CPE 325: Embedded Systems Laboratory
Laboratory Tutorial #6:
MSP430 Interrupts and Universal Clock Subsystem

Aleksandar Milenković
Email: milenka@uah.edu
Web: http://www.ece.uah.edu/~milenka

Objective:

This tutorial will teach you how to write interrupt service routines in assembly and C/C++ programming languages. In addition, it will describe the universal clock subsystem of the MSP430F5529 device that is responsible for generating internal clocks. You will learn the following topics:

Using interrupts in C/assembly (specifically working with port interrupts)
The clock subsystem and clock configuration
Working with the TI experimenter's board

Notes:

All previous tutorials are required for successful completion of this lab, especially, the tutorials introducing the MSP-EXP430F5529LP board and the Code Composer Studio software development environment.

Contents

1 Interfacing Switches and LEDs in Assembly (Polling and Interrupts)

In the handout for Laboratory #3 we learned how to interface with the MSP-EX430F5529LP hardware, specifically LEDs and switches, using C language. We will redo the same examples using the MSP430 assembly language.

1.1 Toggling LEDs in Assembly Language

Figure 1 shows the assembly code of the blink application (Lab6_D1.asm). Here is a brief description of the assembly code for this application. In addition to the portions of the code that were discussed in the previous labs we can discuss some new additions. The .text is a segment control assembler directive that controls how code and data are located in memory. .text is used to mark the beginning of a relocatable code. The linker can recognize any other type of segment (e.g., __STACK_END for code stack). Our main loop that flashes the LEDs starts at the InfLoop label. The code starting at the label SWDelay1 implements the software delay to make sure the LEDs blink at the appropriate interval. To exactly calculate the software delay we need to know the instruction execution time and the clock cycle time. The register R15 is loaded with 65,535 (the maximum unsigned integer that can fit in a 16-bit register). The dec.w instruction takes 1 clock cycle to execute, and jnz L1 takes 2 clock cycles to execute (note: this can be determined by enabling and reading the value of the clock in CCS). The nop instruction takes 1 clock cycle. The number of nop instructions in the loop is determined so that the total number of clocks in the SWDelay1 loop is 16. Determining clock cycle time requires in-depth understanding of the FLL-Clock module of the MSP430 which is discussed later in this tutorial. We note that the processor clock frequency is 1,048,576 Hz (2^{20} Hz) for the default configuration. The total delay is thus $65{,}535*16/2^{20}$ ~ 1s. Note: nop instructions are often used in creating software delays because they do not affect the state of the registers and take exactly one clock cycle to execute.

```
;-------------------------------------------------------------------------------
;   File:        Lab6_D1.asm
;   Description: The program toggles LEDs periodically.
;                LED1 is initialized off, LED2 is initalized on.
;                Main program loop:
;                   the SWDelay1 loop creates 1s delay before
;                   toggling the LEDs (ON/OFF).
;
;   Clocks:      ACLK = 32.768kHz, MCLK = SMCLK = default DCO = 2^20=1,048,576 Hz
;   Platform:    TI EXP430F5529LP Launchpad
;
;                 MSP430xF5529
;              -----------------
;          /|\|                 |
;           | |                 |
;           --|RST              |
;             |             P1.0|-->LED1(RED)
;             |             P4.7|-->LED2(GREEN)
;
;   Author:     Aleksandar Milenkovic, milenkovic@computer.org
```

```
;   Date:        September 14, 2018
;   Modified:   Prawar Poudel, August 08, 2019
;-------------------------------------------------------------------------------
; MSP430 Assembler Code Template for use with TI Code Composer Studio
;
;
;-------------------------------------------------------------------------------
            .cdecls C,LIST,"msp430.h"       ; Include device header file

;-------------------------------------------------------------------------------
            .def    RESET                   ; Export program entry-point to
                                            ; make it known to linker.
;-------------------------------------------------------------------------------
            .text                           ; Assemble into program memory.
            .retain                         ; Override ELF conditional linking
                                            ; and retain current section.
            .retainrefs                     ; And retain any sections that have
                                            ; references to current section.

;-------------------------------------------------------------------------------
RESET       mov.w   #__STACK_END,SP         ; Initialize stackpointer
StopWDT     mov.w   #WDTPW|WDTHOLD,&WDTCTL  ; Stop watchdog timer

SETUP:
            bis.b   #0x01,&P1DIR            ; set P1.0 as output, 0'b0000 0001
            bis.b   #0x80,&P4DIR            ; set P4.7 as output, 0'b1000 0000

            bic.b   #0x01,&P1OUT            ; turn P1.0 OFF
            bis.b   #0x80,&P4OUT            ; turn P4.7 ON
;-------------------------------------------------------------------------------
; Main loop here
;-------------------------------------------------------------------------------
InfLoop:
            mov.w #0xFFFF,R5    ; move 0xFFFF to R5 which will be out counter
SWDelay1:
            nop
            nop
            nop
            nop
            nop
            nop
            nop
            nop
            nop                 ; 13 NOPs + extra 3cc is a delay of 16cc
            nop                 ; so the total delay is 65535*16cc/2^20 ~ 1s
            nop
            nop
            nop
            dec.w R5                        ; 1cc
            jnz SWDelay1                    ; 2cc
            xor.b #0x01,&P1OUT              ; toggle 1.0
            xor.b #0x80,&P4OUT              ; toggle 4.7
            jmp InfLoop                     ; go to InfLoop
            nop
```

```

;-------------------------------------------------------------------------------
; Stack Pointer definition
;-------------------------------------------------------------------------------
            .global __STACK_END
            .sect   .stack

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

```

Figure 1. Blinking the LEDs in Assembly Language

1.2 Interfacing Switches in Assembly Language (Polling)

Figure 2 shows assembly program that interfaces S1 and LED1. S1 is connected to P1.BIT0 (ports are configured by default as input) and LED1 is connected to P2.BIT2 (should be configured as a digital output). BIT0 of P1 is checked. If pressed a logic 0 should be detected in P1IN.BIT0; otherwise it should read as a logic 1. When a press is detected, a software delay of 20 ms is implemented to support de-bouncing of the switch. If the switch is still pressed, the program turns on LED1. The program continually checks whether the switch is still pressed. If a release (depress) is detected, LED1 is turned off.

```
;-------------------------------------------------------------------------------
;   File:        Lab6_D2.asm
;   Description: The program demonstrates Press/Release using S1 and LED1.
;                LED1 is initialized off.
;                When an S1 press is detected, a software delay of 20 ms
;                is used to implement debouncing. The switch is checked
;                again, and if it's on, LED1 is turned on until S1 is released.
;
;   Clocks:      ACLK = 32.768kHz, MCLK = SMCLK = default DCO = 2^20=1,048,576 Hz
;   Platform:    TI EXP430F5529LP Launchpad
;
;                 MSP430F5529
;             -----------------
;         /|\|                 |
;          | |                 |
;          --|RST              |
;            |             P1.0|-->LED1(REd)
;            |             P2.1|<--SW1
;
;   Author:     Aleksandar Milenkovic, milenkovic@computer.org
;   Date:       September 14, 2018
;   Modified:   Prawar Poudel, August 08, 2019
;-------------------------------------------------------------------------------

            .cdecls C,LIST,"msp430.h"       ; Include device header file

;-------------------------------------------------------------------------------
```

```
            .def    RESET                   ; Export program entry-point to
                                            ; make it known to linker.
;-------------------------------------------------------------------------------
            .text                           ; Assemble into program memory.
            .retain                         ; Override ELF conditional linking
                                            ; and retain current section.
            .retainrefs                     ; And retain any sections that have
                                            ; references to current section.

;-------------------------------------------------------------------------------
RESET:      mov.w   #__STACK_END,SP         ; Initialize stack pointer
StopWDT:    mov.w   #WDTPW|WDTHOLD,&WDTCTL  ; Stop watchdog timer
;-------------------------------------------------------------------------------
SetupP2:
            bis.b   #001h, &P1DIR           ; Set P1.0 to output
                                            ; direction (0000_0001)
            bic.b   #001h, &P1OUT           ; Set P1OUT to 0x0000_0001 (ensure
                                            ; LED1 is off)

            bic.b   #002h, &P2DIR           ; SET P2.1 as input for SW1
            bis.b   #002h, &P2REN           ; Enable Pull-Up resister at P2.1
            bis.b   #002h, &P2OUT           ; required for proper IO set up

ChkSW1:     bic.b   #001h, &P1OUT
            bit.b   #002h, &P2IN             ; Check if SW1 is pressed
                                            ; (0000_0010 on P1IN)
            jnz     ChkSW1                  ; If not zero, SW1 is not pressed
                                            ; loop and check again
Debounce:
            mov.w   #2000, R15              ; Set to (2000 * 10 cc = 20,000 cc)
SWD20ms:    dec.w   R15                     ; Decrement R15
            nop
            nop
            nop
            nop
            nop
            nop
            nop
            jnz     SWD20ms                 ; Delay over?
            bit.b   #00000010b, &P2IN       ; Verify SW1 is still pressed
            jnz     ChkSW1                  ; If not, wait for SW1 press

LEDon:      bis.b   #001h, &P1OUT           ; Turn on LED1
SW1wait:    bit.b   #002h, &P2IN            ; Test SW1
            jz      SW1wait                 ; Wait until SW1 is released
            bic.b   #001h, &P1OUT            ; Turn off LED1
            jmp     ChkSW1                  ; Loop to beginning
            nop

;-------------------------------------------------------------------------------
; Stack Pointer definition
;-------------------------------------------------------------------------------
            .global __STACK_END
            .sect   .stack

```

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

Figure 2. Turn on LED1 when S1 is Pressed (Lab6_D2.asm)

1.3 Interfacing Switches in Assembly Language (Interrupt Service Routine)

With microcontrollers, it is often useful to be able to use interrupts in our programs. An interrupt allows an automatic break from the current program flow based on a set of conditions. Some of the I/O ports on the MSP430 have an interrupt capability that you can configure. When the interrupt conditions are met, the program execution departs into a service routine that handles the interrupt event. Once service routine is completed the control transfers back to the main program where it left off using a RETI (return from interrupt) instruction. We will learn more about interrupts in a subsequent lab, but you should understand how interrupt vectors are used and what interrupts do. To set up an interrupt for an I/O port, we have to perform a few tasks:

- Enable global interrupts in the status register
- Enable interrupts to occur for the particular events by setting control bits in corresponding registers associated with ports P1 or P2
- Specify whether the interrupt is triggered on a falling edge or rising edge
- Initialize the interrupt flag by clearing it

An example of using interrupts to interface the switches of the MSP430 experimenter board is shown in Figure 3. The main program configures ports, enables the global interrupts (GIE bit is SR is set), enables interrupt from BIT1 of Port1 (P1IE=0x0000_0010b). As pressing a switch corresponds to having input signal from a logic ‘1’ to a logic ‘0’, the interrupt arises when a falling edge is detected at P1IN.BIT1. The interrupt service routine starts at label SW2_ISR. The state of the input is checked; if P1IN.BIT1 is not a logic 0 we exit the ISR; otherwise, de-bouncing is performed. If SW2 is still pressed after 20 ms, LED1 is turned on. The program then waits for SW2 to be released. Note lines 94 and 95 that initialize the IVT entry 47 reserved for Port 1.

```
1   ;-------------------------------------------------------------------------------
2   ;   File:        Lab6_D3.asm
3   ;   Description: The program demonstrates Press/Release using S2 and LED1.
4   ;                LED1 is initialized off. The main program enables interrupts
5   ;                from P1.BIT1 (S2) and remains in an infinite loop doing nothing.
6   ;                P1_ISR implements debouncing and waits for a S2 to be released.
7   ;
8   ;   Clocks:      ACLK = 32.768kHz, MCLK = SMCLK = default DCO = 2^20=1,048,576 Hz
9   ;   Platform:    TI EXP430F5529LP Launchpad
10  ;
11  ;                 MSP430F5529
```

```
;             -----------------
;         /|\|                 |
;          | |                 |
;          --|RST              |
;            |             P1.0|-->LED1(RED)
;            |             P1.1|<--S2
;
;   Author:     Aleksandar Milenkovic, milenkovic@computer.org
;   Date:       September 14, 2018
;   Modified:   Prawar Poudel, Auguest 8, 2019
;-------------------------------------------------------------------------------
            .cdecls C,LIST,"msp430.h"       ; Include device header file

;-------------------------------------------------------------------------------
            .def    RESET                   ; Export program entry-point to
                                            ; make it known to linker.
            .def    S2_ISR
;-------------------------------------------------------------------------------
            .text                           ; Assemble into program memory.
            .retain                         ; Override ELF conditional linking
                                            ; and retain current section.
            .retainrefs                     ; And retain any sections that have
                                            ; references to current section.

;-------------------------------------------------------------------------------
RESET:      mov.w   #__STACK_END, SP        ; Initialize stack pointer
StopWDT:    mov.w   #WDTPW|WDTHOLD, &WDTCTL ; Stop watchdog timer
;-------------------------------------------------------------------------------
Setup:
            bis.b   #001h, &P1DIR           ; Set P1.0 to output
                                            ; direction (0000_0001)
            bic.b   #001h, &P1OUT           ; Set P1OUT to 0x0000_0001

            bic.b   #002h, &P1DIR           ; SET P1.1 as input for S2
            bis.b   #002h, &P1REN           ; Enable Pull-Up resister at P1.1
            bis.b   #002h, &P1OUT           ; required for proper IO set up


            bis.w   #GIE, SR                ; Enable Global Interrupts
            bis.b   #002h, &P1IE            ; Enable Port 1 interrupt from bit 1
            bis.b   #002h, &P1IES           ; Set interrupt to call from hi to low
            bic.b   #002h, &P1IFG           ; Clear interrupt flag
InfLoop:
            jmp     $                       ; Loop here until interrupt

;----------------------------------------------------------------------------
; P1_0 (S2) interrupt service routine (ISR)
;----------------------------------------------------------------------------
S2_ISR:
            bic.b   #002h, &P1IFG           ; Clear interrupt flag
ChkSW2:     bit.b   #02h, &P1IN             ; Check if S2 is pressed
                                            ; (0000_0010 on P1IN)
            jnz     LExit                   ; If not zero, SW is not pressed
                                            ; loop and check again
Debounce:   mov.w   #2000, R15              ; Set to (2000 * 10 cc )
```

```
SWD20ms:    dec.w   R15                     ; Decrement R15
            nop
            nop
            nop
            nop
            nop
            nop
            nop
            jnz     SWD20ms                 ; Delay over?
            bit.b   #00000010b,&P1IN        ; Verify S2 is still pressed
            jnz     LExit                   ; If not, wait for S2 press
LEDon:      bis.b   #001h,&P1OUT            ; Turn on LED1
SW2wait:    bit.b   #002h,&P1IN             ; Test S2
            jz      SW2wait                 ; Wait until S2 is released
            bic.b   #001,&P1OUT             ; Turn off LED1
LExit:      reti                            ; Return from interrupt
;-------------------------------------------------------------------------------
; Stack Pointer definition
;-------------------------------------------------------------------------------
            .global __STACK_END
            .sect   .stack

;-------------------------------------------------------------------------------
; Interrupt Vectors
;-------------------------------------------------------------------------------
            .sect   ".reset"        ; MSP430 RESET Vector
            .short  RESET
            .sect   ".int47"        ; PORT1_VECTOR,
            .short  S2_ISR         ; please check the MSP430F5529.h header file
            .end

```

Figure 3. Press/Release Using Port 1 ISR (Lab6_D3.asm)

2 Interfacing Switches and LEDs Using Interrupts in C

Figure 4 shows a C program that turns LED1 on when S2 is pressed and turns LED1 off when S2 is released. The main program configures and initializes ports, configures interrupts, and enters an infinite loop where the program waits for S2 to be released to turn off LED1. P1_ISR is entered upon detection of the switch press; the code inside clears P1.IFG1 and turns on LED1. Please note C convention to indicate that Port1_ISR corresponds to PORT1_VECTOR in the interrupt vector table.

```
/******************************************************************************
 *   File:        Lab6_D4.c
 *   Description: The program detects when S2 is pressed and turns on LED1.
 *                LED1 is kept on as long as S2 is pressed.
 *                P1_ISR is used to detect when S2 is pressed.
 *                Main program polls S2 and turns off when a release is detected.
 *   Board:       MSP-EXP430F5529LP Launchpad
 *   Clocks:      ACLK = 32.768kHz, MCLK = SMCLK = default DCO
 *
 *                 MSP430F5529
```

```
 *           +-----------------+
 *           |                 |
 *           |                 |
 *           |                 |
 *           |                 |
 *           |            P1.0|--> LED1
 *           |            P1.1|<-- S2
 *
 *   Author:    Aleksandar Milenkovic, milenkovic@computer.org
 *   Date:      September 2010
 *   Modified:  Prawar Poudel, August 08, 2019
*******************************************************************************/
#include  <msp430.h>
#define   S2 BIT1&P1IN             // S2 is P1IN&BIT1

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

    P1DIR |= BIT0;                   // Set LED1 as output
    P1OUT = 0x00;                    // Clear LED1

    P1DIR &= ~BIT1;                  // Set the direction at S2 as input
    P1REN |= BIT1;                   // Enable Pull-up resistor
    P1OUT |= BIT1;                   // Required for proper IO

    _EINT();                         // Enable interrupts

    P1IE |= BIT1;                    // P1.1 interrupt enabled
    P1IES |= BIT1;                   // P1.1 hi/low edge
    P1IFG &= ~BIT1;                  // P1.1 IFG cleared

    for(;;) {
        while((S2) == 0);            // Wait until S2 is released
        P1OUT &= ~BIT0;              // LED1 is turned off
    }
}

// Port 1 interrupt service routine
#pragma vector = PORT1_VECTOR
__interrupt void Port1_ISR (void) {
    P1OUT |= BIT0;                 // LED1 is turned ON
    P1IFG &= ~BIT1;                // P1.0 IFG cleared
}
```

Figure 4. Press/Release Using Port 1 ISR (Lab6_D4.c)

Looking at the program in Figure 4 we can see that release is detected in the main program. A better implementation would delegate both press and release activities into the P1 ISR as shown in Figure 5. To implement this, we need to establish a global variable called S2pressed that keeps the current state of the switch (0 – released, 1 – pressed). At the beginning we expect a press event, so Port 1 is configured to wait for a falling edge on P1IN.BIT1 (SW2 is pressed). In that case, the ISR turns on LED1, sets the S2pressed and configures P1IES to trigger an interrupt when a rising edge is detected on P1IN.BIT1. When the switch is pressed and we the ISR is entered, the steps are taken to turn LED1 off and configure P1IES so that a new press event can be detected. This way, all work is done inside the P1 ISR and main program can put the processor into sleep state.

```
/******************************************************************************
 *   File:         Lab6_D5.c
 *   Description: The program detects when S2 is pressed and turns on LED1.
 *                 LED1 is kept on as long as S2 is pressed.
 *                 P1_ISR is used to detect both S2 presses and releases.
 *   Board:        EXP430F5529LP Launchpad
 *   Clocks:       ACLK = 32.768kHz, MCLK = SMCLK = default DCO
 *
 *                  MSP430F5529
 *             +-----------------+
 *             |                 |
 *             |                 |
 *             |                 |
 *             |                 |
 *             |             P1.0|--> LED1
 *             |             P1.1|<-- S2
 *
 *   Author: Aleksandar Milenkovic, milenkovic@computer.org
 *   Date:   September 2010
******************************************************************************/
#include  <msp430.h>

unsigned char S2pressed = 0;          // S2 status (0 not pressed, 1 pressed)

void main(void) {
    WDTCTL = WDTPW+WDTHOLD;           // Stop WDT
    P1DIR |= BIT0;                    // Set LED1 as output
    P1OUT = 0x00;                     // Clear LED1 status

    S2pressed = 0;

    P1DIR &= ~BIT1;                   // Set the direction at S2 as input
    P1REN |= BIT1;                    // Enable Pull-up resistor
    P1OUT |= BIT1;                    // Required for proper IO

    _EINT();                          // Enable interrupts
    P1IE |= BIT1;                     // P1IE.BIT1 interrupt enabled
    P1IES |= BIT1;                    // P1IES.BIT1 hi/low edge
    P1IFG &= ~BIT1;                   // P1IFG.BIT1 is cleared

    _BIS_SR(LPM0_bits + GIE);         // Enter LPM0(CPU is off); Enable interrupts
}

// Port 2 interrupt service routine
#pragma vector = PORT1_VECTOR
__interrupt void Port1_ISR (void) {
    if (S2pressed == 0) {
        S2pressed = 1;
        P1OUT |= BIT0;                  // LED1 is turned ON
```

```
        P1IFG &= ~BIT1;              // P1IFG.BIT0 is cleared
        P1IES &= ~BIT1;              // P1IES.BIT0 low/high edge
    } else if (S2pressed == 1) {
        S2pressed = 0;
        P1OUT &= ~BIT0;               // LED1 is turned ON
        P1IFG &= ~BIT1;               // P1IFG.BIT0 is cleared
        P1IES |= BIT1;                // P1IES.BIT0 hi/low edge
    }
}
```

Figure 5. Press/release Using Port 1 ISR – An Improved Implementation (Lab6_D5.c)

3 Clock Module

In the previous examples we have learned how to write a program that toggles the LEDs connected to the MSP430's output ports. We have also learned how write code to generate software delays. In our example, we assumed that the processor clock is around 1 µs (i.e., the clock frequency is approximately 1 MHz). The MSP430 family supports several clock modules and a user has a full control over these modules. By changing the content of relevant clock module control registers, one can change the processor clock frequency, as well as the frequency of other clock signals that are used for peripheral devices. In the next section, we will discuss the organization of the Unified Clock System (UCS) used in the MSP430F5529 device.

3.1 Unified Clock System (UCS)

MSP430x5xx family of microcontrollers have Unified Clock System (UCS) that provides various clocks for the MSP430 modules. The UCS module includes up to five clock sources and provide three clock signals.

The five clock sources included in the UCS module are as follows:

- **XT1CLK**: Low-frequency or high-frequency oscillator that can be used either with low-frequency 32768-Hz watch crystals, standard crystals, resonators, or external clock sources in the 4 MHz to 32 MHz range. XT1CLK can be used as a clock reference into the FLL. Some devices only support the low frequency oscillator for XT1CLK.
- **VLOCLK**: Internal very low power, low-frequency oscillator with 10-kHz typical frequency.
- **REFOCLK**: Internal trimmed low-frequency oscillator with 32768-Hz typical frequency, can be used as a clock reference into the FLL.
- **DCOCLK**: Internal digitally controlled oscillator (DCO) that can be stabilized by the FLL.
- **XT2CLK**: Optional high-frequency oscillator that can be used with standard crystals, resonators, or external clock sources in the 4 MHz to 32 MHz range. XT2CLK can be used as a clock reference into the FLL.

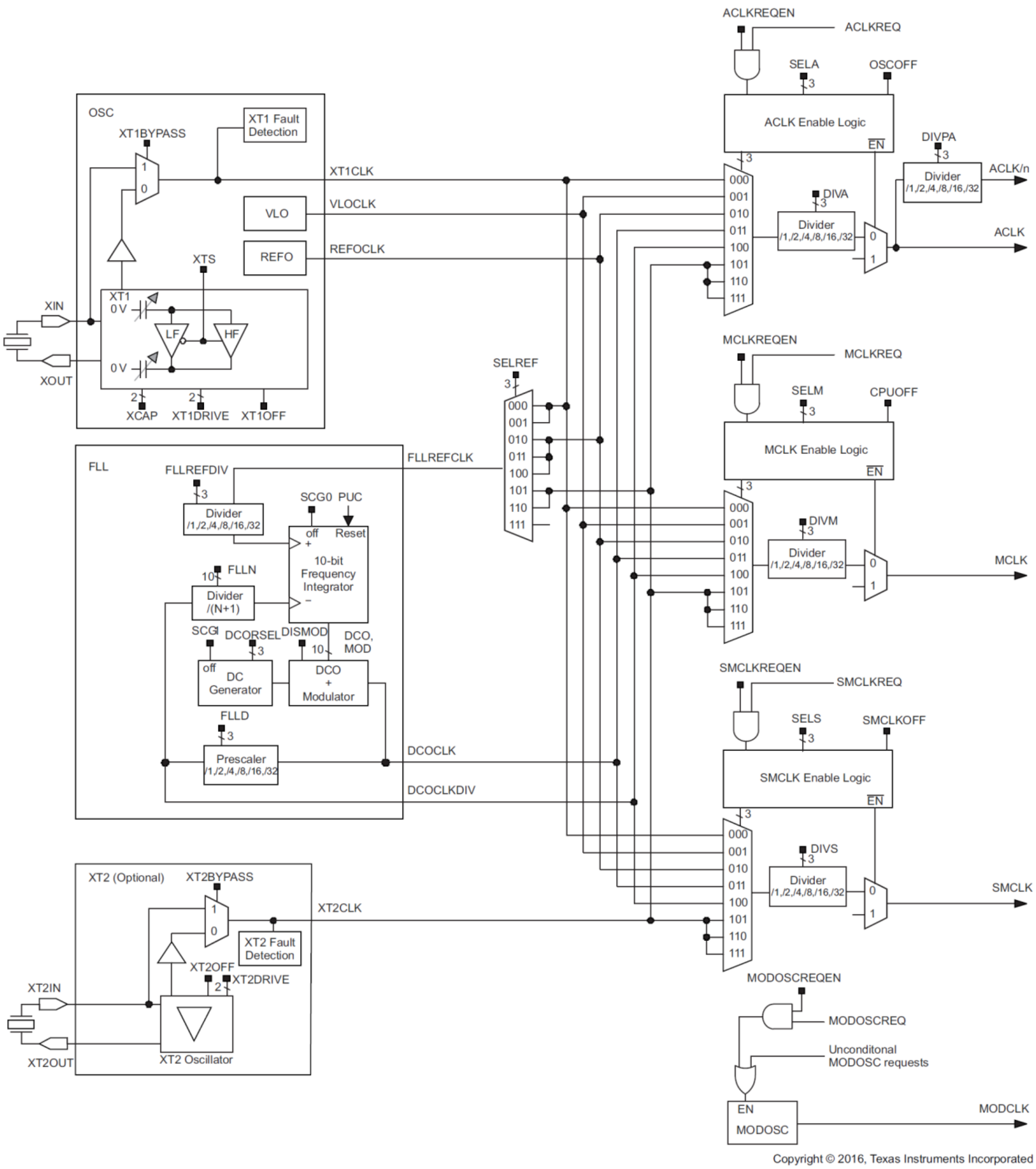


Figure 6. Block diagram of UCS module in MSP430x5xx devices

Following are the clock signals provided by the USC module:

- **ACLK**: Auxiliary clock. The ACLK is software selectable as XT1CLK, REFOCLK, VLOCLK, DCOCLK, DCOCLKDIV, and when available, XT2CLK. DCOCLKDIV is the DCOCLK frequency divided by 1, 2, 4, 8, 16, or 32 within the FLL block. ACLK can be divided by 1, 2, 4, 8, 16, or 32. ACLK/n is ACLK divided by 1, 2, 4, 8, 16, or 32 and is available externally at a pin. ACLK is software selectable by individual peripheral modules.

- **MCLK**: Master clock. MCLK is software selectable as XT1CLK, REFOCLK, VLOCLK, DCOCLK, DCOCLKDIV, and when available, XT2CLK. DCOCLKDIV is the DCOCLK frequency divided by 1, 2, 4, 8, 16, or 32 within the FLL block. MCLK can be divided by 1, 2, 4, 8, 16, or 32. MCLK is used by the CPU and system.
- **SMCLK**: Subsystem master clock. SMCLK is software selectable as XT1CLK, REFOCLK, VLOCLK, DCOCLK, DCOCLKDIV, and when available, XT2CLK. DCOCLKDIV is the DCOCLK frequency divided by 1, 2, 4, 8, 16, or 32 within the FLL block. SMCLK can be divided by 1, 2, 4, 8, 16, or 32. SMCLK is software selectable by individual peripheral modules.

On a PUC, the configuration of UCS is as follows:

- XT1 in LF mode is selected as source for XT1CLK which is selected for ACLK.
- DCOCLKDIV is selected for MCLK and SMCLK.
- FLL operation is enabled and XT1CLK is selected as reference clock for FLL.
- If the 32768 Hz crystal is used for XT1CLK, fault control logic sources ACLK with 32768 Hz REFOCLK because XT1 takes some time to stabilize.
- When the crystal start-up is obtained, FLL stabilizes MCLK and SMCLK at 1048576 Hz (2^20 Hz). f_{DCO} = 2097152 Hz

The operating modes of UCS is controlled using the following registers: **SCG0**, **SCG1**, **OSCOFF** and **CPUOFF**.

The UCS module can be configured using registers from **UCSCTL0** through **UCSCTL8**.

Digital Controlled Oscillator (DCO):

- The frequency of DCO can be adjusted by software using DCORSEL (in UCSCTL1 register), DCO and MOD bits (in UCSCTL0).
- DCO can also be stabilized using FLL to a multiple of FLL reference clock (i.e. multiple frequency of FLLREFCLK/n). FLL can accept different reference sources selectable by SELREF bits (UCSCTL3). The reference sources can be XT1CLK, REFOCLK or XT2CLK if available.
- The value of *n* can be defined in FLLREFDIV bits in UCSCTL3 register (n = 1,2,4,8,12 or 16). Default value is *n*=1.
- FLLD bits (in UCSCTL2 register) can be configured for FLL pre-scalar divider value D of 1,2,4,8,16 or 32. The default of D = 2 and MCLK and SMCLK are sourced from DCOCLKDIV thus providing clock frequency of DCOCLK/2.
- The divider (N+1) and divider D define DCOCLK and DCOCLKDIV frequencies. Value N+1 can be set from FLLN bits (in UCSCTL2 register) where N>0. Setting FLLN to 0 will cause FLLN to be 1 to prevent unintentional write.
- The final frequency of DCOCLK and DCOCLKDIV are as follows

$$f_{DCOCLK} = D \times (N + 1) \times (f_{FLLREFCLK} \div n)$$
$$f_{DCOCLKDIV} = (N + 1) \times (f_{FLLREFCLK} \div n)$$

Figure 7 Formula to compute DCOCLK and DCOCLKDIV frequencies

Frequency Locked Loop (FLL):

- The FLL continuously counts up or down a frequency integrator.
- The count is adjusted +1 with the frequency $f_{FLLREFCLK}/n$ (n=1,2,4,8,12 Or 16) or -1 with the frequency $f_{DCOCLK}/[D*(N+1)]$
- Five of the integrator bits (UCSCTL0bits 12 to 8) set the DCO frequency tap. Thirty-two taps are implemented for the DCO, and each is approximately8% higher than the previous. The modulator mixes two adjacent DCO frequencies to produce fractional taps. For a given DCO bias range setting, time must be allowed for DCO to settle on the proper tap operation. $(n*32)f_{FLLREFCLK}$ cycles are required between taps requiring worst case of $(n*32*32)f_{FLLREFCLK}$ cycles for DCO to settle.

3.2 Changing Processor Clocks: Examples

The following examples illustrate (Figure 8 and Figure 9) how you can change the processor clock frequency by modifying individual bits in the control registers. Please note that these examples only change the clocks and make them visible on external ports (some digital I/O ports have a special function to pass the clocks to the output, so we can observe them from the outside by connecting to oscilloscope). For learning how internal digitally controlled oscillator works read the corresponding user manual.

```
/******************************************************************************
 *  File:         Lab6_D6.c
 *  Description: MSP430F5529 Demo - FLL, Runs Internal DCO at 2.45MHz
 *               This program demonstrates setting the internal DCO to run at
 *               2.45MHz.
 *  Clocks:      ACLK = 32768Hz,
 *               MCLK = SMCLK = DCO = (74+1) x ACLK = 2457600Hz
 *
 *                MSP430F5529
 *            -----------------
 *        /|\|              XIN|-
 *         | |                 | 32kHz
 *         --|RST          XOUT|-
 *           |                 |
 *           |             P7.7|--> MCLK = 2.45MHz
 *           |                 |
 *           |             P2.2|--> SMCLK = 2.45MHz
 *           |             P1.0|--> ACLK = 32kHz
 *           |                 |
 *
 *  Author:     Aleksandar Milenkovic, milenkovic@computer.og
 *  Date:       September 2010
 *  Modified:   Prawar Poudel, August 2020
 ******************************************************************************/
#include  <msp430.h>

void main(void)
{
  WDTCTL = WDTPW + WDTHOLD;      // Stop watchdog timer
  P1DIR |= BIT0;                            // ACLK set out to pins
```

```
    P1SEL |= BIT0;
    P2DIR |= BIT2;                          // SMCLK set out to pins
    P2SEL |= BIT2;
    P7DIR |= BIT7;                          // MCLK set out to pins
    P7SEL |= BIT7;

    UCSCTL3 = SELREF_2;                     // Set DCO FLL reference = REFO
    UCSCTL4 |= SELA_2;                      // Set ACLK = REFO
    UCSCTL0 = 0x0000;                       // Set lowest possible DCOx, MODx

    // Loop until XT1,XT2 & DCO stabilizes - In this case only DCO has to stabilize
    do
    {
      UCSCTL7 &= ~(XT2OFFG + XT1LFOFFG + DCOFFG);
                                            // Clear XT2,XT1,DCO fault flags
      SFRIFG1 &= ~OFIFG;                    // Clear fault flags
    } while (SFRIFG1&OFIFG);                // Test oscillator fault flag

    __bis_SR_register(SCG0);                // Disable the FLL control loop
    UCSCTL1 = DCORSEL_4;                    // Select DCO range for operation
    UCSCTL2 |= 74;                          // Set DCO Multiplier for 2.45MHz
                                            // (N + 1) * FLLRef = Fdco
                                            // (74 + 1) * 32768 = 2.45MHz
    __bic_SR_register(SCG0);                // Enable the FLL control loop

    // Worst-case settling time for the DCO when the DCO range bits have been
    // changed is n x 32 x 32 x f_MCLK / f_FLL_reference. See UCS chapter in 5xx
    // UG for optimization.
    // 32 x 32 x 2.45 MHz / 32,768 Hz = 76600 = MCLK cycles for DCO to settle
    __delay_cycles(76000);

    while(1);
  }

```

Figure 8. Changing DCO to Run at 2.45 MHz using UCS Module (Lab6_D6.c)

```
/******************************************************************************
 *  File:         Lab6_D7.c
 *  Description:  MSP430F5529 Demo - FLL, Runs Internal DCO at 8MHz
 *                This program demonstrates setting the internal DCO to run at
 *                8MHz.
 *  Clocks:       ACLK = 32768Hz,
 *                MCLK = SMCLK = DCO = (121+1) x 2 x ACLK = 7995392Hz
 *
 *                 MSP430F5529
 *             -----------------
 *         /|\|              XIN|-
 *          | |                 | 32kHz
 *          --|RST          XOUT|-
 *            |                 |
 *            |             P7.7|--> MCLK = 8MHz
 *            |                 |
 *            |             P2.2|--> SMCLK = 8MHz
```

```
 *           |              P1.0|--> ACLK = 32kHz
 *           |                  |
 *
 *  Author:     Aleksandar Milenkovic, milenkovic@computer.og
 *  Date:       September 2010
 *  Modified:   Prawar Poudel
 *  Date:       August 2020
 ******************************************************************************/

#include  <msp430.h>

void main(void)
{
   WDTCTL = WDTPW + WDTHOLD;                // Stop watchdog timer

   P1DIR |= BIT0;                           // ACLK set out to pins
   P1SEL |= BIT0;
   P2DIR |= BIT2;                           // SMCLK set out to pins
   P2SEL |= BIT2;
   P7DIR |= BIT7;                           // MCLK set out to pins
   P7SEL |= BIT7;

   UCSCTL3 = SELREF_2;                      // Set DCO FLL reference = REFO
   UCSCTL4 |= SELA_2;                       // Set ACLK = REFO
   UCSCTL0 = 0x0000;                        // Set lowest possible DCOx, MODx

   // Loop until XT1,XT2 & DCO stabilizes - In this case only DCO has to stabilize
   do
   {
     UCSCTL7 &= ~(XT2OFFG + XT1LFOFFG + DCOFFG);
                                                 // Clear XT2,XT1,DCO fault flags
     SFRIFG1 &= ~OFIFG;                          // Clear fault flags
   } while (SFRIFG1&OFIFG);                      // Test oscillator fault flag

   __bis_SR_register(SCG0);                      // Disable the FLL control loop
   UCSCTL1 = DCORSEL_5;                          // Select DCO range 8MHz operation
   UCSCTL2 |= 249;                               // Set DCO Multiplier for 8MHz
                                                 // (N + 1) * FLLRef = Fdco
                                                 // (249 + 1) * 32768 = 8MHz
   __bic_SR_register(SCG0);                      // Enable the FLL control loop

   // Worst-case settling time for the DCO when the DCO range bits have been
   // changed is n x 32 x 32 x f_MCLK / f_FLL_reference. See UCS chapter in 5xx
   // UG for optimization.
   // 32 x 32 x 8 MHz / 32,768 Hz = 250000 = MCLK cycles for DCO to settle
   __delay_cycles(250000);
   while(1);                           // Loop in place
}

```

Figure 9. Changing DCO to Run at 8 MHz using UCS Module (Lab6_D7.c)

```c
/*------------------------------------------------------------------------------
 *  File:        Lab6_D8.c
 *  Description: MSP430F5529 Demo - FLL, Runs Internal DCO at 1MHz
 *               This program demonstrates setting the internal DCO to run at
 *               8MHz when SW1 is pressed.
 *
 *               A LED will keep blinking at 1Hz (0.5s ON and 0.5s OFF) at
 *               clock running at default frequency of 1Mhz. When SW1 is pressed,
 *               the frequency is changed to ~ 8 Mhz. When SW1 is pressed again,
 *               the frequency is restored to 1Mhz.
 *
 *  Clocks:      ACLK = 32768Hz,
 *               MCLK = SMCLK = DCO = (249+1) x ACLK = 8192000Hz
 *
 * Input:        SW1 (P2.1)
 * Output:       LED2 (P4.7)
 *
 *                MSP430F5529
 *             -----------------
 *         /|\|              XIN|-
 *          | |                 | 32kHz
 *          --|RST          XOUT|-
 *            |                 |
 *      SW1-->|P2.1         P7.7|--> MCLK = 1 or 8MHz
 *     LED2<--|P4.7             |
 *            |             P2.2|--> SMCLK = 1 or 8MHz
 *            |             P1.0|--> ACLK = 32kHz
 *            |                 |
 *
 *  Modified:   Prawar Poudel
 *  Date:       August 2020
 *------------------------------------------------------------------------------*/

// mandatory include statement
#include  <msp430.h>

// this function configures the clock sources as follows
// .. use internal REFOCLK for FLL reference clock (UCSCTL3 = SELREF_2)
// .. ACLK is sourced with REFOCLK (UCSCTL4 |= SELA_2)
// .. sets DCO tap to 0 (UCSCTL0 = 0)
// .. sets the modulation bit counter value to 0 (UCSCTL0 = 0)
void configure_clock_sources();
// this function changes the frequency of clock to 8 MHZ
inline void change_clock_freq_8Mhz();
// this function changes the frequency of clock to 8 MHZ
inline void change_clock_freq_1Mhz();


char is8Mhz = 0;

void main(void)
{
   WDTCTL = WDTPW + WDTHOLD;              // Stop watchdog timer

   P1DIR |= BIT0;                         // ACLK set out to pins
```

```
    P1SEL |= BIT0;

    P2DIR |= BIT2;                      // SMCLK set out to pins
    P2SEL |= BIT2;

    P7DIR |= BIT7;                      // MCLK set out to pins
    P7SEL |= BIT7;

    _EINT();                            // enable interrupts
    P2DIR &= ~BIT1;                     // set P2.1 as input (SW1)
    P2REN |= BIT1;                      // enable pull-up resistor
    P2OUT |= BIT1;
    P2IE |= BIT1;                       // enable interrupt at P2.1
    P2IES |= BIT1;                      // enable hi->lo edge for interrupt
    P2IFG &= ~BIT1;                     // clear any errornous interrupt flag

    P4DIR |= BIT7;                      // set P4.7 as output (LED2)

    configure_clock_sources();          // configure the clock sources

    while(1)                            // Loop in place (infinite)
    {
        P4OUT ^= BIT7;                  // toggle LED2
        __delay_cycles(500000);         // arbitrary delay of 500ms
    }
}

// this ISR handles the SW1 key press
#pragma vector = PORT2_VECTOR
__interrupt void PORT2_ISR(void)
{
    // let us clear the flag
    P2IFG &= ~BIT1;

    //debouncing section
    __delay_cycles(25000);

    // if SW1 is not pressed, return
    if((P2IN&BIT1)!=0x00)
        return;


    if(is8Mhz==0)
    {
        // if not at 8Mhz, let us change to 8Mhz
        change_clock_freq_8Mhz();
        is8Mhz = 1;
    }else
    {
        // if already in 8Mhz, let us take back to 1Mhz
        change_clock_freq_1Mhz();
        is8Mhz = 0;
    }
}
```

```c
// this function changes the frequency of clock to 8 MHZ
void change_clock_freq_8Mhz()
{
    __bis_SR_register(SCG0);                    // Disable the FLL control loop
    UCSCTL1 = DCORSEL_5;                        // Select DCO range 8MHz operation
    UCSCTL2 = 249;                              // Set DCO Multiplier for 8MHz
                                                // (N + 1) * FLLRef = Fdco
                                                // (249 + 1) * 32768 = 8MHz
    __bic_SR_register(SCG0);                    // Enable the FLL control loop

    // Worst-case settling time for the DCO when the DCO range bits have been
    // changed is n x 32 x 32 x f_MCLK / f_FLL_reference. See UCS chapter in 5xx
    // UG for optimization.
    // 32 x 32 x 8 MHz / 32,768 Hz = 250000 = MCLK cycles for DCO to settle
    __delay_cycles(250000);
}

// this function changes the frequency of clock to 1 MHZ
void change_clock_freq_1Mhz()
{
    __bis_SR_register(SCG0);                    // Disable the FLL control loop
    UCSCTL1 = DCORSEL_3;                        // Select DCO range 1MHz operation
    UCSCTL2 = 32;                               // Set DCO Multiplier for 1MHz
                                                // (N + 1) * FLLRef = Fdco
                                                // (32 + 1) * 32768 = 1MHz
    __bic_SR_register(SCG0);                    // Enable the FLL control loop

    // Worst-case settling time for the DCO when the DCO range bits have been
    // changed is n x 32 x 32 x f_MCLK / f_FLL_reference. See UCS chapter in 5xx
    // UG for optimization.
    // 32 x 32 x 1 MHz / 32,768 Hz = 33792 = MCLK cycles for DCO to settle
    __delay_cycles(33792);
}


// this function configures the clock sources as follows
// .. use internal REFOCLK for FLL reference clock (UCSCTL3 = SELREF_2)
// .. ACLK is sourced with REFOCLK (UCSCTL4 |= SELA_2)
// .. sets DCO tap to 0 (UCSCTL0 = 0)
// .. sets the modulation bit counter value to 0 (UCSCTL0 = 0)
void configure_clock_sources()
{
    UCSCTL3 = SELREF_2;                     // Set DCO FLL reference = REFO
    UCSCTL4 |= SELA_2;                      // Set ACLK = REFO
    UCSCTL0 = 0x0000;                       // Set lowest possible DCOx, MODx

    // Loop until XT1,XT2 & DCO stabilizes - In this case only DCO has to stabilize
    do
    {
    UCSCTL7 &= ~(XT2OFFG + XT1LFOFFG + DCOFFG);   // Clear XT2,XT1,DCO fault flags
    SFRIFG1 &= ~OFIFG;                            // Clear fault flags
    } while (SFRIFG1&OFIFG);                      // Test oscillator fault flag
}
```

Figure 10 Changing Clock Frequency to Observe Change in LED Blinking

4 References

It is crucial that you become familiar with the basics of how digital ports work – how to set their output direction, read from or write to the ports, set interrupts, and set up their special functions. We will be using these features to control hardware and communication between devices throughout this class. Please reference the following material to gain more insight on the device:

- The MSP-EXP430F5529LP board's user guide
- Chapter 5 in the MSP430F5529 user's guide (pages 158-185)
- Chapter 7 in the John H. Davies' *MSP430 Microcontroller Basics*

CPE 325: Embedded Systems Laboratory
Laboratory #7 Tutorial
MSP430 Timers, Watchdog Timer, Timers A and B

Aleksandar Milenković
Email: milenka@uah.edu
Web: http://www.ece.uah.edu/~milenka

Objective

This tutorial will introduce the watchdog timer (WDT) and the MSP430's TimerA module. You will learn the following topics:

Watchdog timer and its interval mode
Configuration of the peripheral device Timer_A
How to utilize Timer_A operating modes to solve real-world problems

Notes

All previous tutorials are required for successful completion of this lab. Especially, the tutorials introducing the TI Experimenter's Board and the Code Composer Studio software development environment.

Contents

1 Watchdog Timer: Toggling a LED Using Interval Mode ISR

Embedded computer systems usually have at least one timer peripheral device. You can think of timers as simple digital counters that in active mode increment or decrement their value at a specified clock frequency. Before using a timer in our application, we need to initialize it by setting its control registers. During initialization we need to specify timer's operating mode (whether they increment or decrement their value on each clock, pause, etc.), the clock frequency, and whether it will raise an interrupt request once the counter reaches zero (or a predetermined value set by software). Timers may have comparison logic to compare the timer value against a specific value set by software. When the values match, the timer may take certain actions, e.g., rollover back to zero, toggle its output signal, to name just a few possibilities. This might be used, for example to generate pulse width modulated waveforms used to control the speed of motors. Similarly, timers can be configured to capture the current value of the counter when a certain event occurs (e.g., the input signal changes from logic zero to logic one). Timers can also be used to trigger execution of the corresponding interrupt service routines. The MSP430 family supports several types of timer peripheral devices, namely the Watchdog Timer, Basic Timer 1, Real Time Clock, Timer A, and Timer B. Here we will learn more about the watchdog timer and how it can be used to periodically blink a LED.

The primary function of the watchdog-timer module (WDT) is to perform a controlled-system restart after a software problem occurs. If the selected time interval expires, a system reset is generated. If the watchdog function is not needed in an application, the module can work as an interval timer, to generate an interrupt after the selected time interval. Figure 1 illustrates a block diagram of the watchdog timer peripheral. It features a 16-bit control register, WDTCTL, and a 32-bit counter, WDTCNT. The watchdog timer counter (WDTCNT) is a 32-bit up-counter that is not directly accessible by software. The WDTCNT is controlled and time intervals selected through the watchdog timer control register WDTCTL. The WDTCNT can be sourced from any of ACLK, SMCLK, VLOCLK or X_CLK. The clock source is selected with the WDTSSEL bit.

Setting the WDTTMSEL bit to 1 selects the interval timer mode. This mode can be used to provide periodic interrupts. In the interval timer mode, the WDTIFG flag is set at the expiration of the selected time interval. A PUC is not generated in the interval timer mode at expiration of the selected timer interval and the WDTIFG enable bit WDTIE remains unchanged.

When the WDTIE bit and the GIE bit are set, the WDTIFG flag requests an interrupt. The WDTIFG interrupt flag is automatically reset when its interrupt request is serviced, or may be reset by software. The interrupt vector address associated with the interval timer mode is different from the one associated with the interrupt vector address used in the watchdog mode. Our goal is to configure the watchdog timer in the interval timer mode and use its interrupt service routine for blinking the LED. Let us assume that we want to have the LED on for 1 sec and off for 1 second (the period is 2 seconds of 0.5 Hz).

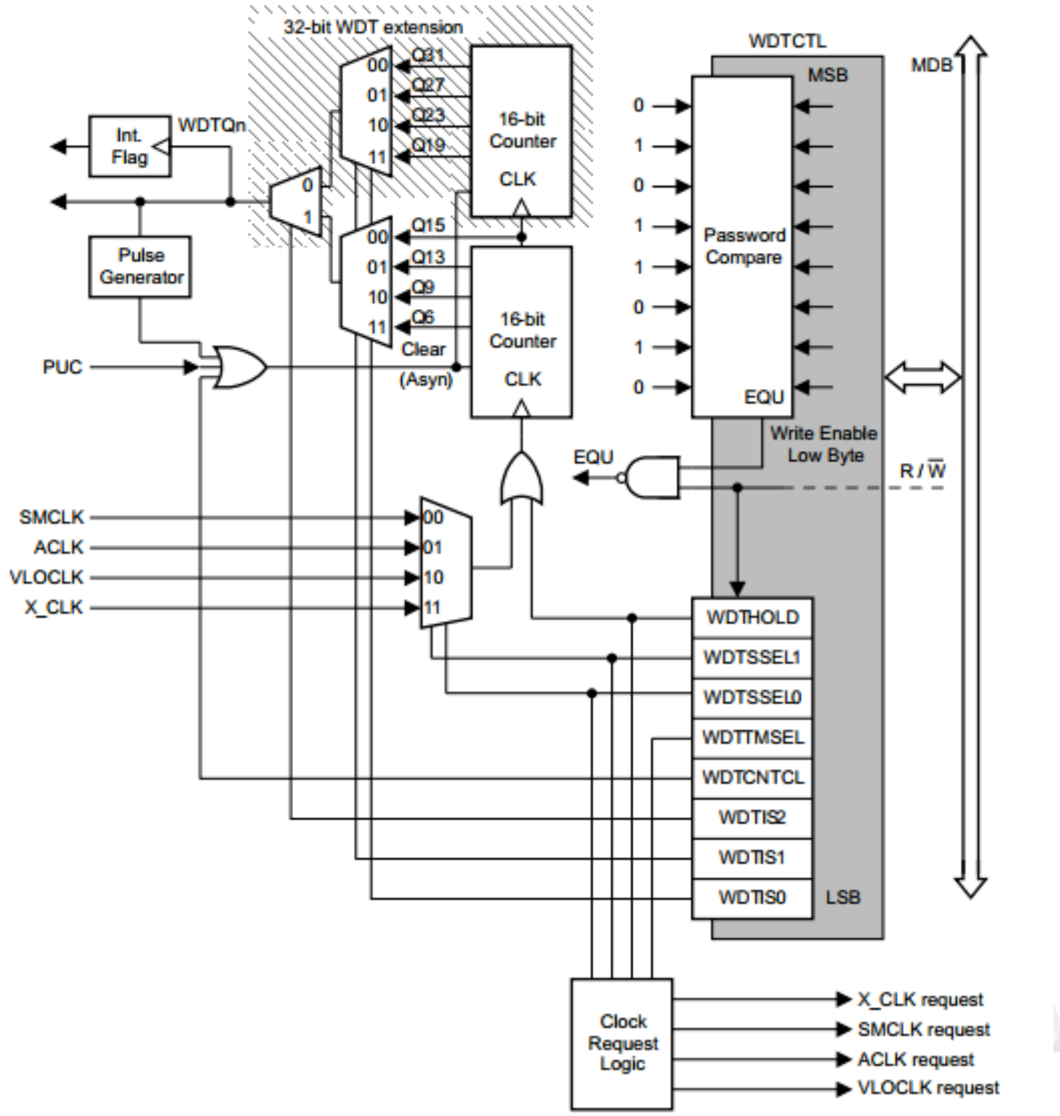


Figure 1. Block Diagram of the Watchdog Timer

Let us first consider how to specify time interval for the watchdog timer. If we select the ACLK clock for the clock source (SSEL = 0: WDTCNT is clocked by SMCLK; SSEL = 1: WDTCNT is clocked by ACLK), the timer clock is ~1 MHz.

Figure 2 shows a program that toggles a LED in the watchdog timer interrupt service routine every second. To generate an interrupt request every second, we configure the WDT as follows: select the ACLK as the clock source, ACLK=32,768 Hz (or 2^{15} Hz), and select the tap to be 2^{15}; the WDT interval time will be exactly 1 second. The WDT control word will look like this: WDTMSEL selects interval mode, WDTSSEL selects ACLK, and WTTCNTCL clears the WDTCNT. Analyze the header file for msp430F5529.h to locate pre-defined command words for the control register (e.g., WDT_ADLY_1000, WDT_ADLY_250, …).

```c
/*------------------------------------------------------------------------------
 * File:        Lab7_D1.c (CPE 325 Lab7 Demo code)
 * Function:    Blinking LED1 using WDT ISR (MPS430F5529)
 *
 * Description: This C program configures the WDT in interval timer mode,
 *              clocked with the ACLK clock. The WDT is configured to give an
 *              interrupt for every 1s. LED1 is toggled in the WDT ISR
 *              by xoring P1.0. The blinking frequency of LED1 is 0.5Hz.
 *
 * Board:       MSP-EXP430F5529 (includes 32-KHZ crystal on XT1 and
 *                               4-MHz ceramic resonator on XT2)
 *
 * Clocks:      ACLK = XIN-XOUT = 32768Hz, MCLK = SMCLK = DCO = default (~1MHz)
 *              An external watch crystal between XIN & XOUT is required for ACLK
 *
 *                          MSP430F5529
 *                       -----------------
 *                   /|\|              XIN|-
 *                    | |                 | 32kHz crystal
 *                    --|RST          XOUT|-
 *                      |                 |
 *                      |             P1.0|-->LED1(RED)
 *                      |                 |
 * Input:       None
 * Output:      LED1 blinks at 0.5Hz frequency
 * Author:      Aleksandar Milenkovic, milenkovic@computer.org
 *              Prawar Poudel
 * Date:        December 2008
 *------------------------------------------------------------------------------*/
#include <msp430.h>

void main(void) {
    WDTCTL = WDT_ADLY_1000;          // 1 s interval timer
    P1DIR |= BIT0;                   // Set P1.0 to output direction
    SFRIE1 |= WDTIE;                 // Enable WDT interrupt
    _BIS_SR(LPM0_bits + GIE);        // Enter LPM0 w/ interrupt
}

// Watchdog Timer Interrupt Service Routine
#pragma vector=WDT_VECTOR
__interrupt void watchdog_timer(void) {
    P1OUT ^= BIT0;                   // Toggle P1.0 using exclusive-OR
}
```

Figure 2. Toggling a LED using WDT_ISR

Figure 3 shows the program that also toggles the LED1 every second in the WDT ISR. However, the watchdog timer uses the SMCLK as the clock source and the maximum tap of 32,768 (2^{15}). The WDT generates an interrupt request every 32 ms. To toggle the LED1 every second we need to use a static local variable that is incremented every time we enter the ISR. When we collect 32 periods of 32 ms we have approximately 1 second of elapsed time, and we can then toggle LED1. Analyze the header file msp430f5529.h to locate pre-defined command word

WDT_MDLY_32. What bits of the control register are set with WDT_MDLY_32? What is the purpose of using static variable in the ISR? What happens if we use normal variable?

```
/*------------------------------------------------------------------------------
 * File:        Lab7_D2.c (CPE 325 Lab7 Demo code)
 *
 * Function:    Toggling LED1 using WDT ISR (MPS430F5529)
 *
 * Description: This C program configures the WDT in interval timer mode,
 *              clocked with SMCLK. The WDT is configured to give an
 *              interrupt for every 32ms. The WDT ISR is counted for 32 times
 *              (32*32.5ms ~ 1sec) before toggling LED1 to get 1 s on/off.
 *              The blinking frequency of LED1 is 0.5Hz.
 *
 * Clocks:      ACLK = XT1 = 32768Hz, MCLK = SMCLK = DCO = default (~1MHz)
 *              An external watch crystal between XIN & XOUT is required for ACLK
 *
 *                          MSP430xF5529
 *                       -----------------
 *                   /|\|              XIN|-
 *                    | |                 | 32kHz
 *                    --|RST          XOUT|-
 *                      |                 |
 *                      |             P1.0|-->LED1(RED)
 *                      |                 |
 * Input:       None
 * Output:      LED1 blinks at 0.5Hz frequency
 * Author:      Aleksandar Milenkovic, milenkovic@computer.org
 *              Prawar Poudel
 * Date:        December 2008
 *------------------------------------------------------------------------------*/
#include <msp430.h>

void main(void)
{
    WDTCTL = WDT_MDLY_32;           // 32ms interval (default)
    P1DIR |= BIT0;                  // Set P1.0 to output direction
    SFRIE1 |= WDTIE;                // Enable WDT interrupt

    _BIS_SR(LPM0_bits + GIE);       // Enter LPM0 with interrupt
}

// Watchdog Timer interrupt service routine
#pragma vector=WDT_VECTOR
__interrupt void watchdog_timer(void) {
    static int i = 0;
    i++;
    if (i == 32) {                  // 31.25 * 32 ms = 1s
        P1OUT ^= BIT0;              // Toggle P1.0 using exclusive-OR
                                    // 1s on, 1s off; period = 2s, f = 1/2s = 0.5Hz
        i = 0;
    }
}
```

Figure 3. Toggling the LED1 using WDT_ISR

Figure 4 shows the program that also toggles LED1 every second. The WDT is still configured in the interval mode and sets the WDTIFG every 1s. The program however does not use the interrupt service routine (the interrupt from WDT remains disabled). Instead, the main program polls repeatedly the status of the WDTIFG. If it is set, LED1 is toggled and the WDTIFG is cleared. Otherwise, the program checks the WDTIFG status again. The program spends majority of time waiting for the flag to be set and this approach is known as software polling. It is inferior to using interrupt service routines, but sometimes can be used to interface various peripherals. What are the advantages of using interrupts over software polling?

```
/*------------------------------------------------------------------------------
 * File:        Lab7_D3.c (CPE 325 Lab7 Demo code)
 * Function:    Blinking LED1 using software polling.
 * Description: This C program configures the WDT in interval timer mode and
 *              it is clocked with ACLK. The WDT sets the interrupt flag (WDTIFG)
 *              every 1 s. LED1 is toggled by verifying whether this flag
 *              is set or not. After it is detected as set, the WDTIFG is cleared.
 * Clocks:      ACLK = LFXT1 = 32768Hz, MCLK = SMCLK = DCO = default (2^20 Hz)
 *              An external watch crystal between XIN & XOUT is required for ACLK
 *
 *                          MSP430F5529
 *                       -----------------
 *                   /|\|              XIN|-
 *                    | |                 | 32kHz
 *                    --|RST          XOUT|-
 *                      |                 |
 *                      |             P1.0|-->LED1(RED)
 *                      |                 |
 * Input:       None
 * Output:      LED1 blinks at 0.5Hz frequency
 *
 * Author:      Aleksandar Milenkovic, milenkovic@computer.org
 * Revised by:  Prawar Poudel
 *------------------------------------------------------------------------------*/
#include <msp430.h>

void main(void)
{
    WDTCTL = WDT_ADLY_1000;             // 1 s interval timer
    P1DIR |= BIT0;                      // Set P2.2 to output direction

    for (;;) {
        // Use software polling
        if ((SFRIFG1 & WDTIFG) == 1) {
            P1OUT ^= BIT0;
            SFRIFG1 &= ~WDTIFG;         // Clear bit WDTIFG in IFG1
        }
    }
}

```

Figure 4. Toggling LED1 using WDT and Software Polling on WDTIFG

2 Timers (A and B)

MSP430 family supports several types of timer peripheral devices, namely the Watchdog Timer, Real Time Clock, Timer A, Timer B and Timer D. In this part of the tutorial we will be studying Timer A and B.

Among the several ways to perform periodic tasks, one is to have a software delay. Using a software delay keeps the processor running while it is not needed, which leads to wasted energy. Further, counting clock cycles and instructions is quite cumbersome. MSP430s have peripheral devices called timers that can raise interrupt requests regularly. It is possible to use these timers to perform periodic tasks. Since a timer can use a different clock signal than the processor, it is possible either to turn off the processor or to work on other computations while the timer is counting. This saves energy and reduces code complexity. Note that a MSP430 can have multiple timers and each timer can be utilized independently from the other timers. Furthermore, each timer has several modes of counting. Though in this lab we will be using Timer A to blink an output signal at regular interval of time, keeping time is not the only use of timers.

Figure 5 shows a block diagram of Timer A peripheral. It consists of a timer block and upto seven configurable capture and compare blocks (TAxCCR0 – TAxCCR6). (In MSP430F5529, there are three instantiations of Timer A: TA0 with 5 capture and compare blocks, TA1 with 3 capture and compare registers, and TA2 with 3 capture and compare blocks. Thus, TAxCCR0 can mean any of TA0CCR0, TA1CCR0 or TA2CCR0. Timer B has 7 capture and compare registers.

The timer block can be configured to act as an 8-bit, 10-bit, 12-bit, or 16-bit counter and supports 4 counting modes: STOP (MCx=00), UP (MCx=01), Continuous (MCx=10), and UP/DOWN (MCx=11). The source clock can be selected among multiple options (TAxSEL bits), and the selected clock can be further divided by 1 (IDx=00), 2 (IDx=01), 4 (IDx=10), and 8 (IDx=11). In the UP and UP/DOWN modes counter counts up to the value that is specified by the TAxCCR0 register. The timer block is configured using Timer A control register, TAxCTL, which contains control bits TAxSEL, IDx, CNTLx, and others. Please examine the format of this register.

Each Capture and Compare block *n* (n from 0 to 6) contains a 16-bit latch register TAxCCRn and corresponding control logic enabling two type of operations CAPTURE and COMPARE. Each capture and compare block *n* is controlled by its own control register, TAxCCTLn.

Capture refers to an operation where the value of the running counter (TAxR) is captured in the TAxCCRn on a hardware event (e.g., external input changes its state from a logic 1 to a logic 0 or from a logic 0 to a logic 1) or on a software trigger (e.g., setting some control bits). This operation is very useful when we want to timestamp certain events. Imagine you are designing a system that needs to log an exact moment when a car enters a parking lot. A sensor can trigger a change of state of an input that further triggers the capture operation on Timer A. This way we can precisely timestamp the event down to a single clock cycle precision with no software-induced delay. The value of the running timer is captured by the TAxCCRn that can then be read and processed in software.

Compare refers to an operation where actions are triggered at specific moments in time. This operation is crucial for generating Pulse-Width-Modulated signals (PWMs) – periodic signals where duty cycles is fully controlled: the period the signal is on over the entire period. The PWM type of signals are often used in robotics to control motors. In compare mode of operation the corresponding latch register, TAxCCRn, is initialized to a certain value. When the value in the running counter (TBR) reaches the value in TAxCCRn, an output signal can change its state (set, reset, or toggle).

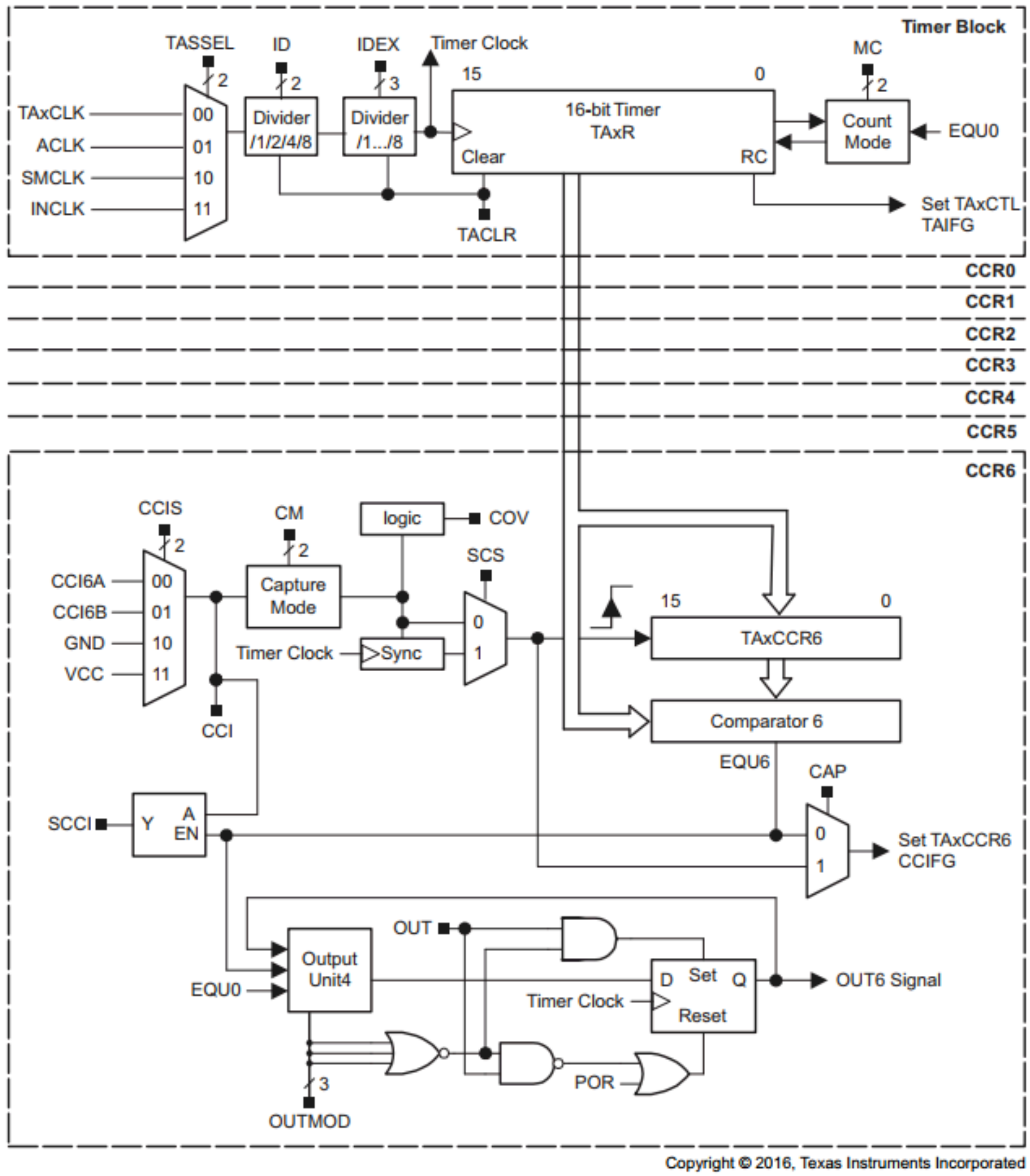


Figure 5. Timer_A Block Diagram

2.1 Toggle an output Using Timer A

Let us first consider an example where we utilize a Timer A2 device to toggle an output on the MSP-EXP430F5529LP board. For this example, we will be toggling the channel 1 of Timer A2, i.e. TA2.1. TA2.1 is multiplexed with P2.4.

P2.4 should be periodically turned on for 0.065 seconds and then turned off for 0.065 seconds (one period is ~0.13 seconds, or toggling rate is ~7.6 Hz). We have learned how to execute the toggling using a software delay or by using the watchdog timer. Now, we would like to utilize the MSP430's Timer A peripheral device.

Figure 7 shows a program that toggles P2.4 as specified. We can see that the port 2.4 is multiplexed with TA2.1 special function, which is the output from the capture and compare block 1 (TA2CCR1). Note: how do we know this? Find a document where this information is available? How to configure Port 2.4 to output TA2.1?

The capture and compare block 1 can be configured to set/reset or toggle the output signal TA2.1 when the value in the running counter reaches the value in the capture and control register TA2CCR1. Thus, when a value in the Timer A2 counter is equal to the value in TA2CCR1, we can configure the Timer A2 to toggle its output, TA2.1, automatically. The default value in TA2CCR1 is 0, thus, the output will be toggled every time the counter rolls over to 0x0000. However, before we can use TA2.1 as an output on P2.4, we need to configure the port 2 selection register, P2SEL, pin 4 to its special I/O function instead of its common digital I/O function (P2SEL |= BIT4;). This way we ensure that the P2.4 mirrors the behavior of the TA2.1 signal.

The next step is to configure clock sources. The MSP430 clocks MCLK, SMCLK, and ACLK have default frequencies as follows: MCLK = SMCLK ~ 1MHz and ACLK = 32 KHz. Timer A2 is configured to use the SMCLK as its clock input and to operate in the continuous mode. Timer A2's counter will count from 0x0000 to 0xFFFF. When the counter value reaches 0x0000, the EQU1 will be asserted indicating that the counter has the same value as the TA2CCR1 (here it is not set because by default it is cleared). We can select the output mode 4 (toggle) that will toggle the output every time EQU1 is asserted. This way we can determine the time period when the TA2.1 is reset and set. The TA2.1 will be set for 65,536*1/2^20 = 0.0625 seconds and will be reset for 65,536*1/2^20 = 0.0625 seconds. Please note that we do not need to use an interrupt service routine to toggle the signal in this case. The Timer A2 will toggle the P2.4 independently, and we can go into a low power mode and remain there for the rest of the application lifetime. *What are the different low power modes available in MSP430F5529? How do they differ from each other? Explain your answers.*

To visualize the output, you can use the Grove Boosterpack. P2.4 is connected to header J14 on the digital section of the board. Using the connecter cable to hook the Buzzer in the pack, you can notice the output in the note played in Buzer.

```
/*------------------------------------------------------------------------------
 * File:        Lab7_D4.c (CPE 325 Lab7 Demo code)
 *
 * Function:    Toggling signal using Timer_A2 in continuous mode (MPS430F5529)
 *
 * Description: In this C program, Timer_A2 is configured for continuous mode. In
 *              this mode, the timer TA2 counts from 0 up to 0xFFFF (default 2^16).
 *              So, the counter period is 65,536*1/2^20 = 62.5ms when SMCLK is
 *              selected. The TA2.1 output signal is configured to toggle every
 *              time the counter reaches the maximum value, which corresponds to
 *              62.5ms. TA2.1 is multiplexed with the P2.4, and there is a extension
 *              header from this pin.
```

```c
 *
 *                Thus the output frequency on P2.4 will be f = SMCLK/(2*65536) ~ 8 Hz.
 *                Please note that once configured, the Timer_A toggles the signal
 *                in pin P2.4 automatically even when the CPU is in sleep mode.
 *                Please use oscillator to see this.
 *
 *                Using the Grove Boosterpack, you can hook-up the Buzzer to the
 *                J14 header. This connects the Signal Pin of buzzer to P2.4.
 *                The buzzer produces sound when the signal value is high
 *                and vice versa.
 *
 * Clocks:        ACLK = LFXT1 = 32768Hz, MCLK = SMCLK = DCO = default (2^20 Hz)
 *                An external watch crystal between XIN & XOUT is required for ACLK
 *
 *                            MSP430F5529
 *                         -----------------
 *                     /|\|              XIN|-
 *                      | |                 | 32kHz
 *                      --|RST          XOUT|-
 *                        |                 |
 *                        |       P2.4/TA2.1|-->Buzzer
 *                        |                 |
 * Input:         None
 * Output:        Toggle output at P2.4 at 8Hz frequency using hardware TA2
 * Author:        Aleksandar Milenkovic, milenkovic@computer.org
 *                Prawar Poudel
 *------------------------------------------------------------------------------*/
#include <msp430F5529.h>

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

    P2DIR |= BIT4;            // P2.4 output (TA2.1)
    P2SEL |= BIT4;            // P2.4 special function (TA2.1 output)

    TA2CCTL1 = OUTMOD_4;      // TA2.1 output is in toggle mode
    TA2CTL = TASSEL_2 + MC_2; // SMCLK is clock source, Continuous mode

    _BIS_SR(LPM0_bits + GIE); // Enter Low Power Mode 0
}

```

Figure 6. C Program for Toggling Pin2.4 Using TimerA, Continuous Mode

Try to modify the code from Figure 6 by selecting a different source clock for Timer A. What happens if we use the following command: TA2CTL = TASSEL_1 + MC_2? What is the period of toggling the signal? Explain your answer. Try using divider for the clock source.

The given example may not be suitable if you want to control the period of toggling since the counter in the continuous mode always counts from 0x0000 to 0xFFFF. This problem can be solved by opting for the UP-counter mode. The counter will count from 0x000 up to the value specified in the TACCR2. This way we can control the time period. Let us consider an example where we want the buzzer to be 1 second on and 1 second off (toggling rate is 0.5 Hz).

Figure 7 shows the C code for this example. Note the changes. How do we specify UP mode? How do we select the ACLK clock as the TimerA source clock? What output mode do we use? Is it better to use ACLK instead of SMCLK in this example? Explain your answers.

```c
/*------------------------------------------------------------------------------
 * File:        Lab7_D5.c (CPE 325 Lab7 Demo code)
 *
 * Function:    Toggle signal using Timer_A2 in up mode (MPS430F5529)
 *
 * Description: In this C program, Timer_A2 is configured for UP mode. In this
 *              mode, the timer TA2 counts from 0 up to value stored in TA2CCR0.
 *              So, the counter period is CCR0*1us. The TA2.1 output signal is
 *              configured to toggle every time the counter reaches the value
 *              in TA2CCR1. TA2.1 is multiplexed with the P2.4. Thus, the output
 *              frequency on P2.4 will be f = ACLK/(2*CCR0) = 0.5Hz. Please note
 *              that once configured, the Timer_A2 toggles the signal automatically
 *              even when the CPU is in sleep mode.
 *
 *              Using the same connection as in Lab7_D4.c, you should be able to
 *              hear Buzzer ON for 1s and OFF for 1s continuously.
 *
 * Clocks:      ACLK = LFXT1 = 32768Hz, MCLK = SMCLK = DCO = default (2^20 Hz)
 *              An external watch crystal between XIN & XOUT is required for ACLK
 *
 *                          MSP430xF5529
 *                       -----------------
 *                   /|\|              XIN|-
 *                    | |                 | 32kHz
 *                    --|RST          XOUT|-
 *                      |                 |
 *                      |      P2.4/TA2.1|--> Buzzer
 *                      |                 |
 * Input:       None
 * Output:      Toggle output at P2.4 at 0.5Hz frequency using hardware TA2
 * Author:      Aleksandar Milenkovic, milenkovic@computer.org
 *              Prawar Poudel
 *------------------------------------------------------------------------------*/
#include <msp430F5529.h>

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

    P2DIR |= BIT4;              // P2.4 output
    P2SEL |= BIT4;              // P2.4 special function (TA2.1 output)

    TA2CCTL1 = OUTMOD_4;        // TA2.1 output is in toggle mode
    TA2CTL = TBSSEL_1 + MC_1;   // ACLK is clock source, UP mode
    TA2CCR0 = 32767;            // Value to count upto for Up mode

    _BIS_SR(LPM3_bits + GIE);   // Enter Low Power Mode 3
}
```

Figure 7. C Program for Toggling Pin2.4 Using TIMER_A (UP MODE)

2.2 Additional Timer_A Functionality

As already seen, Timer A is quite powerful due to the selectable clocks, automated outputs, and adjustable maximum count value. The Timer A peripheral has additional features which greatly expand its functionality and versatility. These features include:

- Multiple capture/compare modules
- Multiple output control modes
- Ability to call multiple interrupts at different count values
- Ability to select from multiple counting modes

Often times, it will be necessary to perform multiple tasks with a single timer peripheral. Fortunately, the Timer A system has multiple channels that can be set up to perform their tasks at designated count values. The MSP430F5529 has 3 instantiations of Timer A, namely TA0, TA1 and TA2. Each of these Timer A have different number of channels. TA0 has 5 channels, TA1 has 3 channels and TA2 has 3 channels. Each channel normally has a shared output pin similar to what we saw with the TA2.1 pin in the examples above. In Figure 8 below, note that some of the pins are shared with TA1 channels. This means that capture/compare channel can directly control output devices connected to these pins.

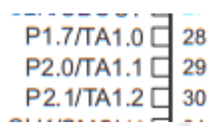


Figure 8. Port pins shared with TA1 channels

If you further examine the MSP-EXP430F5529LP experimenter board schematic, you can find where the other channel output pins are located. In order to use other channels in each instantiation of each of TA), TA1 and TA2, channel 0 must still be set up in each of them. All of the channels have their own configuration register and their own count value register. Each channel can be configured to use its output pin directly (as in above example Lab7_D5), but they can also be used to call interrupt service routines. An interrupt vector is dedicated to channel 0, but other channels each of the timers can be configured to call a separate ISR.

It is important to think about how the counting methods affect the interrupt calls from the different capture/compare channels. The user's guide contains definitive information about the Timer B including examples that demonstrate how the various functions work. In general, the counting modes work as follows:
Counting mode 0 – Stop mode – The timer is inactive
Counting mode 1 – Up mode – The timer counts up to the value for channel 0. An interrupt for each channel set up is called at the corresponding count value on the way up. At the maximum value an interrupt for channel 0 is called, and at the next timer count a general interrupt is generated. Remember that these interrupts may correspond to output pin control or interrupt service routine vectoring depending on the channel configuration.

Counting mode 2 – Continuous mode – The timer counts up to its maximum value (65535 for 16 bit mode). Along the way, corresponding interrupts are called at each channel's count value register. At 0 a general Timer Ax interrupt is set.
Counting mode 3 – Up/Down mode – The timer counts up to the value in the channel 0 register, and then it counts back down to 0 again. The channel interrupts are called when the value is reach on the up count and the down count. The general Timer A interrupt is called when 0 is reached.

In Figure 9 below, channels 0 and 1 are used to call two separate ISRs. Since the timer is in up/down mode, the channel 0 ISR is only called once per counting cycle (on the max value set by the CCR0 register) while the channel 1 ISR is called twice per counting cycle if its CCR1 value is less than CCR0. It is called on the up count and the down count. This mode is especially useful when creating PWM signals since the count register value determines the duty cycle of the output signal.

```
/*------------------------------------------------------------------------------
 * File:        Lab7_D6.c (CPE 325 Lab7 Demo code)
 *
 * Function:    Blinking LED1 & LED2 using Timer_A0 with interrupts (MPS430F5529)
 *
 * Description: In this C program, Timer_A0 is configured for up/down mode with
 *              ACLK source and interrupts for channel 0 and channel 1 are
 *              enabled. In up/down mode timer TA0 counts the value from 0 up to
 *              value stored in TA0CCR0 and then counts back to 0. The interrupt
 *              for TA0 is generated when the counter reaches value in TA0CCR0.
 *              The interrupt TA0.1 is generated whenever the counter reaches value
 *              in TA0CCR1. Thus, TA0.1 gets two interrupts while counting upwards
 *              and counting downwards. This simulates a PWM control - adjusting
 *              the TA0.1 and TA0.0 CCR register values adjusts the duty cycle of the
 *              PWM signal.
 *
 * Clocks:      ACLK = LFXT1 = 32768Hz, MCLK = SMCLK = DCO = default (2^20 Hz)
 *              An external watch crystal between XIN & XOUT is required for ACLK
 *
 *                          MSP430x5529x
 *                       -----------------
 *                   /|\|              XIN|-
 *                    | |                 | 32kHz
 *                    --|RST          XOUT|-
 *                      |                 |
 *                      |             P1.0|--> LED1(RED)
 *                      |             P4.7|--> LED2(GREEN)
 * Input:       None
 * Output:      LED1 blinks at 1.64Hz with 20-80% duty cycle and LED2 blinks at
 *              0.82Hz with 50-50% duty cycle.
 *
 * Author:      Aleksandar Milenkovic, milenkovic@computer.org
 *              Prawar Poudel
 *------------------------------------------------------------------------------*/
#include <msp430F5529.h>
```

```c

void main(void) {
    WDTCTL = WDTPW + WDTHOLD;      // Stop WDT
    _EINT();                       // Enable interrupts

    P1DIR |= BIT0;                 //LED1 as output
    P4DIR |= BIT7;                 //LED2 as output

    P1OUT &= ~BIT0;                // ensure LED1 and LED2 are off
    P4OUT &= ~BIT7;

    TA0CCTL0 = CCIE;               // TA0 count triggers interrupt
    TA0CCR0 = 10000;               // Set TA0 (and maximum) count value

    TA0CCTL1 = CCIE;               // TA0.1 count triggers interrupt
    TA0CCR1 = 2000;                // Set TA0.1 count value

    TA0CTL = TASSEL_1 | MC_3;      // ACLK is clock source, UP/DOWN mode

    _BIS_SR(LPM3);                 // Enter Low Power Mode 3
}

#pragma vector = TIMER0_A0_VECTOR
__interrupt void timerISR(void) {
    P4OUT ^= BIT7;                 // Toggle LED2
}

#pragma vector = TIMER0_A1_VECTOR
__interrupt void timerISR2(void) {
    P1OUT ^= BIT0;                 // Toggle LED1
    TA0CCTL1 &= ~CCIFG;             // Clear interrupt flag
}

```

Figure 9. Code Using Multiple Timer B CC Channels and ISRs to Toggle LEDs

3 References

Getting started with the timers can be confusing at first. It is important to understand their functions and different operating modes. The following texts can help you understand the timers operation better:

- Chapter 8 in the Davies Text (page 275 – 368)
    - Watchdog Timer – page 276 – 281
    - Timer A – page 287 – 300
    - Timer B – page 353 – 356
- The user's guide
    - Watchdog Timer – Chapter 16 (page 453 – 459)
    - Timer A – Chapter 17 (page 460 – 481)
- Timer B – Chapter 18 (page 482 – 406)

CPE 325: Embedded Systems Laboratory
Laboratory #8 Tutorial
UART Serial Communications

Aleksandar Milenković
Email: milenka@uah.edu
Web: http://www.ece.uah.edu/~milenka

Objective

This tutorial will introduce communication protocols used with MSP430 and other devices. Specifically, it will cover asynchronous serial communication using USCI peripheral. You will learn the following topics:

Configuration of the USCI peripheral device for UART mode
Utilization of the USCI in UART mode for serial communication with a workstation
Understanding of workstation clients interfacing serial communication ports (putty) and UAH serial communication application

Notes

All previous tutorials are required for successful completion of this lab. Read CPE323 lecture discussing UART Communication.

Contents

1 Serial Communication

An MSP430-based platform can communicate with another system, such as a personal computer, using either the synchronous or asynchronous communication mode. For two devices to communicate synchronously, they must share a common clock source. In this lab, we are going to interface a MSP430 with a personal computer using an asynchronous communication mode. Since the two devices do not share a clock signal, there should be an agreement between the devices on the speed of the communication before the actual interface starts.

To configure the MSP430 in UART mode, the internal divider and the modulation register should be initialized appropriately. The internal divider is calculated by dividing the clock by the baud rate. But, the division of the clock by the baud rate is usually not a whole number. Therefore, to take account of the fraction part of the division, we use the modulation register. The value in the modulation register is calculated in such a way that the time it takes to receive/transmit each bit is as close as possible to the exact time given by the baud rate. If the appropriate modulation value is not used, the fraction part of the division of clock frequency by the baud rate will accumulate and eventually make the two devices unable to communicate. An MSP430-based platform can be connected to a PC machine using the HyperTerminal application in Windows.

Let us consider a program that sends a character from the PC to the MSP430F5529 microcontroller and echoes the character back to the PC (Figure 1). Since we cannot connect the two systems (our PC and the microcontroller) to the same clock source, we should use the UART mode. The USCI peripheral can be utilized for that purpose. The communication speed is 115,200 bits/s (one-bit period is thus 1/115,200 or ~8.68 us). The USCI clock, UCLK, is connected to SMCLK running at 1,048,576 Hz. To achieve the baud rate of 115,200 bits per second, the internal divider registers are initialized to UCA0BR0=0x09, and UCABR1=0x00, because 1,048,576/115,200 = 9.1 ~ 9. Additionally, the modulation register, UCA0MCTL, is set to 0x02 (bit 0 of the field UCBRSx is set to 1). See the reference manual (Table 36.4 in https://www.ti.com/lit/ug/slau208q/slau208q.pdf) for more common combinations of clock source and baud rate and values for baud rate control registers. Also, the reference manual gives details on how the right value in UCA0MCTL is determined (the idea is to continuously minimize probability of erroneous detection at the receiver side).

Figure 1 shows an implementation using polling. The function UART_setup() is configuring USCI in UART mode: 8-bit characters, no parity, 1 stop bit, and the baud rate is set as described above. Please note that we follow recommended sequence of steps for USCI initialization – the SWRST bit in the control register remains set during initialization and it is cleared once the initialization is over. The main program loop is an infinite loop where we use polling to detect whether a new character is received. The program is waiting in line 71 for new character to be received. When a character is received in the UCA0RXBUF register, the UCA0RXIFG bit is set. Before the character is echoed back through the serial interface, we first check whether the USCI's transmit data buffer is empty (line 73). When the transmit buffer is empty, we proceed with copying the received character that is in UCA0RXBUF into UCA0TXBUF. The LED1 is toggled before we go back to the main loop.

```
/*------------------------------------------------------------------------------
 * File:          Lab8_D1.c
 *
 * Function:      Echo a received character, using polling.
 *
 * Description:   This program echos the character received from UART back to UART.
 *                Toggle LED1 with every received character.
 *                Baud rate: low-frequency (UCOS16=0);
 *                1048576/115200 = ~9.1 (0x0009|0x01)
 *
 * Clocks:        ACLK = LFXT1 = 32768Hz, MCLK = SMCLK = default DCO
 *
 * Board:         MSP-EXP430F5529
 *
 * Instructions: Set the following parameters in putty
 * Port: COMx
 * Baud rate: 115200
 * Data bits: 8
 * Parity: None
 * Stop bits: 1
 * Flow Control: None
 *
 * Note:       If you are using Adafruit USBtoTTL cable, look for COM port
 *             in the Windows Device Manager with the following text:
 *             Silicon Labs CP210x USB to UART Bridge (COM<x>).
 *             Connecting Adafruit USB to TTL:
 *              GND - black wire - connect to the GND pin (on the board or
BoosterPack)
 *              Vcc - red wire - leave disconnected
 *              Rx    white wire (receive into USB, connect on TxD of the board P3.3)
 *              Tx -  green wire (transmit from USB, connect to RxD of the board
P3.4)
 *        MSP430F5529
 *     -----------------
 * /|\ |             XIN|-
 *  |  |                | 32kHz
 *  |--|RST         XOUT|-
 *     |                |
 *     |    P3.3/UCA0TXD|------------>
 *     |                | 115200 - 8N1
 *     |    P3.4/UCA0RXD|<------------
 *     |            P1.0|----> LED1
 *
 * Input:     None (Type characters in putty/MobaXterm/hyperterminal)
 * Output:    Character echoed at UART
 * Author:    A. Milenkovic, milenkovic@computer.org
 * Date:      October 2018, modified August 2020
 *------------------------------------------------------------------------------*/
#include <msp430.h>

void UART_setup(void) {

```

```
    P3SEL |= BIT3 + BIT4;   // Set USCI_A0 RXD/TXD to receive/transmit data
    UCA0CTL1 |= UCSWRST;    // Set software reset during initialization
    UCA0CTL0 = 0;           // USCI_A0 control register
    UCA0CTL1 |= UCSSEL_2;   // Clock source SMCLK

    UCA0BR0 = 0x09;         // 1048576 Hz  / 115200 lower byte
    UCA0BR1 = 0x00;         // upper byte
    UCA0MCTL |= UCBRS0;     // Modulation (UCBRS0=0x01, UCOS16=0)

    UCA0CTL1 &= ~UCSWRST;   // Clear software reset to initialize USCI state machine
}

void main(void) {
    WDTCTL = WDTPW + WDTHOLD;       // Stop WDT
    P1DIR |= BIT0;                  // Set P1.0 to be output
    UART_setup();                   // Initialize UART

    while (1) {
       while(!(UCA0IFG&UCRXIFG));   // Wait for a new character
       // New character is here in UCA0RXBUF
       while(!(UCA0IFG&UCTXIFG));   // Wait until TXBUF is free
       UCA0TXBUF = UCA0RXBUF;       // TXBUF <= RXBUF (echo)
       P1OUT ^= BIT0;               // Toggle LED1
    }
}

```

Figure 1. Echoing a Character Using the USCI in UART Mode and Polling

Figure 2 shows the program that performs the same task, but this time an interrupt service routine tied to the USCI receiver is used. In the main program the USCI is configured to generate an interrupt request when a new character is received. Whenever a character is received and loaded into UCA0RXBUF, the interrupt flag UCA0RXIFG is set and interrupt request is raised. The main program does nothing beyond initialization – the processor is in a low-power mode 0 (LPM0). What clock signals are down in this mode?

All actions in this implementation occurs inside the service routine. The processor wakes up when a new character is received, and we find ourselves inside the service routine. In the ISR before writing the new character to UCA0TXBUF to transmit back to the workstation, we need to make sure that it is indeed empty to avoid loss of data. The UCA0TXIFG interrupt flag is set by the transmitter when the UCA0TXBUF is ready to accept a new character. Note: here we do polling on transmit buffer inside the receiver ISR. When the UCA0TXBUF is ready (UCA0TXIFG flag is set), the content from UCA0RXBUF is copied into the UCA0TXBUF. The LED4 is toggled. When exiting the ISR, the original PC and SR are retrieved bringing the processor back in the LPM0.

```
/*------------------------------------------------------------------------------
 * File:          Lab8_D2.c
 *
 * Function:      Echo a received character, using receiver ISR.
 * Description:   This program echos the character received from UART back to UART.
 *                Toggle LED1 with every received character.
```

```
 *                 Baud rate: low-frequency (UCOS16=0);
 *                 1048576/115200 = ~9.1 (0x0009|0x01)
 * Clocks:         ACLK = LFXT1 = 32768Hz, MCLK = SMCLK = default DCO
 *
 * Instructions: Set the following parameters in putty
 * Port: COM1
 * Baud rate: 115200
 * Data bits: 8
 * Parity: None
 * Stop bits: 1
 * Flow Control: None
 *
 *         MSP430f5529
 *      -----------------
 * /|\ |             XIN|-
 *  |  |                | 32kHz
 *  |--|RST         XOUT|-
 *     |                |
 *     |    P3.3/UCA0TXD|------------>
 *     |                | 115200 - 8N1
 *     |    P3.4/UCA0RXD|<------------
 *     |            P1.0|----> LED1
 *
 * Input:     None (Type characters in putty/MobaXterm/hyperterminal)
 * Output:    Character echoed at UART
 * Author:    A. Milenkovic, milenkovic@computer.org
 * Date:      October 2018
 *------------------------------------------------------------------------------*/
#include <msp430.h>

// Initialize USCI_A0 module to UART mode
void UART_setup(void) {

    P3SEL |= BIT3 + BIT4;    // Set USCI_A0 RXD/TXD to receive/transmit data
    UCA0CTL1 |= UCSWRST;     // Set software reset during initialization
    UCA0CTL0 = 0;            // USCI_A0 control register
    UCA0CTL1 |= UCSSEL_2;    // Clock source SMCLK

    UCA0BR0 = 0x09;          // 1048576 Hz  / 115200 lower byte
    UCA0BR1 = 0x00;          // upper byte
    UCA0MCTL |= UCBRS0;      // Modulation (UCBRS0=0x01, UCOS16=0)

    UCA0CTL1 &= ~UCSWRST;    // Clear software reset to initialize USCI state machine
    UCA0IE |= UCRXIE;        // Enable USCI_A0 RX interrupt
}

void main(void) {
    WDTCTL = WDTPW + WDTHOLD;// Stop WDT
    P1DIR |= BIT0;           // Set P1.0 to be output
    UART_setup();            // InitiAlize USCI_A0 in UART mode

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

// Echo back RXed character, confirm TX buffer is ready first
```

```c
#pragma vector = USCI_A0_VECTOR
__interrupt void USCIA0RX_ISR (void) {
    while(!(UCA0IFG&UCTXIFG));  // Wait until can transmit
    UCA0TXBUF = UCA0RXBUF;      // TXBUF <-- RXBUF
    P1OUT ^= BIT0;              // Toggle LED1
}

```

Figure 2. Echoing a Character Using the USCI Device

2 Real-Time Clock

In this section we will describe a program that implements a real-time clock on the MSP430 platform (Figure 3). The time is measured from the beginning of the application with a resolution of 100 milliseconds (one tenth of a second). The time is maintained in two variables, unsigned int sec (for seconds) and unsigned char tsec for tenths of a second. What is the maximum time you can have in this case? To observe the clock we can display it either on the LCD or send it serially to a workstation using a serial communication interface. In our example we send time through a serial asynchronous link using the MSP430's USCI (Universal Serial Communication Interface) device. This device is connected to a RS232 interface (see MSP EXP430F5529LP schematic) that connects through a serial cable to a PC. On the PC side we can open putty application and observe real-time clock that is sent from our development platform.

The first step is to initialize the USCI device in UART mode for communication using a baud rate 9,600 bits/sec. The next step is to initialize Timer_A to measure time and update the real-time clock variables. The Timer_A ISR is used to maintain the clock and wake up the processor. In the main program, the local variables are taken and converted into a readable string that is then sent to the USCI device.

```c
/*------------------------------------------------------------------------------
/*------------------------------------------------------------------------------
 * File:         Lab8_D3.c
 * Function:     Displays real-time clock in serial communication client.
 * Description:  This program maintains real-time clock and sends time
 *               (10 times a second) to the workstation through
 *               a serial asynchronous link (UART).
 *               The time is displayed as follows: "sssss:tsec".
 *
 *               Baud rate divider with 1048576hz = 1048576/(16*9600) = ~6.8 [16
from UCOS16]
 * Clocks:       ACLK = LFXT1 = 32768Hz, MCLK = SMCLK = default DCO = 1048576Hz
 * Instructions: Set the following parameters in putty/hyperterminal
 * Port: COMx
 * Baud rate: 9600
 * Data bits: 8
 * Parity: None
 * Stop bits: 1
 * Flow Control: None
 *
 *        MSP430F5529
 *     -----------------
```

```
 * /|\ |            XIN|-
 *  |  |               | 32kHz
 *  |--|RST        XOUT|-
 *     |               |
 *     |   P3.3/UCA0TXD|------------>
 *     |               | 9600 - 8N1
 *     |   P3.4/UCA0RXD|<------------
 *     |           P1.0|----> LED1
 *
 * Author:      A. Milenkovic, milenkovic@computer.org
 * Date:        October 2018
--------------------------------------------------------------------------------*/
#include <msp430.h>
#include <stdio.h>

// Current time variables
unsigned int sec = 0;               // Seconds
unsigned int tsec = 0;              // 1/10 second
char Time[8];                       // String to keep current time

void UART_setup(void) {
    P3SEL = BIT3+BIT4;                          // P3.4,5 = USCI_A0 TXD/RXD
    UCA0CTL1 |= UCSWRST;                        // **Put state machine in reset**
    UCA0CTL1 |= UCSSEL_2;                       // SMCLK
    UCA0BR0 = 6;                                // 1MHz 9600 (see User's Guide)
    UCA0BR1 = 0;                                // 1MHz 9600
    UCA0MCTL = UCBRS_0 + UCBRF_13 + UCOS16;     // Mod. UCBRSx=0, UCBRFx=0,
                                                // over sampling
    UCA0CTL1 &= ~UCSWRST;                       // **Initialize USCI state machine**
}

void TimerA_setup(void) {
    TA0CTL = TASSEL_2 + MC_1 + ID_3; // Select SMCLK/8 and up mode
    TA0CCR0 = 13107;                 // 100ms interval
    TA0CCTL0 = CCIE;                 // Capture/compare interrupt enable
}

void UART_putCharacter(char c) {
    while (!(UCA0IFG&UCTXIFG));     // Wait for previous character to transmit
    UCA0TXBUF = c;                   // Put character into tx buffer
}

void SetTime(void) {
    tsec++;
    if (tsec == 10){
        tsec = 0;
        sec++;
        P1OUT ^= BIT0;               // Toggle LED1
    }
}

void SendTime(void) {
    int i;
    sprintf(Time, "%05d:%01d", sec, tsec);// Prints time to a string

```

```
    for (i = 0; i < sizeof(Time); i++) {  // Send character by character
        UART_putCharacter(Time[i]);
    }
    UART_putCharacter('\r');        // Carriage Return
}

void main(void) {
    WDTCTL = WDTPW + WDTHOLD;       // Stop watchdog timer
    UART_setup();                   // Initialize UART
    TimerA_setup();                 // Initialize Timer_B
    P1DIR |= BIT0;                  // P1.0 is output;

    while (1) {
        _BIS_SR(LPM0_bits + GIE);   // Enter LPM0 w/ interrupts
        SendTime();                 // Send Time to HyperTerminal/putty
    }
}

#pragma vector = TIMER0_A0_VECTOR
__interrupt void TIMERA_ISA(void) {
    SetTime();                       // Update time
    _BIC_SR_IRQ(LPM0_bits);          // Clear LPM0 bits from 0(SR)
}

```

Figure 3. Display Real-Time Clock Through UART

Please note that sprintf with modifiers requires full printf support. This should have been already set by you when creating the project. If you did not, it is under MSP430 Compiler->Advanced Options->Language Options as shown in Figure 4.

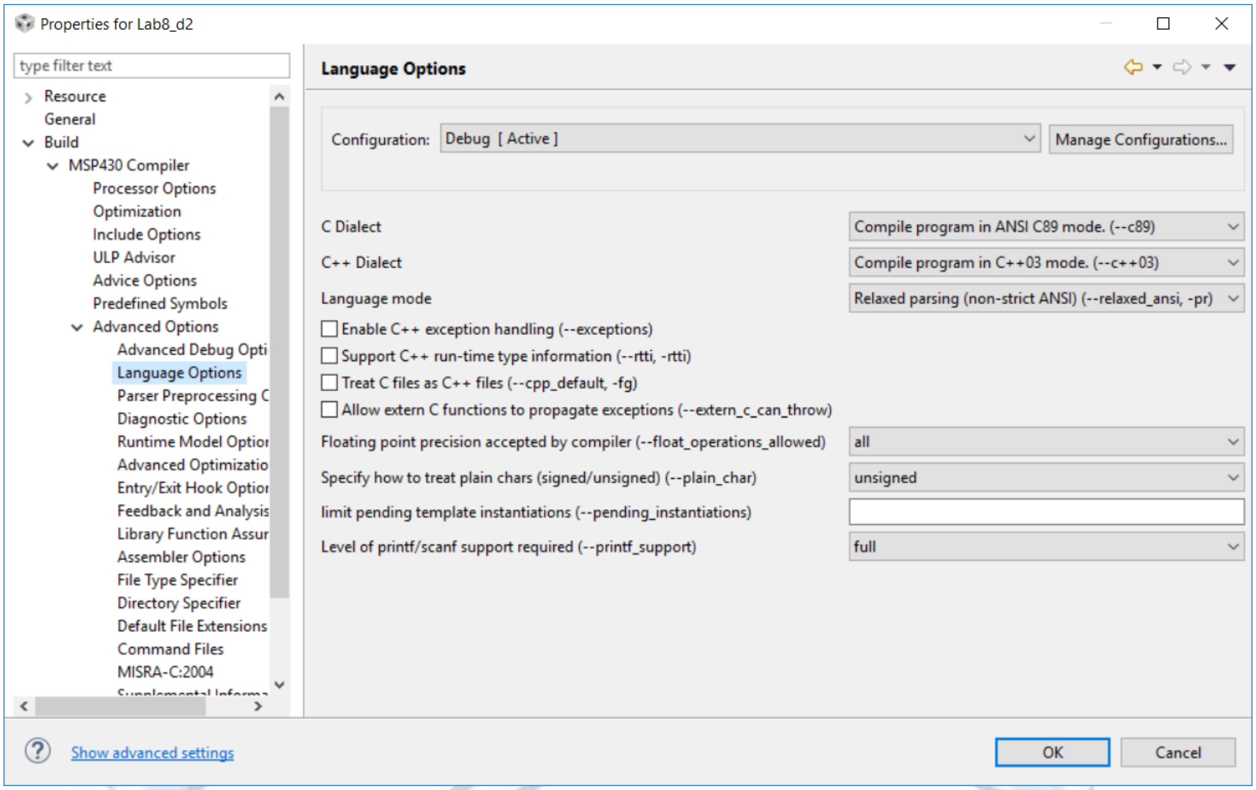


Figure 4. Setting Code Composer to Support sprintf

3 Putty versus Serial App

As a final note, it's important to keep in mind how information is being sent through the UART connection. As we begin this lab, we will generally use the Putty application (available for download at http://portal.mhealth.uah.edu/public/index.php/serial-port-application). The Putty application can only display ASCII characters. Since the UART communication protocol sends 8-bit chunks of information, the USCI peripheral has buffers that are best suited to sending or receiving 1-byte size data (with the added stop bits, etc.). It is simplest, therefore, to send and receive ASCII characters as they are a convenient 8-bit size. Putty can only handle character data types. If it receives non-character information, it will be interpreted as characters and gibberish will appear on the screen.

However, we do not always want to send characters – we often want to send and view data of different types (ints, floats, etc.). To view this type of information, we can use the convenient UAH Serial Application developed by our former student Mladen Milosevic. This application translates serial packets that are sent to it, and it can graphically represent the data versus time. Being able to construct packets with the MSP430 and read them with a software application is an important part of communication.

Because the UART protocol specifies that data is sent in 1-byte chunks, we must create a larger structure of information that we'll send. This is called a packet. The packet consists of predetermined bytes that we construct and tell the receiving software application how to interpret. The UAH Serial Application expects a packet that has a 1-byte header followed by the data followed by an optional checksum. The software must be told how many bytes of information to expect as well as the type and number of data was sending and how it's ordered. To send the data from the MSP430, we first send our header byte followed by our data that has been broken up into 1-byte chunks. The USCI UART buffer will then be fed each byte at a time. It is important in this process to ensure that the packet that you are sending has the same structure that the receiving device is expecting.

Figure 5 shows a demo program for sending a floating-point variable through UART. The 4-byte float variable is sent in a 5-byte packet: header (1-byte) and 4-byte data (LSB byte is sent first). The variable is increased by 0.1 every second with modulus 10.0 and reported through UART as shown in the WDTISR.

```
/*------------------------------------------------------------------------------
 * File:        Lab08_D4.c
 * Function:    Send floating data to Serial port
 * Description: UAH serial app expects lower byte first so send each byte at a
 *              time sending lowest byte first. Serial App can be downloaded at
 *              portal.mhealth.uah.edu/public/index.php/serial-port-application

 *              FOR PROPER OPERATION:
 *              In the UAH serial app, please do the following:
 *              1. In the "Active Session" tab, select the appropriate Com Port.
 *              2. Check the box that says "Enable Chart"
 *              3. In "Settings" tab,
 *                 a. Number of Channels = 1
 *                    i. We are sending a single floating point number to plot
 *                 b. Packet Size = 5 bytes
 *                    i. 1 byte header and 4 bytes for the floating point number
 *                 c. Header = 85 or 0x55
 *                    i. Header byte we are sending at the beginning of each
 *                       packet from our program
 *                 d. In CH0, select Type = Single 32bit
 *                 e. Set the position to 1
 *                 f. Check Show on Graph
 * Instruction: Set the following parameters in putty/hyperterminal
 * Port:        COMx
 * Baud rate:   115200
 * Data bits:   8
 * Parity:      None
 * Stop bits:   1
 * Flow Ctrl:   None
 * Clocks:      ACLK = LFXT1 = 32768Hz, MCLK = SMCLK = default DCO
 *                      MSP-EXP430F5529LP
 *                     --------------------
 *                 /|\|              XIN|-
 *                  | |                 | 32kHz
 *                  --|RST          XOUT|-
 *                    |                 |
 *                    |     P3.3/UCA0TXD|------------>
 *                    |                 | 115200 - 8N1
 *                    |     P3.4/UCA0RXD|<------------
```

```
 *                      |             P1.0|----> LED1
 * Input:       None
 * Output:      Ramp signal in UAH Serial app
 * Author(s):   Prawar Poudel, prawar.poudel@uah.edu
 * Date:        October 2018
 * --------------------------------------------------------------------------*/
#include <msp430.h>
#include <stdint.h>

volatile float myData;

void UART_setup(void)
{

    P3SEL |= BIT3 + BIT4;    // Set USCI_A0 RXD/TXD to receive/transmit data
    UCA0CTL1 |= UCSWRST;     // Set software reset during initialization
    UCA0CTL0 = 0;            // USCI_A0 control register
    UCA0CTL1 |= UCSSEL_2;    // Clock source SMCLK

    UCA0BR0 = 0x09;          // 1048576 Hz  / 115200 lower byte
    UCA0BR1 = 0x00;          // upper byte
    UCA0MCTL = 0x02;         // Modulation (UCBRS0=0x01, UCOS16=0)

    UCA0CTL1 &= ~UCSWRST;    // Clear software reset to initialize USCI state machine
}

void UART_putCharacter(char c)
{
    while (!(UCA0IFG&UCTXIFG)); // Wait for previous character to transmit
    UCA0TXBUF = c;              // Put character into tx buffer
}

int main()
{
    WDTCTL = WDT_ADLY_1000;
    UART_setup();               // Initialize USCI_A0 module in UART mode
    SFRIE1 |= WDTIE;               // Enable watchdog interrupts

    myData = 0.0;
    __bis_SR_register(LPM0_bits + GIE);
}

// Sends a ramp signal; amplitude of one period ranges from 0.0 to 9.9
#pragma vector = WDT_VECTOR
__interrupt void watchdog_timer(void)
{
    char index = 0;
    // Use character pointers to send one byte at a time
    char *myPointer = (char* )&myData;

    UART_putCharacter(0x55);              // Send header
    for(index = 0; index < 4; index++)
    {    // Send 4-bytes of myData
        UART_putCharacter(myPointer[index]);
    }
```

```
    // Update myData for next transmission
    myData = (myData + 0.1);
    if(myData >= 10.0)
    {
        myData = 0.0;
    }
}
```

Figure 5. MSP430 Program for Sending Floating-Point Data (UAH Serial App)

Figure 6 shows how to properly configure UAH Serial App for viewing the RAMP signal. We are using a single channel, the size of the packet is 5 bytes, we are plotting one sample at a time (they are arriving rather slowly in this example). Figure 7 shows the RAMP signal in the UAH Serial App sent by the program from Figure 5.

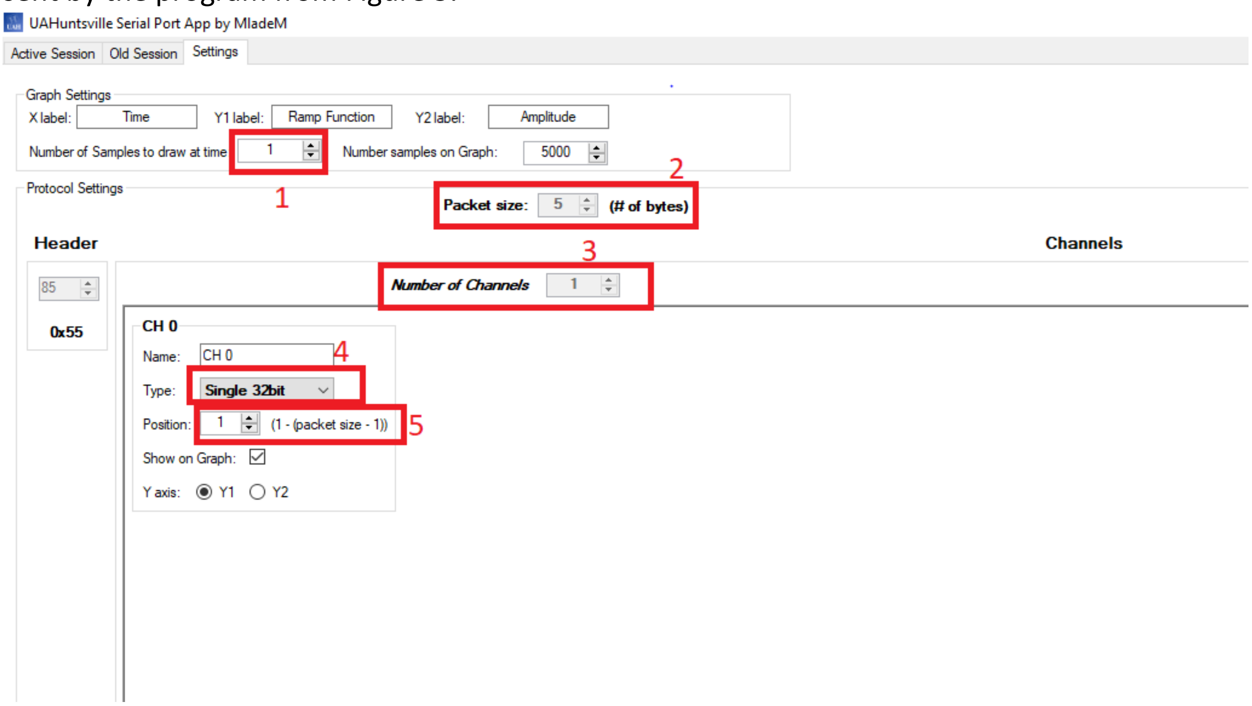


Figure 6. Configuring UAH Serial App for Viewing Ramp Signal

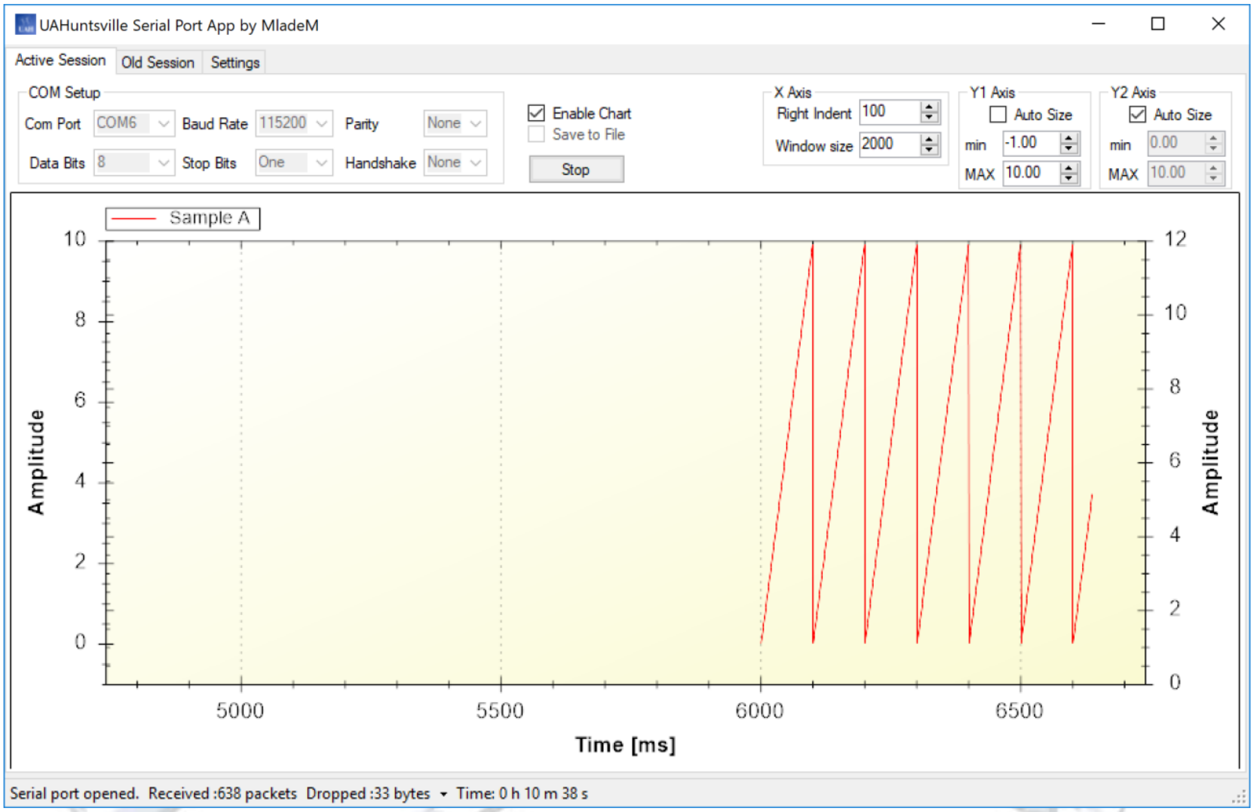


Figure 7. The RAMP signal in UAH Serial App

4 References

To understand more about UART communication and the USCI peripheral device, please read the following references:

- Davies' *MSP430 Microcontroller Basics*, pages 493 – 497 and pages 574 – 590
- MSP430 User's Guide, Chapter 36, pages 937– 966

CPE 325: Embedded Systems Laboratory
Laboratory #9 Tutorial
Analog-to-Digital Converter

Aleksandar Milenković
Email: milenka@uah.edu
Web: http://www.ece.uah.edu/~milenka

Objective

This tutorial will introduce the configuration and operation of the MSP430 12-bit analog-to-digital converter (ADC12). Programs will demonstrate the use of ADC12 to interface an on-board temperature sensor as well as external analog inputs. Specifically, you will learn how to:

Configure the ADC12
Choose reference voltages to maximize signal resolution
Create waveform lookup table in MATLAB
Interface of an on-board temperature sensor
Interface external analog signal inputs

Notes

All previous tutorials are required for successful completion of this lab, especially the tutorials introducing the TI Experimenter's board, UART communication, and Timer_A.

Contents

1 Analog-to-Digital Converters

The world around us is analog. Sensors or transducers convert physical quantities such as, temperature, force, light, sound, and others, into electrical signals, typically voltage signals that we can measure. Analog-to-digital converters allow us to interface these analog signals and convert them into digital values that can further be stored, analyzed, or communicated.

The MSP430 family of microcontrollers has a variety of analog-to-digital converters with varying features and conversion methods. In this laboratory we focus on the ADC12 converter used in the MSP430F5529. The ADC12 converter has 16 configurable input channels; 12 input channels are routed to corresponding analog input pins; remaining 4 input channels are routed to internal voltages and an on-chip temperature sensor.

1.1 ADC Resolution, Reference Voltages, and Signal Resolution

There are several key factors that should be regarded when configuring your ADC12 to most effectively read the analog signal. The first parameter you should understand is the device's voltage resolution, i.e., the smallest change of an input analog signal that causes a change in the digital output. We will be using the ADC12 peripheral that has a vertical resolution of 12 bits. That means that it can distinguish between 2^{12} (0 to 4095) input voltage levels. An A/D converter described as "n-bit" can distinguish between 0 and 2^n-1 voltage steps.

After acknowledging your ADC vertical resolution, the reference voltages need to be set. Setting the reference voltages dials in the minimum and maximum values read by the ADC. For instance, you could set your V_- to -5V and your V_+ to 10 V. With that setup on the ADC12, the numerical sampled value 0 would correspond to a signal input of -5 V, and a sampled value of 4095 would correspond to a 10 V input.

It is very important to characterize the input signal you are expecting before you set up your ADC. If you expect a signal input between 0 V and 3 V, you should set your reference voltages to 0 V and 3 V. If you set them to -5V and +5V, you would be wasting a large amount of your sample "bit depth," and your overall sample resolution would suffer because your sample input values would stay between 2048 and 3275. There would only be (3275–2048=1227) steps of resolution for your input signal rather than 4095 if you choose 0 V and 3 V as your reference voltages.

An ADC typically relies on a timer to periodically generate a trigger to start sampling of the incoming signals. You should choose a timer period that triggers sampling frequently enough to recreate the original input signals (the minimum sampling frequency should be at least two times the frequency of the signal's largest harmonic).

1.2 On-Chip Temperature Sensor

The MSP430's ADC12 has an internal temperature sensor that creates an analog voltage proportional to its temperature. A sample transfer characteristic of the temperature sensor in a different MSP430 (namely MSP430FG4618) is shown in Figure 1. The output of the temperature sensor is connected to the input multiplexor channel 10 (INCHx=1010 which is true for MSP430F5529 as well). When using the temperature sensor, the sample time (the time ADC12 is looking at the analog signal) must be greater than 30 μs. From the transfer characteristic, we get that the temperature in degrees Celsius can be expressed as $TEMPC = \frac{\mathrm{VTEMP} - 986\ mV}{3.55\ mV}$, where V_{TEMP} is the voltage from the temperature sensor (in milivolts). The transfer characteristic mentioned in Figure 1 is expressed in Volts.

The ADC12 transfer characteristic gives the following equation: $ADCResult = 4095 \cdot \frac{V_{TEMP}}{V_{REF}}$, or $V_{TEMP} = V_{REF} \cdot \frac{ADCResult}{4095}$. This can easily be deduced using the relation $ADCResult = (2\hat{\ }n - 1) \cdot \frac{V_{TEMP - V_{REF-}}}{V_{REF+} - V_{REF-}}$.

By using the internal voltage generator V_{REF+}=1,500 mV (1.5 V) and V_{REF-} = 0V, we can derive temperature as follows: $TEMPC = \frac{(ADCResult - 2692) \cdot 423}{4095}$.

Make sure your calculations match the equation given. How would equation change if instead of using V_{REF}=1.5 V we use V_{REF}=2.5 V?

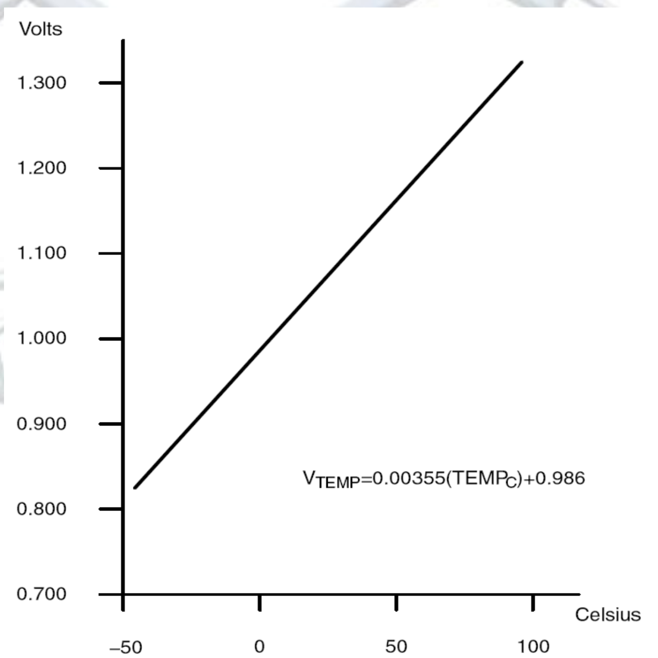


Figure 1. Internal Temperature Sensor Transfer Characteristic: V=f(T) for MSP430F4618

For MSP430F5529, since the transfer characteristic given does not present the relation between the input temperature and output voltage generated by the sensor. Thus, we will be using a slightly different approach. In the sample code presented as Lab10_D1 in Figure 3, we use double intercept form of the given characteristic to determine the transfer function. Since the reference voltage is known to us, we can consult Page 106 of the datasheet of MSP430F5529 to refer to the values we need.

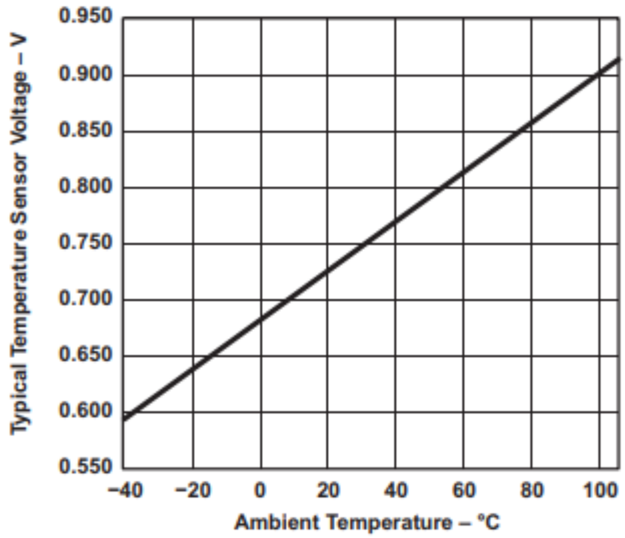


Figure 2. Temperature Sensor Transfer Function for MSP430F5529

In the C application shown in Figure 3 that samples the on-chip temperature sensor, converts the sampled voltage from the sensor to temperature in degrees Celsius and Fahrenheit, and sends the temperature information through a RS232 link to the Putty or MobaXterm application. Analyze the program and test it on the TI Experimenter’s Board. Answer the following questions.

What does the program do?
What are configuration parameters of ADC12 (input channel, clock, reference voltage, sampling time, ...)?
What are configuration parameters of the USART0 module?
How does the temperature sensor work?

```
/*------------------------------------------------------------------------------
 * File:        Lab10_D1.c (CPE 325 Lab10 Demo code)
 *
 * Function:    Measuring the temperature (MPS430F5529)
 *
 * Description: This C program samples the on-chip temperature sensor and
 *              converts the sampled voltage from the sensor to temperature in
 *              degrees Celsius and Fahrenheit. The converted temperature is
 *              sent to HyperTerminal over the UART by using serial UART.
 *
 * Clocks:      ACLK = LFXT1 = 32768Hz, MCLK = SMCLK = DCO = default (~1MHz)
 *              An external watch crystal between XIN & XOUT is required for ACLK
 *
 * Instructions:Set the following parameters in HyperTerminal
 *                  Port :        COM1
 *                  Baud rate :   115200
 *                  Data bits:    8
 *                  Parity:       None
 *                  Stop bits:    1
 *                  Flow Control: None
```

```
 *
 *                         MSP430F5529
 *                      -----------------
 *                  /|\|              XIN|-
 *                   | |                 | 32kHz
 *                   --|RST          XOUT|-
 *                     |                 |
 *                     |     P3.3/UCA0TXD|------------>
 *                     |                 | 115200 - 8N1
 *                     |     P3.4/UCA0RXD|<------------
 *                     |                 |
 * Input:       Character Y or y or N or n
 *
 * Output:      Displays Temperature in Celsius and Fahrenheit in HyperTerminal
 * Author:      Aleksandar Milenkovic, milenkovic@computer.org
 *              Prawar Poudel
 *------------------------------------------------------------------------------*/

#include  <msp430.h>
#include  <stdio.h>

#define CALADC12_15V_30C  *((unsigned int *)0x1A1A)   // Temperature Sensor
Calibration-30 C
                                                      //See device datasheet for TLV
table memory mapping
#define CALADC12_15V_85C  *((unsigned int *)0x1A1C)   // Temperature Sensor
Calibration-85 C

char ch;                    // Holds the received char from UART
unsigned char rx_flag;      // Status flag to indicate new char is received

char gm1[] = "Hello! I am an MSP430. Would you like to know my temperature? (Y|N)";
char gm2[] = "Bye, bye!";
char gm3[] = "Type in Y or N!";

long int temp;                       // Holds the output of ADC
long int IntDegF;                    // Temperature in degrees Fahrenheit
long int IntDegC;                    // Temperature in degrees Celsius

char NewTem[25];

void UART_setup(void) {

    P3SEL |= BIT3 + BIT4;   // Set USCI_A0 RXD/TXD to receive/transmit data
    UCA0CTL1 |= UCSWRST;    // Set software reset during initialization
    UCA0CTL0 = 0;           // USCI_A0 control register
    UCA0CTL1 |= UCSSEL_2;   // Clock source SMCLK

    UCA0BR0 = 0x09;         // 1048576 Hz  / 115200 lower byte
    UCA0BR1 = 0x00;         // upper byte
    UCA0MCTL = 0x02;        // Modulation (UCBRS0=0x01, UCOS16=0)

    UCA0CTL1 &= ~UCSWRST;   // Clear software reset to initialize USCI state machine
    UCA0IE |= UCRXIE;                   // Enable USCI_A0 RX interrupt
}
```

```
void UART_putCharacter(char c) {
    while (!(UCA0IFG&UCTXIFG));    // Wait for previous character to transmit
    UCA0TXBUF = c;                  // Put character into tx buffer
}

void sendMessage(char* msg, int len) {
    int i;
    for(i = 0; i < len; i++) {
        UART_putCharacter(msg[i]);
    }
    UART_putCharacter('\n');        // Newline
    UART_putCharacter('\r');        // Carriage return
}

void ADC_setup(void) {
    REFCTL0 &= ~REFMSTR;                      // Reset REFMSTR to hand over control
to
                                              // ADC12_A ref control registers
    ADC12CTL0 = ADC12SHT0_8 + ADC12REFON + ADC12ON;
                                              // Internal ref = 1.5V
    ADC12CTL1 = ADC12SHP;                     // enable sample timer
    ADC12MCTL0 = ADC12SREF_1 + ADC12INCH_10;  // ADC i/p ch A10 = temp sense i/p
    ADC12IE = 0x001;                          // ADC_IFG upon conv result-ADCMEMO
    __delay_cycles(100);                       // delay to allow Ref to settle
    ADC12CTL0 |= ADC12ENC;
}

void main(void) {
    WDTCTL = WDTPW | WDTHOLD;           // Stop watchdog timer
    UART_setup();                       // Setup USCI_A0 module in UART mode
    ADC_setup();                        // Setup ADC12

    rx_flag = 0;                        // RX default state "empty"
    _EINT();                            // Enable global interrupts
    while(1) {
        sendMessage(gm1, sizeof(gm1));// Send a greetings message

        while(!(rx_flag&0x01));         // Wait for input
        rx_flag = 0;                    // Clear rx_flag
        sendMessage(&ch, 1);            // Send received char

        // Character input validation
        if ((ch == 'y') || (ch == 'Y')) {

            ADC12CTL0 &= ~ADC12SC;
            ADC12CTL0 |= ADC12SC;                   // Sampling and conversion start

            _BIS_SR(CPUOFF + GIE);      // LPM0 with interrupts enabled

            //in the following equation,
            // ..temp is digital value read
            //..we are using double intercept equation to compute the
            //.. .. temperature given by temp value
            //.. .. using observations at 85 C and 30 C as reference
```

```c
            IntDegC = (float)(((long)temp - CALADC12_15V_30C) * (85 - 30)) /
                    (CALADC12_15V_85C - CALADC12_15V_30C) + 30.0f;

            IntDegF = IntDegC*(9/5.0) + 32.0;

            // Printing the temperature on HyperTerminal/Putty
            sprintf(NewTem, "T(F)=%ld\tT(C)=%ld\n", IntDegF, IntDegC);
            sendMessage(NewTem, sizeof(NewTem));
        }
        else if ((ch == 'n') || (ch == 'N')) {
            sendMessage(gm2, sizeof(gm2));
            break;                      // Get out
        }
        else {
            sendMessage(gm3, sizeof(gm3));
        }
    }                                   // End of while
    while(1);                           // Stay here forever
}

#pragma vector = USCI_A0_VECTOR
__interrupt void USCIA0RX_ISR (void) {
    ch = UCA0RXBUF;                 // Copy the received char
    rx_flag = 0x01;                 // Signal to main
    LPM0_EXIT;
}

#pragma vector = ADC12_VECTOR
__interrupt void ADC12ISR (void) {
    temp = ADC12MEM0;               // Move results, IFG is cleared
    _BIC_SR_IRQ(CPUOFF);            // Clear CPUOFF bit from 0(SR)
}
```

Figure 3. C Program that Samples On-Chip Temperature Sensor

1.3 Example: Analog Thumbstick Configuration

The above program details configuration and use of the ADC12 for single channel use. However, many analog devices or systems would require multiple channel configurations. As an example, let us imagine an analog joystick as is used by controllers for most modern gaming consoles. So-called thumbsticks have X and Y axis voltage outputs depending on the vector of the push it receives as input. For this example, we will use a thumbstick that has 0 to 3V output in the X and Y axes. No push on either axis results in a 1.5V output for both axes. In Figure 4 below, note how a push at about 120° with around 80% power results in around 2.75V output for the Y axis and 0.8V output for the X axis.

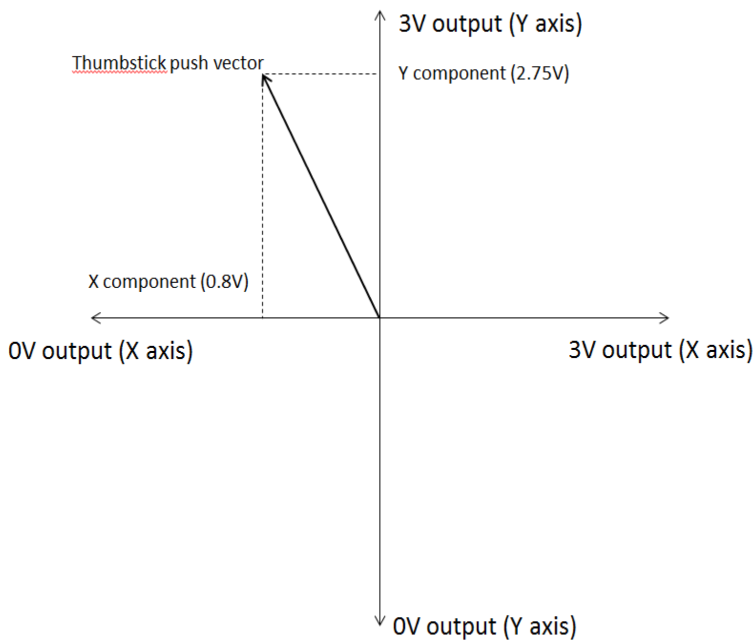


Figure 4. Performance data for hypothetical thumbstick

We want to test the thumbstick output using the UAH Serial App. To do this, we will first hook the thumbstick outputs to our device. Let's say we will use analog input A0 (P6.0) for the X axis and A1 (P6.1) for the Y axis.

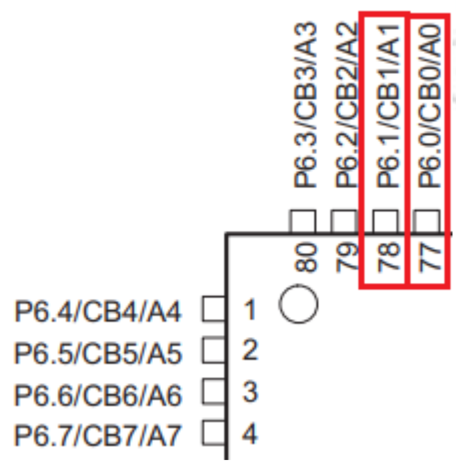


Figure 5. Pinouts and Header Connections for Analog Inputs

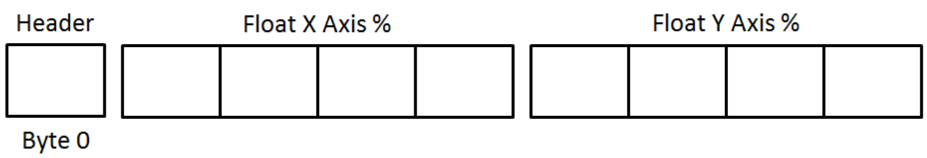
Because the outputs are from 0 to 3V, we need to set our reference voltages accordingly. We can use the board's ground and 3V supply as references.

We will want to have our output as the float datatypes because the output for each axis should be a percentage. In Figure 4, for example, the converted Y axis output would be 91.67% and the

X axis output would be 26.67%. Here is the formula you would use to convert the values (remember, the microcontroller is going to be receiving values from 0 to 4095 based on voltage values from 0V to 3V that we set as our references):

$$Input\ ADC\ Value\ in\ steps \times \frac{3V}{4095} \times \frac{100\%}{3V} = \%Power$$

We could send our information in a variety of ways including a vector format, signed percentage, or even just ADC “steps.” If we are using the percentage calculated as shown above, our packet to send to the UAH serial app would look like the one below (1 header byte, 2 single precision floating-point numbers). Figure 6 shows how to configure UAH Serial App to accept two channels including single-precision floating-point numbers. Figure 7 shows signals representing the percentage of HORZ and VERT direction of the thumbstick (read line, CH0, represents HORZ and blue line, CH1, represents VERT) when it is moved along HORZ and VERT axes. The value 100 (100%) of the red line indicates that thumbstick is moved fully in the horizontal direction.



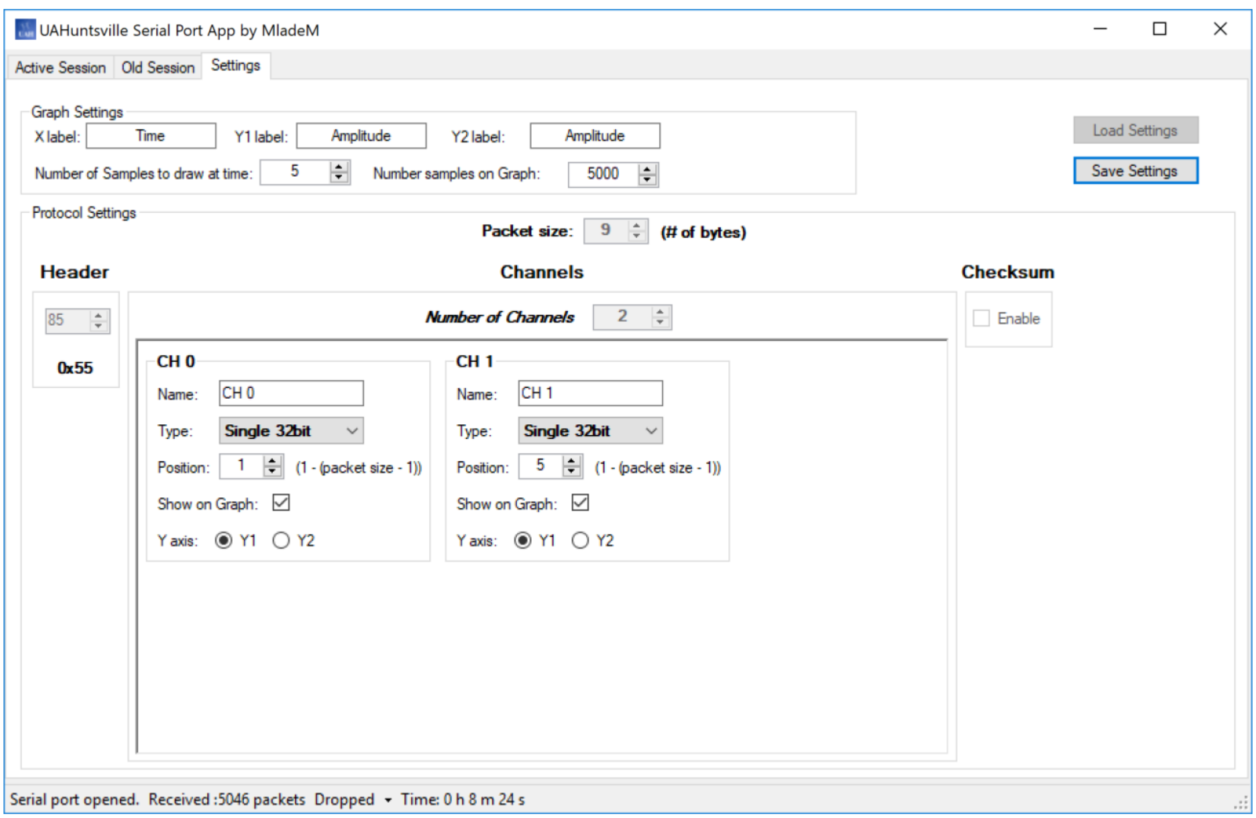


Figure 6. UAH Serial App Settings

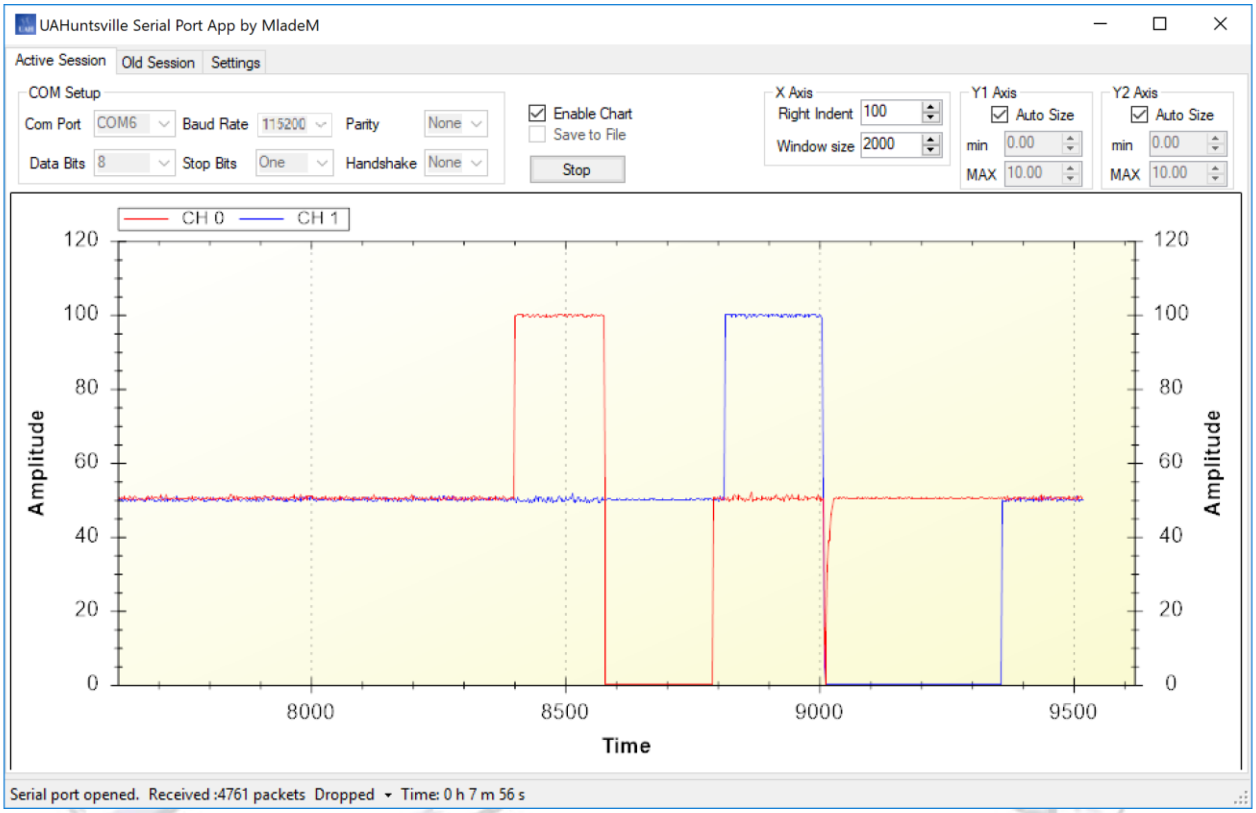


Figure 7. UAH Serial App Showing Percentage Signals from Thumbstick (CH0 – HORZ, CH1 – VERT)

Figure 8 shows demo code that could be used to set up the ADC12 and UART and send the thumbstick information to the UAH Serial App. Analyze the code and answer the following questions.

What does the program do?
What are configuration parameters of ADC12 (input channel, clock, reference voltage, sampling time, ...)?
How many samples per second is taken from ADC12?
How many samples per second per axis is sent to UAH Serial App?

You can connect your thumbstick to the pins shown in Figure 5 for use with launchpad kit. If you are using Grove Kit, note that the analog input A3 (port P6.3) corresponds to the pin Pin26 and analog input A7 (port P6.7) corresponds to the Pin 27 on the Grove Starter Kit. These pins can be accessible at J8 jumper when the Grove Kit is placed with its female connector attached to male connector of MSP-EXPF5529LP board. These pins are where we should connect horizontal HORZ and vertical VERT wires of the thumbstick when using the Grove Kit. Make sure to make appropriate changes in the source code presented in Figure 8 for use with Grove Kit.

```
/*------------------------------------------------------------------------------
 * File:        Lab09_D2.c
 * Function:    Interfacing thumbstick
```

```
 * Description: This C program interfaces with a thumbstick sensor that has
 *              x (HORZ on Thumbstick connected to P6.0) and
 *              y (VERT on Thumbstick connected to P6.1) axis
 *              and outputs from 0 to 3V. A sample joystick can be found at
 *
https://www.digikey.com/htmldatasheets/production/2262974/0/0/1/512.html
 *
 *              The value of x and y axis is sent as the percentage
 *              of power to the UAH Serial App.
 *
 * Clocks:      ACLK = LFXT1 = 32768Hz, MCLK = SMCLK = DCO = default (~1MHz)
 *              An external watch crystal beten XIN & XOUT is required for ACLK
 *                      MSP-EXP430F5529LP
 *                     --------------------
 *                   /|\|              XIN|-
 *                    | |                 | 32kHz
 *                    --|RST          XOUT|-
 *                      |                 |
 *                      |     P3.3/UCA0TXD|------------>
 *            P6.0(A0)-->|                 | 115200 - 8N1
 *            P6.1(A1)-->|     P3.4/UCA0RXD|<------------
 *                      |                 |
 * Input:       Connect thumbstick to the board
 * Output:      Displays % of power in UAH serial app
 * Author(s):   Prawar Poudel, prawar.poudel@uah.edu
 *              Micah Harvey
 * Date:        August 8, 2020
 *--------------------------------------------------------------------------*/
#include <msp430.h>

volatile long int ADCXval, ADCYval;
volatile float Xper, Yper;

void TimerA_setup(void)
{
    TA0CCTL0 = CCIE;                    // Enabled interrupt

    TA0CCR0 = 3277;                     // 3277 / 32768 Hz = 0.1s
    TA0CTL = TASSEL_1 + MC_1;           // ACLK, up mode
}


void ADC_setup(void)
{
    // configure ADC converter
    P6SEL = 0x03;                            // Enable A/D channel inputs
    ADC12CTL0 = ADC12ON+ADC12MSC+ADC12SHT0_8; // Turn on ADC12, extend sampling time
                                             // to avoid overflow of results

    ADC12CTL1 = ADC12SHP+ADC12CONSEQ_1;      // Use sampling timer, repeated
sequence
    ADC12MCTL0 = ADC12INCH_0;                // ref+=AVcc, channel = A0
    ADC12MCTL1 = ADC12INCH_1+ADC12EOS;       // ref+=AVcc, channel = A1, end seq.

    ADC12IE = 0x02;                          // Enable ADC12IFG.1
```

```
    ADC12CTL0 |= ADC12ENC;                    // Enable conversions
}


void UART_putCharacter(char c)
{
    while (!(UCA0IFG&UCTXIFG));    // Wait for previous character to transmit
    UCA0TXBUF = c;                  // Put character into tx buffer
}


void UART_setup(void)
{
    P3SEL |= BIT3 + BIT4;   // Set USCI_A0 RXD/TXD to receive/transmit data

    UCA0CTL1 |= UCSWRST;    // Set software reset during initialization
    UCA0CTL0 = 0;           // USCI_A0 control register
    UCA0CTL1 |= UCSSEL_2;   // Clock source SMCLK

    UCA0BR0 = 0x09;         // 1048576 Hz  / 115200 lower byte
    UCA0BR1 = 0x00;         // upper byte
    UCA0MCTL |= UCBRS0;     // Modulation (UCBRS0=0x01, UCOS16=0)

    UCA0CTL1 &= ~UCSWRST;   // Clear software reset to initialize USCI state machine
}


void sendData(void)
{
    int i;
    Xper = (ADCXval*3.0/4095*100/3);    // Calculate percentage outputs
    Yper = (ADCYval*3.0/4095*100/3);

    // Use character pointers to send one byte at a time
    char *xpointer=(char *)&Xper;
    char *ypointer=(char *)&Yper;

    UART_putCharacter(0x55);            // Send header
    for(i = 0; i < 4; i++)
    {             // Send x percentage - one byte at a time
        UART_putCharacter(xpointer[i]);
    }
    for(i = 0; i < 4; i++)
    {             // Send y percentage - one byte at a time
        UART_putCharacter(ypointer[i]);
    }
}


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

    // Enable interrupts globally
    __enable_interrupt();
```

```
    ADC_setup();                        // Setup ADC
    UART_setup();
    TimerA_setup();

    while (1)
    {
        __bis_SR_register(LPM0_bits + GIE); // Enter LPM0
        sendData();
    }
}

#pragma vector = ADC12_VECTOR
__interrupt void ADC12ISR(void)
{
    ADCXval = ADC12MEM0;                    // Move results, IFG is cleared
    ADCYval = ADC12MEM1;
    __bic_SR_register_on_exit(LPM0_bits); // Exit LPM0
}

#pragma vector = TIMER0_A0_VECTOR
__interrupt void timerA_isr()
{
    ADC12CTL0 |= ADC12SC;
}


```

Figure 8 C Program that takes the x- and y- axis Samples from a Thumbstick

3 References

To understand more about the ADC12 peripheral and its configuration, please refer the following materials:

- Davies Text, pages 407-438 and pages 485-492
- MSP430x5xx User’s Guide, Chapter 28, pages 730-760 (ADC12)
- MSP430x5xx User’s Guide, page 744 (Internal temperature sensor)