A interrupt enable. This bit enables the TAIFG interrupt request. 0 Interrupt disabled 1 Interrupt enabled
TAIFG	Bit 0	Timer_A interrupt flag 0 No interrupt pending 1 Interrupt pending

3. TACCRO Register: This register holds the 16-bit numeric value that is compared against the value in the TAR register. When the TAR register reaches the value stored in TACCR0 register, an interrupt is generated and TAR register is set to zero. The generation of this timer interrupt allows for an interrupt service function to be executed at a given rate. In this manner, time critical applications can be implemented. A one second interrupt rate for the interrupt service function with an input clock source to the timer A system of 4096 Hz (as described in Step 2 above) would require a value of 4096 (0x1000) be stored in the TACCR0 register. This would result in an interrupt being generated every 4096 counts or once a second. An interrupt rate of twice a second would require a numeric value of 2048 (0x800).

TACCRx, Timer_A Capture/Compare Register x



TACCRx	Bits	Timer_A capture/compare register.
	15-0	Compare mode: TACCRx holds the data for the comparison to the timer value in the Timer_A Register, TAR. Capture mode: The Timer_A Register, TAR, is copied into the TACCRx register when a capture is performed.

4. TACCTL0 Register: For this laboratory experiment, the only bit of interest in the TACCTL0 register is the CCIE bit. This bit enables the generation of an interrupt whenever the value stored in compare register TACCR0 equals the value in the TAR register. A one must be written into this bit location to turn on interrupts using the capture compare option in the timer (A) system. As such, a value of 0x0010 is written in to the TACCTL0 register.

TACCTLx, Capture/Compare Control Register

15	14	13	12	11	10	9	8
CMx		CCISx		SCS	SCCI	Unused	CAP
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	r	r0	rw-(0)

7	6	5	4	3	2	1	0
OUTMODx			CCIE	CCI	OUT	COV	CCIFG
rw-(0)	rw-(0)	rw-(0)	rw-(0)	r	rw-(0)	rw-(0)	rw-(0)

CMx	Bit 15-14	Capture mode 00 No capture 01 Capture on rising edge 10 Capture on falling edge 11 Capture on both rising and falling edges
CCISx	Bit 13-12	Capture/compare input select. These bits select the TACCRx input signal. See the device-specific data sheet for specific signal connections. 00 CCIxA 01 CCIxB 10 GND 11 V_{CC}
SCS	Bit 11	Synchronize capture source. This bit is used to synchronize the capture input signal with the timer clock. 0 Asynchronous capture 1 Synchronous capture
SCCI	Bit 10	Synchronized capture/compare input. The selected CCI input signal is latched with the EQUx signal and can be read via this bit.
Unused	Bit 9	Unused. Read only. Always read as 0.
CAP	Bit 8	Capture mode 0 Compare mode 1 Capture mode
OUTMODx	Bits 7-5	Output mode. Modes 2, 3, 6, and 7 are not useful for TACCR0 because EQUx = EQU0. 000 OUT bit value 001 Set 010 Toggle/reset 011 Set/reset 100 Toggle 101 Reset 110 Toggle/set 111 Reset/set

5. SR Register: Before the timer interrupt routine can be used, the general purpose interrupts within the MSP430 must be enabled by setting the General Interrupt Enable (GIE) bit in the Status Register (SR). In the C programming language, this is accomplished using the following function:

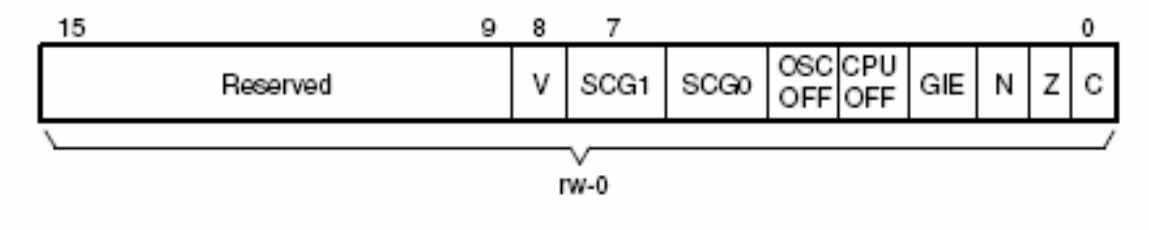
```
__enable_interrupt();
```

To disable interrupts the disable interrupt function is used:

```
__disable_interrupt();
```

Using assembly language, the SR register is loaded directly using the bit set (BIS) assembly instruction:

```
bis.w #0x0008,SR
```



6. The C language program included with this laboratory experiment first initializes the timer system by writing to TACCTL0, TA0CTL, and TACCR0 registers. Next, the GIE bit is set in the SR register. The last item the C language must provide is to set the timer interrupt vector to the address of the C language program interrupt service function. When a timer interrupt occurs, the timer interrupt vector holds the address of the location of where in memory the interrupt function to be executed is located. It is the responsibility of the C language program to load the address of interrupt service function into the timer interrupt vector. There are three items required to setup a timer interrupt function. First, the function must be defined as an interrupt function:

```
__interrupt void Time_ISR(void);
```

Next, the interrupt function needs to be associated with the timer interrupt vector:

```
#pragma vector = TIMERA0_VECTOR;  ,
```

and finally, the interrupt function itself is defined:

```
_interrupt void Timer_ISR(void)
   {
   // Simply blink the light using the ex-or operator
   P2OUT ^=0x02;
   }
```

In the example code above, this interrupt function simply turns the yellow LED on and off every time the interrupt function is called.

The following C language program gives an example on how to generate an interrupt at a fixed rate of time to flash the yellow LED on and off using the timer A system. The main body of the program is presently empty running forever using an endless for loop.

```
//*******************************************************************************
//  LED BLINK program using the timer A function and Interrupts
//     experimenter board RAM at 0x1100 - 0x30ff, FLASH at 0x3100 - 0xfbff
//     Port 2 is used for the LED's Port 2 bit 2 is the green LED,
//     Port 2 bit 1 is the yellow LED
//
//*******************************************************************************
//  must include the C header to get the predefined variable names
//
//-------------------------------------------------------------------------------
#include "msp430fg4618.h"
```

```
#include "intrinsics.h"
// include if input and output routines are used in c
#include "stdio.h"

__interrupt void Time_ISR(void);

int main(void)
  {
  // tell the compiler not to optimize the variable i volatile unsigned int i;
  WDTCTL = WDTPW + WDTHOLD;            // Stop watchdog timer
  P2DIR |= 0x02;                       // port 2 bit 1  to as an output
                                       // 1=output 0 = input
  P2OUT = 0x02;                        // go turn the LED ON

// TACCTL0 capture compare register
// need only to enable interrupts with this reg all other values are left 0
// using compare mode
// Bit 4 is the compare/capture enable interrupt bit 1 = enabled
// TACCTL0 = CCIE;
   TACCTL0 = 0x010;

// next the general purpose maskable interrupt bit GEI must be set in the SR
// register
// to turn on the maskable interrupts
// next must use an intrinsic function given in intrinsics.h
// to turn interrupt on use  __enable_interrupt();
// to turn interrupt off use  __disable_interrupt();
// TACTL =TA0CTL                 // timer a control register
// bits 8 and 9 set the clock input 00=TACLk, 01=ACLK(32768 Hz), 10=SMCLK(1
// MHz)
// bits 7 and 6 set the divider 00=/1, 01=/2, 10=/4, 11=/8
// bits 4 and 5 set the mode 00 stop, 01 up count from  0 to the value in the
// TACCR0 reg
// 10 counts from 0 to 0xffff, 1 up / down mode 0 to TACCR0 then to 0
// bit 3 unused bit 2 =1 clears counter, bit 1=1 enables interrupt
// bit 0 status Timer a interrupt flag = 1 interrupt pending
// ACLK 01, CLR timer = 1, /8, and count up mode
// in binary 0000 0001 1101 0100
// TA0CTL = MC_1 | ID_3 | TASSEL_1 | TACLR;
   TA0CTL = 0x01D4;

// TACCR0 counter value register
// ALCK = 32768 Hz
// for one second time t= (N*8) /32768   N=4096 = 0x1000 max count =6 5535
   TACCR0=0x1000;

// Go enable the interrupts now that the timer is setup as desired
 __enable_interrupt();

  for (;;){
// add main body of code here as required
  }
}

// must now setup the interrupt function in C need to point the function
```

```
// address to the
// timer interrupt vector located at 0xFFF2, this is done with the #pragma
// statement
#pragma vector = TIMERA0_VECTOR;
// the __interrupt tells C that the following function is an interrupt
// function
// the interrupt function must pull off the stack the SR reg
__interrupt void Timer_ISR(void)
  {
// Simply blink the yellow LED using the ex-or operator
      P2OUT ^=0x02;
  }
```

7. The equivalent assembly language program to initialize the timer A system is given below. The timer A system is initialized by writing to the TACCTL0, TA0CTL, and TACCR0 registers. Next, the GIE bit is set in the SR register using the move.w instruction. The last item required by the assembly language program is to set timer interrupt vector to the address of the assembly language program interrupt service function. This is accomplished using .sect assembler directive:

```
.sect    ".int22"
.word    TIMER_ISR
```

".int22" is the location in the interrupt table reserved for the timer interrupt and is defined in the *.cmd file and is included as one of the project files. ".int22" equates to the physical address of 0xFFEC for the MSP430FG4618. The .word assembler directive that follows the .sect assembler directive reserves two bytes in memory allocated to the address of the timer interrupt function. At assembly time, the assembler places in two these two bytes the address of the timer interrupt function given by the label name TIMER_ISR.

In the assembly language interrupt service function given below, the timer interrupt function ends with a return from interrupt instruction (reti) to indicate the end of assembly instructions associated with the interrupt service function:

```
; here's the timer interrupt function
; it simple flashers the green LED
TIMER_ISR
      xor.b #0x04, &P2OUT
      reti
```

This interrupt function example simply turns the green LED on and off every time the interrupt function is called.

The equivalent assembly language program to the C language program above is given below. This program initializes the timer A system and sets up the timer interrupt service function to turn on and off the green LED. The main body of the assembly program is presently empty running forever using an endless loop via a jmp instruction back onto itself.

```
;*******************************************************************************
;   LED BLINK program using the timer A function and Interrupts
;      experimenter board RAM at 0x1100 - 0x30ff, FLASH at 0x3100 - 0xfbff
;      Port 2 is used for the LED's Port 2 bit 2 is the green LED,
;      Port 2 bit 1 is the yellow LED
;
;*******************************************************************************
;-------------------------------------------------------------------------------
;  must include the C header to get the predefined variable names
;-------------------------------------------------------------------------------

            .cdecls C,LIST,"msp430fg4618.h"  ; cdecls tells assembler to allow
                                             ; the c header file
;-------------------------------------------------------------------------------
;  the .sect directives are defined in lnk_msp430f4618.cmd
;-------------------------------------------------------------------------------

;-------------------------------------------------------------------------------
;   Main Code
;-------------------------------------------------------------------------------

            .text                            ; program start
            .global _START                   ; define entry point

;-------------------------------------------------------------------------------
START       mov.w   #0x300,SP                ; Initialize 'x1121 stackpointer
StopWDT     mov.w   #WDTPW+WDTHOLD,&WDTCTL   ; Stop WDT
SetupP2     bis.b   #0x04,&P2DIR             ; port 2 bit 2  to as an output
                                             ; 1=output 0 = input

Mainloop   mov.b   #0x04,&P2OUT              ; set P2.2 =1 turn LED on

; TACTL = TA0CTL timer a control register
; bits 8 and 9 set the clock input 00=TACLk, 01=ACLK(32768 Hz), 10=SMCLK (1
; MHz)
; bits 7 and 6 set the divider 00=/1, 01=/2, 10=/4, 11=/8
; bits 4 and 5 set the mode 00 stop, 01 up count from  0 to the value in the
; TACCR0 reg
; 10 counts from 0 to 0xffff, 1 up / down mode 0 to TACCR0 then to 0
; bit 3 unused bit 2 = 1 clears counter, bit 1 = 1 enables interrupt
; bit 0 status Timer a interrupt flag = 1 interrupt pending
; ACLK 01, CLR timer = 1, divide 8, and count up mode
; in binary 0000 0001 1101 0100
; TACTL = MC_1 | ID_3 | TASSEL_1 | TACLR |TAIE;
;                   TACTL = 0x01D4;
                    mov.w #0x1D4, &TACTL

; TACCR0 counter value register
; ALCK = 32768 Hz
; for one second time t= (N*8) /32768   N=4096 = 0x1000 max count =65535
; TACCR0=49999;
                    mov.w   #0x1000,&TACCR0
; TACCTL0 capture compare register
; need only to enable interrupts with this reg all other values are left 0
; using compare mode
```

```
; Bit 4 is the compare/capture enable interrupt bit 1 = enabled
; TACCTL0 = 0x010;
; TACCTL0 = CCIE;
                    mov.w #0x0010,&TACCTL0

; next the general purpose maskable interrupt bit GEI must be set in the SR
register
; to turn on the maskable interrupts.
; go enable the interrupts in the SR register GIE = bit 3 =  0x0008
                    bis.w #0x0008,SR
;                   bis.w #GIE,SR

; main body of code goes here
; just a loop for now
lpaq                jmp lpaq

; here's the timer interrupt function
; it simply flashes the green LED
TIMER_ISR           xor.b #0x04, &P2OUT
                    reti

;-------------------------------------------------------------------------------
;           Interrupt Vectors
;-------------------------------------------------------------------------------
; to load and interrupt vector the .sect assembler diretive must be used
; the .sect directives are defined in lnk_msp430f4618.cmd
; the various interrupt vectors are defined at the end of the msp430fg4618.h
; header file for TIMERA0 the vector address is 0xFFEC.
; From the lnk_msp430f4618.cmd file this corresponds to int22

                    .sect   ".int22"
                    .word   TIMER_ISR
                    .sect   ".reset"          ; MSP430 RESET Vector
                    .short  START
                    .end
```

Procedure: Part 1. C language program implementation of interrupts using the timer included with the MSP430FG4618

1. Read the C language program included with this laboratory experiment. Become familiar with the setting of the various registers for the timer used by the MSP430FG4618.

2. Build a CCS project using the steps described in experiment #1 using the C language program included with this laboratory.

3. Run the program and verify that the yellow LCD is blinking.

4. Change the timer TACCR0 register so that the interrupt function is executed every two seconds instead of every one second.

5. Write a C language program using the results of laboratory experiments #4 and #5 that implements a 12 hour real time clock using the LCD display unit on the MSP430FG4618 experimenter board. The clock should include hours, minutes, and seconds:

11:55:23

Every second, the LCD display should be updated with the new time. The LCD display should be called from the timer interrupt service function to update the various seven segment displays based upon a set of time variables. The following gives the pseudo code to increment the various time variables. For example, whenever the second tenths variable reaches 6, this variable is reset to zero and the minute units variable is incremented by 1.

```
su = second units
st = second tenths
mu = minute units
mt = minute tenths
h =  hours
hu = hour units
ht = hour tenths
su++;
if(su == 10)
  {su = 0; st ++;}
if(st == 6)
     {st = 0; mu++}
if(mu == 10)
     {mu = 0; mt++}
if(mt == 6)
     {mt = 0; h++}
 if(h == 13)
  {h = 1;}
hu = h;
ht = 0;
if(h == 10);
  {hu = 0; ht =1);
if(h == 11);
   {hu = 1; ht =1);
if(h == 12);
   {hu = 2; ht =1);
```

Procedure: Part 2. An assembly language program implementation of the MSP430FG4618 LCD Display

1. Read the assembly language program included with this laboratory experiment. Become familiar with the setting of the various registers for the timer used by the MSP430FG4618.

2. Build a CCS project using the steps described in experiment #1 using the assembly language program included with this laboratory.

3. Run the program and verify that the green LCD is blinking.

4. Change the timer TACCR0 register so that the interrupt function is executed every two seconds instead of every one second.

5. Write an assembly language program that implements a 4 digit stopwatch "9999" using the two switches SW1 and SW2 on the MSP430FG4618 experimenter board. Laboratory #1 used these two switches to turn on the two LEDS. When SW1 is pressed, the stopwatch starts counting up at the rate of 0.1 seconds and when SW2 is pressed, the stopwatch is reset to zero. The best place to poll the status of the SW1 and SW2 is in the main loop and using a flag along with a counter variable in the interrupt service routine. If SW1 is pressed during a timer interrupt, the counter is incremented by 1. Also, in the timer interrupt service function, the update of the LCD display should occur.

Report

1. Include in the laboratory report the objective of the laboratory experiment.

2. The procedure used to generate and execute the C-language and assembly language programs (To be written in your own words not cut paste from the project).

3. The source code for the C-language language programs in Part 1 of the procedures (1 commented program: the real time clock).

4. The source code for the assembly language programs in Part 2 of the procedures (1 commented program: the stopwatch).

5. A summary of the laboratory experiment and what you have learned.

EXPERIMENT #7
ANALOG TO DIGITAL CONVERSION USING THE MSP430FG4618 EXPERIMENTER BOARD

Goals: To develop C language and assembly language programs that uses the 12-bit analog to digital converter in the MSP430FG4618 to read the analog voltage from a LM34 temperature sensor. The value read from the temperature sensor will them be converted to a binary-coded-decimal number and displayed on the LCD display of the experimenter board representing the temperature in degrees Fahrenheit.

References: MSP430x4xx Family User's Guide (Chapter 28), MSP430 Assembly Language Tools v 4.0 User's Guide, MSP430FG4618/F2013 Experimenter's Board, the help menu in code composer studio, MSP430 Microcontroller Basics by John H. Davies, and laboratory experiments #4 and #5.

Equipment: A desktop computer system running code composer, the MSP430FG4618 experimenter board and the MSP430 USB-Debug interface (MSP-FET430UIF).

Pre-laboratory: Read this experiment carefully to become familiar with the procedural steps in this experiment. Study in detail and become familiar with the C language and the assembly language programs provided with this lab. Develop an approach to implement the C language and assembly language programs required in Steps 1 - 5 of the procedures.

Discussion: The laboratory experiment will give the step by step procedures on creating a C language and an assembly language program that provide interfacing to the MSP430FG4618 analog to digital A(/D) functions used to read the LM temperature sensor connected to the A1 analog input pin.

1. The A/D section of the MSP430FG4618 is a 12-bit A/D that converts an analog voltage between the two voltages of ref- and ref+. Ref+ and ref- is user selectable and can be derived from an internal voltage reference or from an externally supplied voltage source. A part of the A/D system is a sample and hold circuit that samples the input voltage and holds this voltage value for the A/D during the analog-to digital conversion process. The input to this sample and hold circuit is a multiplexer that selects one of eight analog inputs (A0-A8) (the number of analog inputs depends on the actual MSP430 selected). The A/D conversion process can be a single channel or it can be a sequence of channels starting at a selected channel and ending at another selected channel. An A/D conversion can be started via an interrupt or via software control. In non-interrupt mode, a single bit is used by software to start an A/D conversion (ADC12SC). Setting this bit to a one starts an A/D conversion. A part of the timing for the A/D a master clock is used

and can be set to be one of the several clocks available for the MSP430FG4618. The result of an A/D conversion result are then stored in the MSP430FG4618 memory starting at ADC12MEM0.

2. ADC12CTL0 Register: There are two control registers used to configure the A/D converter system on the MSP430FG4618. ADC12CTL0 register sets the timing as well as the reference voltage source used by the A/D system. Before any of the registers can be set, it is required that the ADC12ON bit be set to zero to make sure that the A/D system is off. This is easily accomplished by writing a zero to ADC12CTL0. For this laboratory experiment, the sample and hold number of clock cycles will be set to 4 (b15 - b12 = 0000) and the number of A/D conversion clock cycles will be set to 64 (b11 – b8 = 0100). Since the goal of the A/D conversion for this laboratory experiment is to convert a single channel (A1), the MSC bit should be set to 0 for a single A/D conversion. The internal reference source will be used for the A/D voltage reference input and the 2.5 volt reference value will be selected. This reference needs to be turned on by setting the REFON bit to 1. Finally, the A/D system should be turned back on ADC12ON = 1 (b7 – b4 = 0111). Since this laboratory experiment will not be using interrupts, both the ADC12VIE and the ADC12TOVIE bits should be set to zero. Until all of the A/D system registers have been configured, the ENC and the ADC12SC bit should be set to zero. This sets bits b15 – b0 as 0x470.

ADC12CTL0, ADC12 Control Register 0

15	14	13	12	11	10	9	8
SHT1x				SHT0x			
rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)

7	6	5	4	3	2	1	0
MSC	REF2_5V	REFON	ADC12ON	ADC12OVIE	ADC12 TOVIE	ENC	ADC12SC
rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)

Modifiable only when ENC = 0

SHT1x Bits 15-12 Sample-and-hold time. These bits define the number of ADC12CLK cycles in the sampling period for registers ADC12MEM8 to ADC12MEM15.

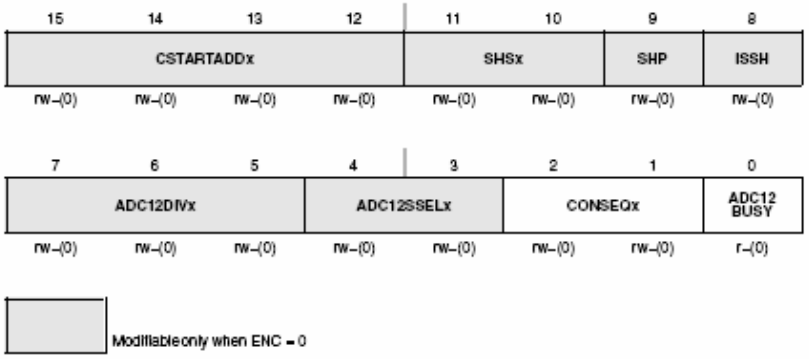
SHT0x Bits 11-8 Sample-and-hold time. These bits define the number of ADC12CLK cycles in the sampling period for registers ADC12MEM0 to ADC12MEM7.

SHTx Bits	ADC12CLK cycles
0000	4
0001	8
0010	16
0011	32
0100	64
0101	96
0110	128
0111	192
1000	256
1001	384
1010	512
1011	768
1100	1024
1101	1024
1110	1024
1111	1024

MSC	Bit 7	Multiple sample and conversion. Valid only for sequence or repeated modes. 0 The sampling timer requires a rising edge of the SHI signal to trigger each sample-and-conversion. 1 The first rising edge of the SHI signal triggers the sampling timer, but further sample-and-conversions are performed automatically as soon as the prior conversion is completed.
REF2_5V	Bit 6	Reference generator voltage. REFON must also be set. 0 1.5 V 1 2.5 V
REFON	Bit 5	Reference generator on 0 Reference off 1 Reference on
ADC12ON	Bit 4	ADC12 on 0 ADC12 off 1 ADC12 on
ADC12OVIE	Bit 3	ADC12MEMx overflow-interrupt enable. The GIE bit must also be set to enable the interrupt. 0 Overflow interrupt disabled 1 Overflow interrupt enabled
ADC12 TOVIE	Bit 2	ADC12 conversion-time-overflow interrupt enable. The GIE bit must also be set to enable the interrupt. 0 Conversion time overflow interrupt disabled 1 Conversion time overflow interrupt enabled
ENC	Bit 1	Enable conversion 0 ADC12 disabled 1 ADC12 enabled
ADC12SC	Bit 0	Start conversion. Software-controlled sample-and-conversion start. ADC12SC and ENC may be set together with one instruction. ADC12SC is reset automatically. 0 No sample-and-conversion-start 1 Start sample-and-conversion

3. ADC12CTL1 Register: This register is the second control register used by the MSP430FG4618 to configure the A/D converter. Bits b15 – b12 select the memory location where the A/D conversion results will be stored. For this laboratory experiment, it is desired to have the conversion results stored in memory location ADC12MEM0 (b15 – 12 = 0000). Bits b11 – 19 select the timing control source for the sample and hold circuitry and the A/D converter. The goal of this laboratory experiment is to use the ASC12SC bit to control the start an A/D conversion. As such, the SHSX bits should both be set to zero. The SHP bit tells the A/D converter where to obtain its sampling signal from and should be set to 0 so that the source is directly from the sampled input. The input signal should not be inverted (ISSH = 0) and the ADC12 clock divider should be set to a divide by 1. This set bits b15 - b0 to 0x0000.

ADC12CTL1, ADC12 Control Register 1



CSTART ADDx	Bits 15-12	Conversion start address. These bits select which ADC12 conversion-memory register is used for a single conversion or for the first conversion in a sequence. The value of CSTARTADDx is 0 to 0Fh, corresponding to ADC12MEM0 to ADC12MEM15.
SHSx	Bits 11-10	Sample-and-hold source select 00 ADC12SC bit 01 Timer_A.OUT1 10 Timer_B.OUT0 11 Timer_B.OUT1
SHP	Bit 9	Sample-and-hold pulse-mode select. This bit selects the source of the sampling signal (SAMPCON) to be either the output of the sampling timer or the sample-input signal directly. 0 SAMPCON signal is sourced from the sample-input signal. 1 SAMPCON signal is sourced from the sampling timer.
ISSH	Bit 8	Invert signal sample-and-hold 0 The sample-input signal is not inverted. 1 The sample-input signal is inverted.
ADC12DIVx	Bits 7-5	ADC12 clock divider 000 /1 001 /2 010 /3 011 /4 100 /5 101 /6 110 /7 111 /8
ADC12 SSELx	Bits 4-3	ADC12 clock source select 00 ADC12OSC 01 ACLK 10 MCLK 11 SMCLK
CONSEQx	Bits 2-1	Conversion sequence mode select 00 Single-channel, single-conversion 01 Sequence-of-channels 10 Repeat-single-channel 11 Repeat-sequence-of-channels
ADC12 BUSY	Bit 0	ADC12 busy. This bit indicates an active sample or conversion operation. 0 No operation is active. 1 A sequence, sample, or conversion is active.

4. ADC12MEM0 – ADC12MEM15 registers: These 16 registers are used to store the analog to digital conversion results from the 16 A/D inputs. For the MSP430FG618 experimenter board analog inputs A0-A8 are available on header J8 and use ADCMEM0 – ADCMEM7. The 12 bit conversion result is stored in the lower 12 bits of this 16 bit number with the upper 4 bits always set to zero.

ADC12MEMx, ADC12 Conversion Memory Registers

15	14	13	12	11	10	9	8
0	0	0	0	Conversion Results			
r0	r0	r0	r0	rw	rw	rw	rw

7	6	5	4	3	2	1	0
Conversion Results							
rw	rw	rw	rw	rw	rw	rw	rw

Conversion Results	Bits 15-0	The 12-bit conversion results are right-justified. Bit 11 is the MSB. Bits 15-12 are always 0. Writing to the conversion memory registers will corrupt the results.

5. ADC12MCTL0 register: This register selects the type of A/D conversion that will be performed either a sequence of channels or a single channel, the A/D reference source, and the input channel(s) used. Since this laboratory experiment needs a single analog input channel of A1, the EOS bit should be set to 0 to indicate not an end of a sequence and the INCHX bits should be set to 0001 for channel A1. Next, the A/D reference voltage input must be selected to use the internal 2.5 volt reference. This is accomplished by setting Vr+ to VREF+ and Vr- to AVSS (ground) (SREFX bits set to 001). Combining these bits together gives b7 - b0 = 0x11.

ADC12MCTLx, ADC12 Conversion Memory Control Registers

7	6	5	4	3	2	1	0
EOS	SREFx			INCHx			
rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)

Modifiable only when ENC = 0

EOS	Bit 7	End of sequence. Indicates the last conversion in a sequence. 0 Not end of sequence 1 End of sequence
SREFx	Bits 6-4	Select reference 000 $V_{R+} = AV_{CC}$ and $V_{R-} = AV_{SS}$ 001 $V_{R+} = V_{REF+}$ and $V_{R-} = AV_{SS}$ 010 $V_{R+} = Ve_{REF+}$ and $V_{R-} = AV_{SS}$ 011 $V_{R+} = Ve_{REF+}$ and $V_{R-} = AV_{SS}$ 100 $V_{R+} = AV_{CC}$ and $V_{R-} = V_{REF-}$/ Ve_{REF-} 101 $V_{R+} = V_{REF+}$ and $V_{R-} = V_{REF-}$/ Ve_{REF-} 110 $V_{R+} = Ve_{REF+}$ and $V_{R-} = V_{REF-}$/ Ve_{REF-} 111 $V_{R+} = Ve_{REF+}$ and $V_{R-} = V_{REF-}$/ Ve_{REF-}
INCHx	Bits 3-0	Input channel select 0000 A0 0001 A1 0010 A2 0011 A3 0100 A4 0101 A5 0110 A6 0111 A7 1000 Ve_{REF+} 1001 V_{REF-}/Ve_{REF-} 1010 Temperature sensor 1011 $(AV_{CC} – AV_{SS}) / 2$ 1100 $(AV_{CC} – AV_{SS}) / 2$, A12 on 'FG43x and 'FG461x devices 1101 $(AV_{CC} – AV_{SS}) / 2$, A13 on 'FG43x and 'FG461x devices 1110 $(AV_{CC} – AV_{SS}) / 2$, A14 on 'FG43x and 'FG461x devices 1111 $(AV_{CC} – AV_{SS}) / 2$, A15 on 'FG43x and 'FG461x devices

6. ADC12IE register: This register is used to enable interrupts for each of the 16 analog inputs. An interrupt can be set to occur anytime one of these sixteen A/D conversions has been completed. For this laboratory experiment, the goal is to use software to start the A/D conversion (ADC12SC =1). As such this register should be set to zero.

ADC12IE, ADC12 Interrupt Enable Register

15	14	13	12	11	10	9	8
ADC12IE15	ADC12IE14	ADC12IE13	ADC12IE12	ADC12IE11	ADC12IE10	ADC12IE9	ADC12IE8
rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)

7	6	5	4	3	2	1	0
ADC12IE7	ADC12IE6	ADC12IE5	ADC12IE4	ADC12IE3	ADC12IE2	ADC12IE1	ADC12IE0
rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)

ADC12IEx	Bits 15-0	Interrupt enable. These bits enable or disable the interrupt request for the ADC12IFGx bits. 0 Interrupt disabled 1 Interrupt enabled

7. Finally, since all of the A/D registers are configured, it is now OK to turn on the A/D system by setting the ENC bit to a one and setting the ASC12SC bit to 0 in the ADC12CTL0 register (ADC12CTL0 = ADC12CTL0 'or' with 0x02). An A/D conversion will now occur anytime the ADC12SC bit goes from a zero to a one. This is accomplished by first setting the ADC12SC bit to zero (ADC12CTL0 = ADC12CTL0 'and" 0xFFFE) and then setting it to a one (ADC12CTL0 = ADC12CTL0 'or' 0x0001).

8. A C program example is given below that uses the timer interrupt to sample the analog input A1 at a rate of 10 times a second. Also given in the timer interrupt routine, is a comment on where to place code for the LCD display function to display either the 12 bit or the upper 8 bit A/D conversion result. The variable nc holds the most significant eight bits of the 12 bit A/D conversion result. As a final note, Port 6 which is also the A/D input has been set to an input to minimize leakage current between the digital output circuitry of Port 6 and the A/D input pins (A0 – A7).

```
//-------------------------------------------------------------------
// A/D conversion using the MSP 430FG4618
// experimenter board.
// The A/D is set for single sample mode and
// is started by the timer (A) interrupt.  It uses the timer interrupt
// to time the A/D conversions
//-------------------------------------------------------------------

#include "msp430fg4618.h"
#include "stdio.h"
#include "intrinsics.h"

void Init_Timer(void);

volatile unsigned int n=0;     // need a variable to store the A/D
conversion
volatile unsigned char nc=0;  // need a 8 bit variable
```

```
int main(void)
    {
      volatile unsigned char a;
      volatile unsigned int i;      // volatile to prevent optimization

      WDTCTL = WDTPW + WDTHOLD;     // Stop watchdog timer
      P2DIR |= 0x02;                // Set P1.0 to output direction

      // to eliminate any leakage current from the digital
      // port buffer port 6
      // which is used for the A/D inputs should be made as an
      // input port
      // make all of port six inputs
      P6DIR = 0x00;
      // go initialize the timer for a timed interrupt
      Init_Timer();
      // Set the ADC12 to sample single-channel conversion mode
      // Initialize the state of the ADC control register to zero
      // the ENC bit ADC12CTL0 bit 1 must be zero to set
      // many of the A/D
      // control register values
      ADC12CTL0 =0x0000;
      // There are two control registers the must be
      // set ADC12CTL0 and ADC12CTL1
      // bits 15 -12 set the number of clocks cycles for
      // the sample and hold time
      // this program will use 64 clock cycles
      // (bits 15 - 12 not used here so set to 0000)
      // bits 11 - 8 set the clock cycle required for
      // the A/D result registers
      // ADC12MEM0 - ADC12MEM15 (bits 11 - 8 = 0100) Set to
      // 64 clock cycles
      // MSC bit 7 only used for multiply samples not
      // used here so set to zero
      // REF2_5V bit 6 set to a 2.5 volts bit 6 = 1
      // The internal reference REFON bit 5 set to on bit 5 = 1
      // A/D system on ASC12ON bit 4 = 1
      // ADC120VIE bit 3 not used set to zero bit 3 = 0
      // ADC 12TOVIE bit 2 not used set to zero bit 2 = 0
      // ENC bit 1 set to zero to configure the registers
      // ENC set to 1 to enable A/D conversion
      // ADC12SC bit 0 to 1 to start A/D conversion
      // ADC12SC must be set to zero to stop A/D conversion
      // 15 - 0
      // 0000 0100 0111 0000
      ADC12CTL0 = 0x0470;
      // ADC12CTL1
      // Bits 15 - 12 start A/D results memory address
      // address 0 - 15 corresponding
      // ADC12MEM0 - ADC12MEM15
      // Bits 11 - 10 sample and hold clock source
      // set to 00 for software control via ADC12SC bit
      // bits 9 Sample and hold input signal
      // set to 0 for source is from the sample input signal
      // bit 8 the input can be inverted
      // bit 8 set to 0 for no inversion
      // bits 7 - 5 A/D conversion clock rate divider from
      // the input clock source
      // Set to 000 for divide by 1 the A/D clock freq =
      // the clock source freq
      // bit 4 - 3  sets the ADC12 clock source
      // set to 00 to use the ADC12OSC use the ADC oscillator
```

```
      // bits 2 - 1 type of conversion single/ multiple
      // set to 00 for single channel conversion
      // bit 0 is a flag indicating id the A/D is busy ADC12BUSY
      // Bits 15 - 0
      // 0000 00 0 0 000 00 00 0
      // Setup to start at ADC12MEM0
      ADC12CTL1 = 0;

      // ADC12MCTL0 register is used to select the reference
      // input and the start channel in the sequence.  For the
      // program A1 will be in the input
      // the reference will be the internal VREF and ground (AVSS)
      // bit 7 indicate last conversion set to zero
      // bits 4 - 6 set VREF = 001 for internal reference
      // bits 0 - 3 set input channel to A1 = 0001
      // ADC12MCTL0 =  bits 7 - 0 = 0 001 0001 = 0x011
      ADC12MCTL0 = 0x11;
      // The ADC2IE register sets the interrupt enable for each of the
      //16 inputs A0-A15.  0 = disabled, 1 = enabled
      ADC12IE = 0x0000;   // All interrupts disable
      // Now that the all of the A/D registers have been
      // configured go ahead and enable the A/D converter
      // system by setting ENC to 1 (ADC12CTL0 bit 1)
      ADC12CTL0 |=  0x02;
      // go start the A/D first conversion to be read by the
      // next timer interrupt
      ADC12CTL0 |= 0x01;
    for (;;)
        {
          // this is the main program loop
        }
    }

//-------------------------------------------------------------------
//          This is the timer interrupt service routine
//-------------------------------------------------------------------
// must now setup the interrupt function in C need to point the
// function address to the timer interrupt vector located at 0xFFF2,
// this is done with the #pragma statement
#pragma vector = TIMERA0_VECTOR;
    // the __interrupt tells C that the following function is an
interrupt
    // function
    // the interrupt function must pull off the stack the SR reg
__interrupt void Timer_ISR(void)
    {
      // Go get the 12 bit binary value from the A/D conversion
     // need to get data from the start ADC memory location of ADC12MEM0
     n = ADC12MEM0;
    // Need to set the ADC12SC bit to zero so another A/D
    // conversion can be
    // started at the end of this timer interrupt routine by setting
ADC12SC to 1
    // ADC12SC is located in ADC12CTL0 bit 0
     // Need to leave all the other bits alone
     ADC12CTL0 &= 0xFFFE;
     // Need to get the upper eight bits on the result is a 12 bit
     // number eliminate the lower 4 bits and use the upper 8 bits
    nc = (unsigned char)(n>>4);
     // add LCD display routine here
    // Go start a new A/D conversion that will be completed the next
    // time this interrupt routine is called
     ADC12CTL0 |= 0x01;
     // Simply blink the yellow LED using the ex-or operator
```

```
    // to tell the the timer routine is working
    P2OUT ^=0x02;
   }

//-------------------------------------------------------------------
//                      Initialize the Timer A system
//-------------------------------------------------------------------
void Init_Timer(void)
      {
      // TACCTL0 capture compare register
      // need only to enable interrupts with this reg all other
      // values are left 0
      // using compare mode
      // Bit 4 is the compare/capture enable interrupt bit 1 = enabled
      // TACCTL0 = CCIE;
      TACCTL0 = 0x010;
      // next the general purpose maskable interrupt bit GEI must
      //be set in the SR
      // register
      // to turn on the maskable interrupts
      // next must use an intrinsic function given in intrinsics.h
      // to turn interrupt on use __enable_interrupt();
      // to turn interrupt off use __disable_interrupt();
      // TACTL =TA0CTL              // timer a control register
      // bits 8 and 9 set the clock input 00=TACLK,
      // 01=ACLK(32768 Hz), 10=SMCLK(1
      // MHz)
      // bits 7 and 6 set the divider 00=/1, 01=/2, 10=/4, 11=/8
      // bits 4 and 5 set the mode 00 stop, 01 up count from
      // 0 to the value in the
      // TACCR0 reg
      // 10 counts from 0 to 0xffff, 1 up / down mode 0 to
      // TACCR0 then to 0
      // bit 3 unused bit 2 =1 clears counter, bit 1=1
      // enables interrupt
      // bit 0 status Timer a interrupt flag = 1 interrupt pending
      // ACLK 01, CLR timer = 1, /8, and count up mode
      // in binary 0000 0001 1101 0100
      // TA0CTL = MC_1 | ID_3 | TASSEL_1 | TACLR;
      TA0CTL = 0x01D4;
      // TACCR0 counter value register
      // ALCK = 32768 Hz
      // for one second time t= (N*8) /32768   N=4096 =
      // 0x1000 max count =6 5535
      // for 1/4 of a second  N=0x400
      TACCR0=0x400;
      // Go enable the interrupts now that the timer is
      // setup as desired
      __enable_interrupt();
      }
```

8. An assembly language program example is given below that uses the timer interrupt to sample analog input A1 at a rate of 10 times a second. Also given in the timer interrupt routine is where to place code to display on the LCD either the 12 bit or the upper 8 bit A/D conversion result. The register R7 holds the most significant eight bits of the 12 bit A/D conversion result. As a final note, Port 6, which is also the A/D inputs, has been set to an input to minimize leakage current between the digital output circuitry of Port 6 and the A/D input pins (A0 – A7).

```
;-------------------------------------------------------------------
; A/D conversion using the MSP 430FG44618
; experimenter board.
```

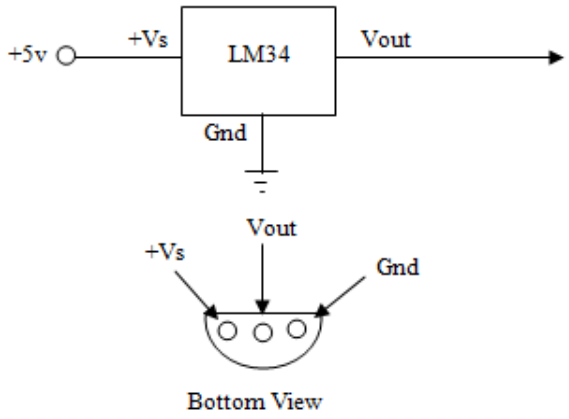
```
; The A/D is set for single sample mode
; and is started by the timer (A) interrupt.  It uses the timer
; interrupt to time the A/D conversions
;-------------------------------------------------------------------------

            .cdecls C,LIST,"msp430fg4618.h"  ; cdecls tells
                                             ; assembler to allow
                                             ; the device header file
;-------------------------------------------------------------------------
;  data area ram starts 0x1100
;-------------------------------------------------------------------------
;
;  the .sect directives are defined in lnk_msp430f4618.cmd
;                  .sect ".stack"      ; data ram for the stack
;                  .sect ".const"      ; data rom for initialized data
                                       ; constants
;                   .sect ".text"      ; program rom for code
;                  .sect ".cinit"      ; program rom for global inits
;                  .sect ".reset"      ; MSP430 RESET Vector

                   .sect ".sysmem"    ; data ram for initialized
                                      ; variables

; This is the code area
; flash begins at address 0x3100
;-------------------------------------------------------------------------
;   Main Code
;-------------------------------------------------------------------------
            .text                     ; program start
            .global _START            ; define entry point
;-------------------------------------------------------------------------
START       mov.w   #0x300,SP         ; Initialize 'x1121 stackpointer
StopWDT     mov.w   #WDTPW+WDTHOLD,&WDTCTL   ; Stop WDT
SetupP1     bis.b   #0x04,&P2DIR              ; P2.2 output

; go initialize the Timer
                  call    #Init_Timer

; to eliminate any leakage current from the digital port buffer port 6
; which is used for the A/D inputs should be made as an input port
; make all of port six inputs
                  mov.b #0x00, &P6DIR
; Set the ADC12 to sample single-channel conversion mode
; Initialize the state of the ADC control register to zero
; the ENC bit ADC12CTL0 bit 1 must be zero to set many of the A/D
; control register values
                  mov.w #0x0000, &ADC12CTL0
; There are two control registers the must be set ADC12CTL0
; and ADC12CTL1 bits 15 -12 set the number of clocks cycles for the
; sample and hold time
; this program will use 64 clock cycles
; (bits 15 - 12 not used here so set to 0000)
; bits 11 - 8 set the clock cycle required for the A/D result registers
; ADC12MEM0 - ADC12MEM15 (bits 11 - 8 = 0100) Set to 64 clock cycles
; MSC bit 7 only used for multiply samples not used here so set to zero
; REF2_5V bit 6 set to a 2.5 volts bit 6 = 1
; The internal reference REFON bit 5 set to on bit 5 = 1
; A/D system on ASC12ON bit 4 = 1
; ADC120VIE bit 3 not used set to zero bit 3 = 0
; ADC 12TOVIE bit 2 not used set to zero bit 2 = 0
; ENC bit 1 set to zero to configure the registers
; ENC set to 1 to enable A/D conversion
; ADC12SC bit 0 to 1 to start A/D conversion
; ADC12SC must be set to zero to stop A/D conversion
```

```
; 15 - 0
; 0000 0100 0111 0000
                  mov.w #0x0470, &ADC12CTL0
; ADC12CTL1
; Bits 15 - 12 start A/D results memory address 0 - 15
; corresponding ADC12MEM0 - ADC12MEM15
; Bits 11 - 10 sample and hold clock source
; set to 00 for software control via ADC12SC bit
; bits 9 Sample and hold input signal
; set to 0 for source is from the sample input signal
; bit 8 the input can be inverted
; bit 8 set to 0 for no inversion
; bits 7 - 5 A/D conversion clock rate divider from the input
; clock source
; Set to 000 for divide by 1 the A/D clock freq = the clock source freq
; bit 4 - 3  sets the ADC12 clock source
; set to 00 to use the ADC12OSC use the ADC oscillator
; bits 2 - 1 type of conversion single/ multiple
; set to 00 for single channel conversion
; bit 0 is a flag indicating id the A/D is busy ADC12BUSY
; Bits 15 - 0
; 0000 00 0 0 000 00 00 0
; Setup to start at ADC12MEM0
                  mov.w #0x0000, &ADC12CTL1
; ADC12MCTL0 register is used to select the reference input and the
; start channel in the sequence.  For the program A1 will be
; in the input the reference will be the internal VREF
; and ground (AVSS)
; bit 7 indicate last conversion set to zero
; bits 4 - 6 set VREF = 001 for internal reference
; bits 0 - 3 set input channel to A1 = 0001
; ADC12MCTL0 =  bits 7 - 0 = 0 001 0001 = 0x11
                  mov.b #0x11, &ADC12MCTL0
; The ADC2IE register sets the interrupt enable for each of the
; 16 inputs A0-A15.  0 = disabled, 1 = enabled
                  mov.w #0x0000, &ADC12IE
; All interrupts disable
; Now that the all of the A/D registers have been configured
; go ahead and
; and enable the A/D converter system by setting ENC to 1
; (ADC12CTL0 bit 1)
                  bis.w  #0x0002,   &ADC12CTL0
; go start the A/D first conversion to be read by the next timer
; interrupt
                  bis.w #0x0001, &ADC12CTL0
;
Mainloop
; Main body of the program code goes here
            jmp     Mainloop                ; do it again

;-------------------------------------------------------------------
;                       The timer ISR is routine goes here
;-------------------------------------------------------------------
; here's the timer interrupt function
; it simply flashes the green LED
TIMER_ISR   xor.b #0x04, &P2OUT
; Go get the 12 bit binary value from the A/D conversion
; need to get data from the start ADC memory location of ADC12MEM0
; go store it in register R7
; n = ADC12MEM0;
            mov.w &ADC12MEM0, R7
; Need to set the ADC12SC bit to zero so another A/D conversion can be
; started at the end of this timer interrupt routine by
; setting ADC12SC to 1 ADC12SC is located in ADC12CTL0 bit 0
```

```
; Need to leave all the other bits alone
; ADC12CTL0 &= 0xFE;
                  and.w #0xFFFE, &ADC12CTL0
; go shift over R7 by 4 bits to the right to get the most
; significant 8 bits of
; the A/D conversion
                  rra.w R7
                  rra.w R7
                  rra.w R7
                  rra.w R7
; Add code here to display the upper 8 bits of the result
; go start the A/D first conversion to be read by the next timer
; interrupt
                  bis.w #0x0001, &ADC12CTL0
                  reti

;-------------------------------------------------------------------
;                       Initialize the timer system
;-------------------------------------------------------------------
Init_Timer
; TACTL = TA0CTL timer a control register
; bits 8 and 9 set the clock intput 00=TACLk,
; 01=ACLK(32768 Hz), 10=SMCLK (1 MHz)
; bits 7 and 6 set the divider 00=/1, 01=/2, 10=/4, 11=/8
; bits 4 and 5 set the mode 00 stop, 01 up count from  0 to the
; value in the TACCR0 reg
; 10 counts from 0 to 0xffff, 1 up / down mode 0 to TACCR0 then to 0
; bit 3 unused bit 2 = 1 clears counter, bit 1 = 1 enables interrupt
; bit 0 status Timer a interrupt flag = 1 interrupt pending
; ACLK 01, CLR timer = 1, divide 8, and count up mode
; in binary 0000 0001 1101 0100
; TACTL = MC_1 | ID_3 | TASSEL_1 | TACLR |TAIE (TACTL = 0x01D4)
                  mov.w #0x1D4, &TACTL
; TACCR0 counter value register
; ALCK = 32768 Hz
; for one second time t= (N*8) /32768   N=4096 = 0x1000
; max count =65535 for 1/4 second N = 0x0400
                  mov.w   #0x0400,&TACCR0
; TACCTL0 capture compare register
; need only to enable interrupts with this reg all other
; values are left 0 using compare mode
; Bit 4 is the compare/capture enable interrupt bit 1 = enabled
; TACCTL0 = 0x010;
; TACCTL0 = CCIE;
                  mov.w #0x0010,&TACCTL0
; next the general purpose maskable interrupt bit GEI must be
; set in the SR register to turn on the maskable interrupts.
; go enable the interrupts in the SR register GIE = bit 3 =  0x0008
                  bis.w #0x0008,SR
;                 bis.w #GIE,SR
                  ret
                                          ;
;-------------------------------------------------------------------
;           Interrupt Vectors
;-------------------------------------------------------------------
; to load and interrupt vector the .sect assembler directive
; must be used the .sect directives are defined in lnk_msp430f4618.cmd
; the various interrupt vectors are defined at the end of the
; msp430fg4618.h header file for TIMERA0 the vector address is 0xFFEC.
; From the lnk_msp430f4618.cmd file this corresponds to int22

                  .sect   ".int22"
                  .word   TIMER_ISR
                  .sect   ".reset"        ; MSP430 RESET Vector
```

```
.short   START
.end
```

9. The LM34 is configured on the experimenter board as shown below and is connected to analog A/D input A1.



The LM34 produces an output voltage of 0.01 volts per degrees Fahrenheit and the A/D produces 2048 counts (12 bit) in the voltage range of 0 to 2.5 volts (Vr+ = VREF+ = 2.5 volts). Since only the most significant 8 bits will be used, this is equivalent to 2.5 volts per 256 counts. To convert the binary count from the A/D to degrees Fahrenheit, the following conversion must be used:

Degrees Fahrenheit = (A/D conversion result) * 2.5 volts / 255 counts * 1 / (.01 degrees Fahrenheit / volt)

Degrees Fahrenheit = (A/D conversion result) * 0.98039215 Degrees Fahrenheit / count

For example consider a temperature of 78 degree Fahrenheit. The A/D conversion will return a result of about 80 (decimal) counts after shifting the twelve bit result by 4 bits to the right to obtain the most significant 8 bits. To perform the conversion from A/D counts to temperature in degrees Fahrenheit, a binary multiplication is needed. Since the MSP430 does not do floating point arithmetic operations, the floating point number must be represented by an integer number. The binary point for the coefficient of 0.98039215 could be moved to the right by 8 bits. Then an integer multiplication could be performed and then the binary point of this multiplication could then be relocated 8 bits to the left of the most right bit. This is essentially multiplying 0.98039215 by 256 = 250.98 (or the closest integer value of 251 = 0xFB). Consider the closest binary representation for 0.98039215.

0.98039215	D7	D6	D5	D4	D3	D2	D1	D0	
.	1	1	1	1	1	0	1	1	0xFB

and the 8-bit A/D result of 80 counts.

20 decimal	D7	D6	D5	D4	D3	D2	D1	D0	
.	0	1	0	1	0	0	0	0	0x50

The multiplication of these two 8 bit numbers (FB times 50) yields a 16 bit result of 0x4E70. Remembering to place back the radix (binary) point 8 bits to the left of the most right bit gives a final value of 0x4E.70. The integer value of this result gives the integer value of the temperature for the LM34 in degrees Fahrenheit (0x4E in decimal is 78). The conversion from A/D counts to degrees Fahrenheit requires the multiplication of two 8 bit integer numbers and then using the upper eight bits of the 16 bit multiplication result.

10. To convert a binary number to a binary-coded-decimal (BCD) number, the shift left and add three algorithm is used (/http://en.wikipedia.org/wiki/Double_dabble). Consider converting an 8 bit number to a BCD number. Since the maximum value for an 8 bit number is 255, 3 BCD digits will be required (12 bits). The algorithm starts with all zeros loaded into the resulting 3 BCD numbers (all 12 bits). The binary value is shifted to the left one bit at a time. Anytime any one of the 4 bit BCD digits are greater or equal to 5 a value of 3 (0011) is added to this BCD digit. After 8 shifts, the 8 bit binary number has been converted to a 3 digit BCD number. Below is a table that gives an example for converting the 8 bit binary number of 0xFF to 3 BCD digits (12 bits) using the shift left and add three algorithm. Anytime the resulting BCD digit is greater or equal to 5, a value of 3 is added to this BCD digit. After eight shifts, the three BCD digits are 2, 5, and 5, which is the decimal value for 0xFF = 255.

B	C	D	1	B	C	D	2	B	C	D	3	B	I	N	A	R	Y		
												1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	shl
										1	1	1	1	1	1	1	1		shl
									1	1	1	1	1	1	1	1			shl
0	0	0	0	0	0	0	0	0	0	1	1								add
								1	0	1	0	1	1	1	1	1			rslt
							1	0	1	0	1	1	1	1	1				shl
0	0	0	0	0	0	0	0	0	0	1	1								add
							1	1	0	0	0	1	1	1	1				rslt
						1	1	0	0	0	1	1	1	1					shl
					1	1	0	0	0	1	1	1	1						shl
0	0	0	0	0	0	1	1	0	0	0	0								add
0	0	0	0	1	0	0	1	0	0	1	1	1	1						rslt
			1	0	0	1	0	0	1	1	1	1							shl
0	0	0	0	0	0	0	0	0	0	1	1								add
0	0	0	1	0	0	1	0	1	0	1	0	1							rslt
0	0	1	0	0	1	0	1	0	1	0	1								
2				5				5				2	5	5	b				

Below is a pseudo C language program that converts an eight bit binary number to three BCD digits. The program loops through all 256 values of the variable I converting each of these values to the three BCD digits of h, t, and u. The final result is the BCD conversion result which is printed on the screen using the prinf function.

```
#include <cstdlib>
#include <iostream>
#include <stdio.h>

using namespace std;

int main(int argc, char *argv[])
{
    unsigned int u,t,h;
    unsigned int i,j,num, temp;
    // scan through all 256 values to be converted to BCD
    for(i = 1;i <= 255;i++){
      num = 0;
      temp = i;
      for(j = 0;j <= 7;j++){
          temp = temp <<1;
          num = num + ((temp & 0x100) >> 8);
          u = num & 0x00f;
          t = num & 0x0f0;
          h = num & 0xf00;
          if(u >= 5)
             num = num + 0x003;
           if(t >= (5 << 4))
             num = num +0x030;
          if(h >= (5 << 8))
             num = num + 0x300;
          num = num <<1;
          }
          num = num >>1;
          u = u;
          t = t >> 4;
          h = h >> 8;
          printf("\n%i %i%i%i\n",i,h,t,u);
          }
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Procedure: Part 1. C language program to display the temperature of the LM34 on the LCD display of MSP430FG4618 experimenter board

1. Read the C language program included with this laboratory experiment. Become familiar with the setting of the various registers for the A/D converter used by the MSP430FG4618.

2. Using the C language example for setting up the A/D converter, the LCD programs written for laboratory experiment #4, and the pseudo C language program to convert a binary number to BCD, write a C language program that displays on the LCD display of the MSP430FG4618 experimenter board the temperature of the LM34 temperature sensor in degrees Fahrenheit. Do not forget to use only the most significant 8 bits of the 12 bit converter result.

3. Build a CCS project using the steps described in laboratory 1 using the C language program developed in Step 2.

4. Run the program and verify it displays the correct temperature in the range of 70 to 80 degrees Fahrenheit. Touch the LM34 temperature sensor with your fingers and verify the temperature on the LCD display increases in value.

Procedure: Part 2. Assembly language program to display the temperature of the LM34 on the LCD display of MSP430FG4618 experimenter board

1. Read the assembly language program included with this laboratory experiment. Become familiar with the setting of the various registers for the A/D converter used by the MSP430FG4618.

2. Using the assembly language example for setting up the A/D converter, the LCD programs written for laboratory experiment #4, and the pseudo C language program to convert a binary number to BCD, write an assembly language program that displays on the LCD display of the MSP430FG4618 experimenter board the temperature of the LM34 temperature sensor in degrees Fahrenheit. Do not forget to use only the most significant 8 bits of the 12 bit converter result. Since the MSP430 does not have a multiply instruction, the assembly language multiply routine written for laboratory project #5 should be used for the multiplication required to scale the A/D converter result.

3. Build a CCS project using the steps described in laboratory 1 using the assembly language program developed in Step 2.

4. Run the program and verify it displays the correct temperature in the range of 70 to 80 degrees Fahrenheit. Touch the LM34 temperature with your fingers and verify the temperature on the LCD display increases in value.

Report

1. Include in the laboratory report the objective of the laboratory experiment.

2. The procedure used to generate and execute the C-language and assembly language programs (To be written in your own words not cut paste from the project).

3. The source code for the C-language language program in Part 1 of the C language procedures (1 commented program).

4. The source code for the assembly language program in Part 2 of the assembly language procedures (1 commented program).

5. A summary of the laboratory experiment and what you have learned.