CPE 323
MSP430 Interrupts

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

Objective:
Introduce Interrupt Mechanism:
Topics: Type of interrupts, Tracking Pending Interrupts, Masking (Selective and Global), Exception Processing, Interrupt Service Routines, Interrupt Vector Table (Location, Initialization)

Contents

1 Introduction

Exception or interrupts are events caused by either hardware or software that require urgent response. These events are typically asynchronous to program execution, i.e., they can occur at any time during instruction execution. In addition, multiple such events can arise concurrently. These events can arise from within a processor – e.g., when trying to execute a non-existing instruction (due to corruption of opcode in memory) or when trying to divide by a zero, or from outside – e.g., a switch is pressed, a new sample is ready in an ADC, or a communication interface is ready to sending a character. The processors handle such events by stopping execution of current programs and handling these events in so called interrupt service routines (ISRs). Once processing is finished, the processor should continue execution of the interrupted program like this departure has never happened.

In embedded systems, exceptions or interrupts are crucial as we are interfacing real-world (e.g., sensors and actuators) and the majority of tasks is triggered by external events that are asynchronous to programs that are being executed. Exception processing deals with questions such as:

- How does a processor keep track of pending interrupt requests?
- How does a processor decide which request to accept in presence of multiple pending requests?
- What steps are performed to ensure continuation of program execution once the accepted request is handled?
- How does a processor locate starting addresses of routines that handle interrupt requests?
- How do interrupt service routines look like?

In this section we will look at the exception processing in MSP430 and explain both hardware and software aspects on exception processing and how to write interrupt service routines.

2 Sources of Interrupts

As mentioned above, interrupts can come from inside the CPU or from input/output peripherals. In general, we use interrupts to:

- Handle urgent tasks that need to be executed at higher priority than the main code.
  For example, we need to activate brakes when a breaking pedal is pressed regardless of what the main processor is currently doing, or we need to read a data packet received by a communication device before it gets overwritten by another incoming packet.
- Handle infrequent tasks.
  For example, when scanning input from a keyboard we do not want processor to do the scanning because it will be a waste of processor time. We can enter a limited number of characters – if you type superfast, perhaps a dozen of characters in a second, whereas processors work on much higher speeds executing millions or billions of instructions in a second.

- Wake the processor up from sleep. Processors may be in sleeping modes (all clocks are turned off to conserve energy) and are awaken through interrupts.
- Call an operating system by invoking interrupts in software.

MSP430 does not have internally-generated interrupts caused by illegal arithmetic operations (e.g., division by a zero – actually, there is no divide instruction in MSP430 ISA) or illegal opcode. Interrupts are triggered by events coming from peripheral devices – timers after a specified period of times has expired, ADC converters when a new sample is ready, communication interfaces when they are ready to send a new character or when they received a new character, direct-memory-access controllers when they finished a specified transfer, and so on. Practically, all peripherals in MSP430 are capable of generating interrupt requests.

3 Tracking Pending Interrupts and Generating Interrupt Requests

To track pending interrupt requests, hardware employs so called interrupt tracking bits or flags (IFs). These bits are located in either system-wide special-purpose registers, such as IFG1 and IFG2, or peripheral-specific control registers. Each event capable of generating an interrupt has its own interrupt tracking flag. For example, the MSP430's Watchdog Timer peripheral has a corresponding WDTIFG – Watchdog Timer Interrupt Flag -- that tracks whether a predefined period of time has expired. Generally, one peripheral may have multiple interrupt flags that may be served by one or multiple interrupt service routines.

Setting an interrupt tracking flag indicates that a certain event in hardware has occurred. However, this does not mean that an interrupt request is automatically generated. Each interrupt tracking bit or flag has its corresponding Interrupt Enable bit that needs to be set to allow an interrupt to become visible to the processor. For example, WDTGIE – Watchdog Timer Interrupt Enable bit -- corresponds to the WDTIFG bit. The masking bits are located in system-wide special-purpose registers, such as IE1 and IE2, or peripheral-specific control registers. In addition, the status register SR (R2) includes a so called Global Interrupt Enable bit that allows for global masking of all maskable interrupts. Thus, to accept an interrupt from a specific event, its corresponding interrupt flag and enable flag have both to be set as well as the GIE bit in the status register. The masking bits are generally set by software developers. By selectively enabling (setting) or disabling (clearing) masking bits, software developers can fully control what hardware events can generate interrupt requests and when they can do so. Similarly, but setting/resetting the GIE bits they can globally enable or disable all maskable interrupts.

Figure 1 shows the content of two system-wide registers IFG1 and IFG2. Please notice that BIT0 of this register is actually WDTIFG that is set when the watchdog timer overflows or when a wrong key is written in its control register. The other two flags are related to detecting an oscillator fault and whether a RST#/NMI pin is active. While on this figure, please notice rw-0, rw-1, rw-(0) markings below each bit. This is a concise way to indicate that these bits can be programmatically read from (r), written to (w), and what their initial states upon powering MSP430 up are (-0 means they are cleared, -1 means they are set). The notation "-(0)" below WDTIFG means that this bit is cleared on Power-On-Reset (hard reset activated when powering

up the chip or physical reset), but that it retains its value on Power-Up-Clear (soft reset activated when the watchdog timer expires). This notation is used when describing the format and initial state of all special-purpose and peripheral-specific registers. Figure 2 illustrates the format of the system-wide interrupt enable registers, IE1 and IE2. An important implication of this is that WDTIFG can be set by executing a BIS instruction that set BIT0 to 1. This way, an interrupt request is generated from software, rather than from hardware.

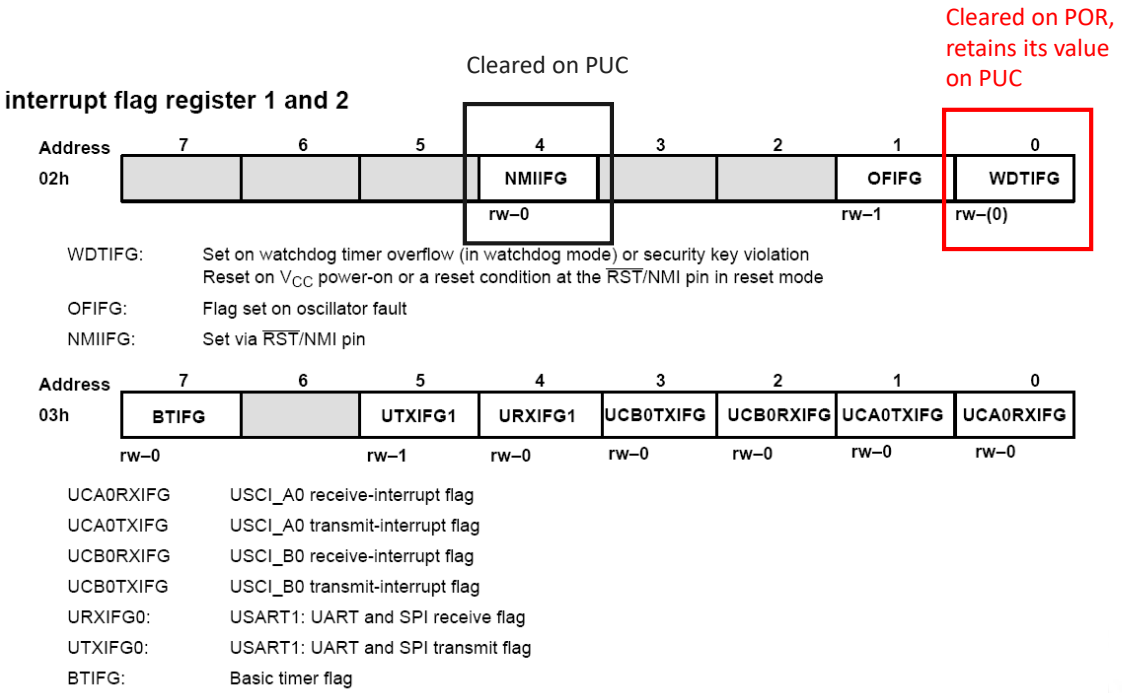

Figure 1. System-wide Interrupt Flag Registers.

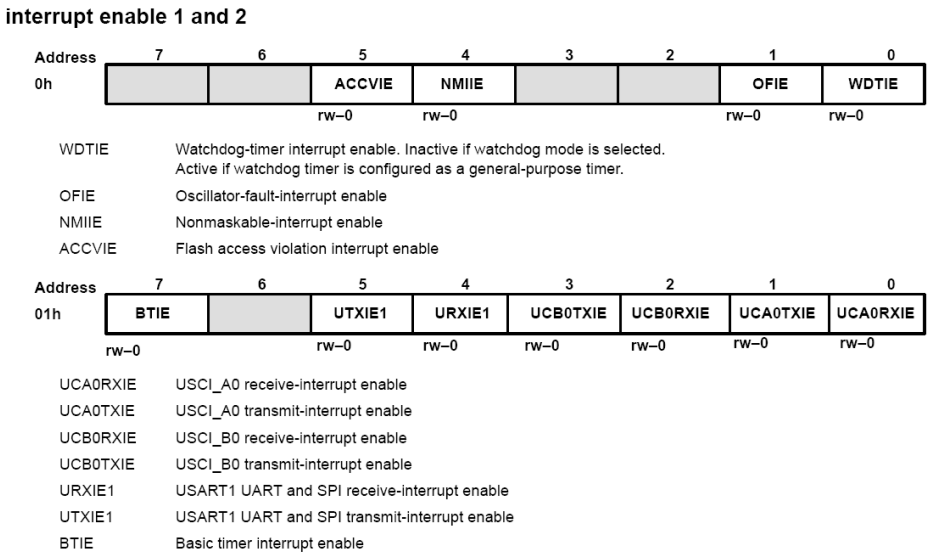

Figure 2. System-wide Interrupt Enable Registers.

What happens with interrupt flag bits and how do they get cleared? These bits remain set until the processor handles the event by executing the corresponding interrupt service routine. For so-called single-sourced interrupts – interrupts that have their own interrupt service routine – the interrupt flag bits are cleared in hardware during exception processing. For so-called multi-source interrupts when multiple hardware events are handled by a single interrupt service routine, the responsibility for clearing interrupt flag bits is on software developers who will clear flag bits based on how multiple events are handled inside the service routine.

4 Exception Processing Steps

Here we will detail the steps taken by the MSP430 processor as a response to a pending interrupt request. The following sequence of events is performed:

1. Any currently executed instruction is completed (this is the other way of saying that Exception Processing occurs at the end of an instruction execution).
2. The PC, which points to the next instruction, is pushed onto the stack.
3. The SR, status register, is pushed onto the stack.
4. The interrupt with the highest priority is selected, if multiple interrupts occurred during the last instruction and are pending for service.
5. The interrupt request flag (IF) resets automatically on single-source interrupts. Multiple-source interrupt flags remain set for servicing by software.
6. The SR is cleared with the exception of SCG0, which is left unchanged. This terminates any low-power mode. Because the GIE bit is cleared, further interrupts are disabled.
7. The content of the interrupt vector is loaded into the PC: the program continues with the interrupt service routine at that address.

This sequence of steps take 6 clock cycles to complete. There are two noteworthy implications of these steps. First is that step 6 performs clearing bits in the status register which means that, by default, other new or pending interrupts will not be considered while executing instructions inside the current service routine. In other words, by default, the nesting of interrupt service routines is not possible. However, software developers may opt to explicitly set the GIE bit inside the ISR, thus allowing other interrupts to be serviced before completing the servicing of this one. Second, if the processor was placed in a sleep mode (that is done by setting specific bits in the status register), the exception processing steps triggered by an interrupt are the only way to wake the processor up. Thus, to exit a sleep mode, we rely on interrupts. In other words, if we go into a sleep mode and do not enable any of the interrupts, we will never exit a low-power mode.

Upon completion of the exception processing steps, the processor is ready to start executing instructions inside the corresponding service routine. Interrupt service routines look like subroutines that are written to handle one or more hardware events. They are executed at unpredictable times and they should be carried out in such a way to allow the main code to

resume execution without any error, like the ISR has never happened. This means that any registers used inside the ISR should be backed up on the stack at the beginning of the ISR and restored right before exiting the ISR. Interrupt service routines do not have input or output parameters, but they can change global variables in memory.

The last machine instruction executed inside an ISR has to be RETI – Return From Interrupt. The RETI performs the following operations:

1. The SR with all previous settings pops from the stack. All previous settings of GIE, CPUOFF, etc. are now in effect, regardless of the settings used during the interrupt service routine.
2. The PC pops from the stack and begins execution at the point where it was interrupted.

The RETI instruction takes 5 clock cycles to execute.

Steps 4 and 7 are further elaborated in the following section.

5 Interrupt Vector Table and Interrupt Priorities

How do we find the starting addresses of corresponding interrupt service routines in step 7 of the exception processing? The MSP430 maintains a so-called Interrupt Vector Table (IVT) that keeps the starting addresses of interrupt service routines. This table is placed at the top of first 64 KB of address space and depending on type of MSP430 can have either 16 or 32 entries as shown in Figure 3. Each entry is one 16-bit word that contains the starting address of the corresponding interrupt service routine. The entry at the address 0xFFFE corresponds to the so-called RESET vector – the MSP430 devices fetch the content of this location upon powering up and store it in the PC to start execution. The interrupt vector table in Figure 3 contains 16 entries and each entry is given priority that corresponds to the entry number. Entry 15 (RESET vector) has the highest priority, followed by entry 14, and so on. The entry 14 contains the starting address of the ISR that handles NMI, Oscillator Fault, or Flash Memory Access Violation. This is an example of an interrupt service routine with multiple sources – these three hardware events are served by a single ISR and the code in this ISR has to investigate why the ISR is entered – which of these 3 flags is set and then handle them separately as shown in Figure 4. Also, we can notice that clearing these flags is one of the steps that has to be performed explicitly inside the ISR.

INTERRUPT SOURCE	INTERRUPT FLAG	SYSTEM INTERRUPT	WORD ADDRESS	PRIORITY
Power-up, external reset, watchdog, flash password	WDTIFG KEYV	Reset	0FFFEh	15, highest
NMI, oscillator fault, flash memory access violation	NMIIFG OFIFG ACCVIFG	(non)-maskable (non)-maskable (non)-maskable	0FFFCh	14
Device-specific			0FFFAh	13
Device-specific			0FFF8h	12
Device-specific			0FFF6h	11
Watchdog timer	WDTIFG	maskable	0FFF4h	10
Device-specific			0FFF2h	9
Device-specific			0FFF0h	8
Device-specific			0FFEEh	7
Device-specific			0FFECh	6
Device-specific			0FFEAh	5
Device-specific			0FFE8h	4
Device-specific			0FFE6h	3
Device-specific			0FFE4h	2
Device-specific			0FFE2h	1
Device-specific			0FFE0h	0, lowest

Figure 3. Interrupt Vector Table.

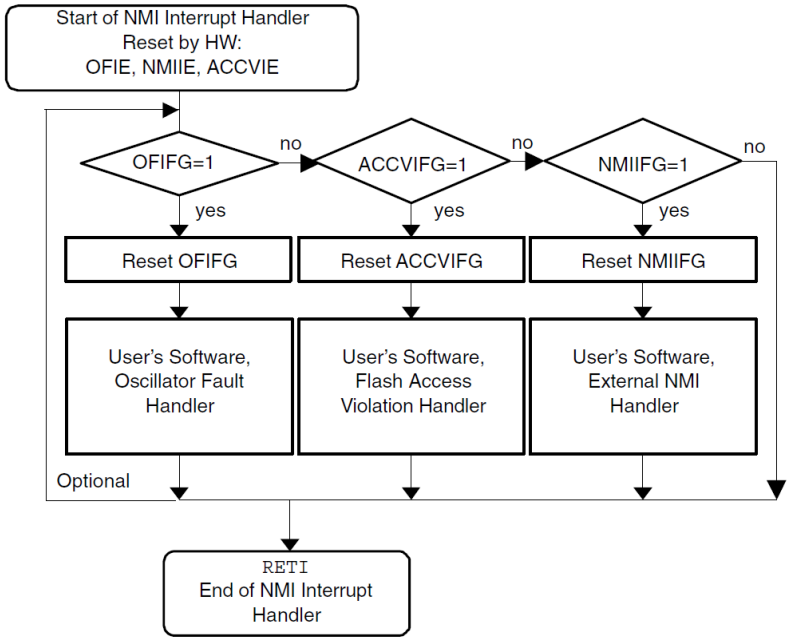

Figure 4. NMI/OF/ACCIFG Handler.

Figure 5 shows an excerpt from an include file that specifies interrupt vector table for the MSP430FG4618 device.

```
#ifdef __ASM_HEADER__ /* Begin #defines for assembler */
#define DAC12_VECTOR            ".int14"                    /* 0xFFDC DAC 12 */
#else
#define DAC12_VECTOR            (14 * 1u)                   /* 0xFFDC DAC 12 */
#endif
#ifdef __ASM_HEADER__ /* Begin #defines for assembler */
#define DMA_VECTOR              ".int15"                    /* 0xFFDE DMA */
#else
#define DMA_VECTOR              (15 * 1u)                   /* 0xFFDE DMA */
#endif
#ifdef __ASM_HEADER__ /* Begin #defines for assembler */
#define BASICTIMER_VECTOR       ".int16"                    /* 0xFFE0 Basic Timer / RTC */
#else
#define BASICTIMER_VECTOR       (16 * 1u)                   /* 0xFFE0 Basic Timer / RTC */
#endif
#ifdef __ASM_HEADER__ /* Begin #defines for assembler */
#define PORT2_VECTOR            ".int17"                    /* 0xFFE2 Port 2 */
#else
#define PORT2_VECTOR            (17 * 1u)                   /* 0xFFE2 Port 2 */
#endif
#ifdef __ASM_HEADER__ /* Begin #defines for assembler */
#define USART1TX_VECTOR         ".int18"                    /* 0xFFE4 USART 1 Transmit */
#else
#define USART1TX_VECTOR         (18 * 1u)                   /* 0xFFE4 USART 1 Transmit */
#endif
#ifdef __ASM_HEADER__ /* Begin #defines for assembler */
#define USART1RX_VECTOR         ".int19"                    /* 0xFFE6 USART 1 Receive */
#else
#define USART1RX_VECTOR         (19 * 1u)                   /* 0xFFE6 USART 1 Receive */
#endif
#ifdef __ASM_HEADER__ /* Begin #defines for assembler */
#define PORT1_VECTOR            ".int20"                    /* 0xFFE8 Port 1 */
#else
#define PORT1_VECTOR            (20 * 1u)                   /* 0xFFE8 Port 1 */
#endif
#ifdef __ASM_HEADER__ /* Begin #defines for assembler */
#define TIMERA1_VECTOR          ".int21"                    /* 0xFFEA Timer A CC1-2, TA */
#else
#define TIMERA1_VECTOR          (21 * 1u)                   /* 0xFFEA Timer A CC1-2, TA */
#endif
#ifdef __ASM_HEADER__ /* Begin #defines for assembler */
#define TIMERA0_VECTOR          ".int22"                    /* 0xFFEC Timer A CC0 */
#else
#define TIMERA0_VECTOR          (22 * 1u)                   /* 0xFFEC Timer A CC0 */
#endif
#ifdef __ASM_HEADER__ /* Begin #defines for assembler */
#define ADC12_VECTOR            ".int23"                    /* 0xFFEE ADC */
#else
#define ADC12_VECTOR            (23 * 1u)                   /* 0xFFEE ADC */
#endif
#ifdef __ASM_HEADER__ /* Begin #defines for assembler */
#define USCIAB0TX_VECTOR        ".int24"                    /* 0xFFF0 USCI A0/B0 Transmit */
#else
#define USCIAB0TX_VECTOR        (24 * 1u)                   /* 0xFFF0 USCI A0/B0 Transmit */
#endif
#ifdef __ASM_HEADER__ /* Begin #defines for assembler */
#define USCIAB0RX_VECTOR        ".int25"                    /* 0xFFF2 USCI A0/B0 Receive */
#else
#define USCIAB0RX_VECTOR        (25 * 1u)                   /* 0xFFF2 USCI A0/B0 Receive */
#endif
#ifdef __ASM_HEADER__ /* Begin #defines for assembler */
#define WDT_VECTOR              ".int26"                    /* 0xFFF4 Watchdog Timer */
#else
```

```
64  #define WDT_VECTOR              (26 * 1u)                   /* 0xFFF4 Watchdog Timer */
65  #endif
66  #ifdef __ASM_HEADER__ /* Begin #defines for assembler */
67  #define COMPARATORA_VECTOR      ".int27"                    /* 0xFFF6 Comparator A */
68  #else
69  #define COMPARATORA_VECTOR      (27 * 1u)                   /* 0xFFF6 Comparator A */
70  #endif
71  #ifdef __ASM_HEADER__ /* Begin #defines for assembler */
72  #define TIMERB1_VECTOR          ".int28"                    /* 0xFFF8 Timer B CC1-2, TB */
73  #else
74  #define TIMERB1_VECTOR          (28 * 1u)                   /* 0xFFF8 Timer B CC1-2, TB */
75  #endif
76  #ifdef __ASM_HEADER__ /* Begin #defines for assembler */
77  #define TIMERB0_VECTOR          ".int29"                    /* 0xFFFA Timer B CC0 */
78  #else
79  #define TIMERB0_VECTOR          (29 * 1u)                   /* 0xFFFA Timer B CC0 */
80  #endif
81  #ifdef __ASM_HEADER__ /* Begin #defines for assembler */
82  #define NMI_VECTOR              ".int30"                    /* 0xFFFC Non-maskable */
83  #else
84  #define NMI_VECTOR              (30 * 1u)                   /* 0xFFFC Non-maskable */
85  #endif
86  #ifdef __ASM_HEADER__ /* Begin #defines for assembler */
87  #define RESET_VECTOR            ".reset"                    /* 0xFFFE Reset [Highest Priority] */
88  #else
89  #define RESET_VECTOR            (31 * 1u)                   /* 0xFFFE Reset [Highest Priority] */
90  #endif
```

Figure 5. Interrupt Vector Table Definitions for MSP430FG4618.

6 An Example: Toggling LEDs

Let us start with a program that toggles LEDs on the TI Experimenter's board. The program shown in Figure 6 stops the watchdog timer (line 27), initializes ports P2.1 and P2.2 as outputs (line 28), turns the LEDs off (line 29), and then enters an infinite loop. A software delay of ~1s is implemented using a do-while loop (line 33-49). Inside the loop, a number of assembly NOP instructions is inserted to ensure that one loop iteration takes 16 processor clock cycles. The number of iterations is set to 65,535 so that the delay corresponds to 16*65,535 clock cycles. The default processor clock frequency is 2^{20} = 1,048,576 Hz. The total delay is thus approximately 1s. The P2OUT is xored with a constant 0x06 resulting in toggling LEDs every 1s (1s off, 1s on).

```
1   /*****************************************************************************
2    *   File:         ToggleLEDs_1sSD_C.c
3    *   Description:  Program toggles LED1 and LED2 by
4    *                 xoring port pins inside of an infinite loop.
5    *                 1s software delay is implemented using a do-while loop.
6    *   Board:        MSP430FG461x/F20xx Experimenter Board
7    *   Clocks:       ACLK = 32.768kHz, MCLK = SMCLK = default DCO
8    *
9    *                 MSP430FG461x
10   *             +-----------------+
11   *             |                 |
12   *             |                 |
```

```
13   *            |                 |
14   *            |                 |
15   *            |             P2.1|--> LED2
16   *            |             P2.2|--> LED1
17   *
18   *   Author: Aleksandar Milenkovic, milenkovic@computer.org
19   *   Date:   September 2010
20  ******************************************************************************/
21  #include  <msp430.h>
22
23  void main(void) {
24
25     unsigned int i=0;
26
27     WDTCTL = WDTPW + WDTHOLD;    // Stop watchdog timer
28     P2DIR |= (BIT1 | BIT2);      // Set P2.1 and P2.2 to output direction (0000_0110)
29     P2OUT = 0x00;                // Clear output port P2, P2OUT=0000_0000b
30     for (;;) {
31           i=0xFFFF;              // Set the number of iterations 65535
32                                  // Delay is 65535*16cc/2^20 ~ 1s
33           do {                   // The number of NOPs ensures
34                                  // that the one iteration takes 16 clock cycles
35                 asm(" nop");
36                 asm(" nop");
37                 asm(" nop");
38                 asm(" nop");
39                 asm(" nop");
40                 asm(" nop");
41                 asm(" nop");
42                 asm(" nop");
43                 asm(" nop");
44                 asm(" nop");
45                 asm(" nop");
46                 asm(" nop");
47                 asm(" nop");
48                 i--;
49           } while(i!=0);
50           P2OUT ^= (BIT1 | BIT2); // Toggle P2.1 and P2.2 using exclusive-OR
51     }
52  }
```

Figure 6. ToggleLEDs Using Software Delay.

Figure 7 shows an alternative implementation of the toggling LEDs. Instead of software delay we rather use a watchdog timer peripheral configured in line 25. The watchdog control register is set as follows: the watchdog works in its interval mode setting WDTIFG bit every 1s (input clock is ACLK which is 2^{15} Hz and the counter is set to 2^{15} clock ticks). The counter inside the watchdog timer continually counts up on every ACLK clock with modulus 2^{15}. When 2^{15} clocks are counted, the WDTIFG is set. The while loop in line 30 practically implements a wait state – the program repeatedly tests the WDTIFG bit in the IFG1 register. If WDTIFG is not set (the counter has not counted the specified number of clock cycles), the WDTIFG is checked again. Once the WDTIFG is set (after 1s), the program continues with toggling the LEDs. The WDTIFG is cleared and the process is repeated over and over. Generally, this approach to interfacing

peripherals is called polling – the processor is busy waiting for an event to occur in hardware. A better alternative is to enable interrupts from the watchdog timer.

```
1  /*******************************************************************************
2   *   File:        ToggleLEDs_1sCWDTPolling.c
3   *   Description: Program toggles LED1 and LED2 by
4   *                xoring port pins inside of an infinite loop.
5   *                1s delay is implemented using WDT in interval mode.
6   *                Polling on WDTIFG is used inside the loop to detect 1s periods.
7   *   Board:       MSP430FG461x/F20xx Experimenter Board
8   *   Clocks:      ACLK = 32.768kHz, MCLK = SMCLK = default DCO
9   *
10  *                 MSP430FG461x
11  *             +-----------------+
12  *             |                 |
13  *             |                 |
14  *             |                 |
15  *             |                 |
16  *             |             P2.1|--> LED2
17  *             |             P2.2|--> LED1
18  *
19  *   Author: Aleksandar Milenkovic, milenkovic@computer.org
20  *   Date:   September 2010
21 *******************************************************************************/
22 #include  <msp430.h>
23
24 void main(void) {
25    WDTCTL = WDT_ADLY_1000;           // 1 s interval timer
26    P2DIR |= BIT2 + BIT1;             // Set P2.1 and P2.2 to output direction
27    P2OUT = 0x00;                     // LEDs are off
28    // use polling on WDTIFG (it's set every 1s)
29    for (;;) {
30       while ((IFG1 & WDTIFG) == 0);
31       P2OUT ^= (BIT1 | BIT2);
32       IFG1 &= ~WDTIFG;               // Clear WDTIFG in IFG1
33    }
34 }
```

Figure 7. ToggleLEDs Using WDT Polling.

Figure 8 shows a program that toggles the LEDs using the watchdog timer interrupt service routine. Again the watchdog timer is configured to work in interval mode and to count to 2^{15} ACLK clock ticks (line 23). This time however we also enable interrupts from the watchdog timer (line 26). The BIT0 of IE1 keeps the WDTIE bit – by setting this bit we allow watchdog timer to generate an interrupt request once the WDTIFG bit is set. The statement in line 28 puts the processor into a sleep mode where the processor clock is off and program execution stops. When the watchdog timer counts the specified period, WDTIFG is set and as WDTIE and GIE are set too, the watchdog timer interrupt request is raised. This will trigger the processor to wake up (exception processing steps are carried out) and enter the watchdog timer interrupt service routine shown in lines 32-35. Using pragma we specify that this is the interrupt service routine for the watchdog timer (so that the corresponding entry in the interrupt vector table is properly

initialized). The ISR itself includes just a single statement that toggles the LEDs. Please note that since this is a single-source interrupt service routine, the WDTIFG bit is cleared automatically in exception processing.

```
/*******************************************************************************
 *   File:        ToggleLEDs_1sCWDTISR.c
 *   Description: Program toggles LED1 and LED2 inside the WDT_ISR (interval mode).
 *                WDT is set to raise an interrupt every 1s.
 *   Board:       MSP430FG461x/F20xx Experimenter Board
 *   Clocks:      ACLK = 32.768kHz, MCLK = SMCLK = default DCO
 *
 *                 MSP430FG461x
 *             +-----------------+
 *             |                 |
 *             |                 |
 *             |                 |
 *             |                 |
 *             |             P2.1|--> LED2
 *             |             P2.2|--> LED1
 *
 *   Author: Aleksandar Milenkovic, milenkovic@computer.org
 *   Date:   September 2010
*******************************************************************************/
#include <msp430xG46x.h>

void main(void) {
    WDTCTL = WDT_ADLY_1000;                 // 1s interval
    P2DIR |= BIT2 + BIT1;                   // Set P2.2 and P2.1 to output direction
    P2OUT = 0x00;                           // LEDs are off
    IE1 |= WDTIE;                           // Enable WDT interrupt (WDTIE is set)

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

// Watchdog Timer interrupt service routine
#pragma vector=WDT_VECTOR
__interrupt void watchdog_timer(void) {
    P2OUT ^= (BIT2 | BIT1);             // Toggle P2.1 and P2.2 using exclusive-OR
}

```

Figure 8. ToggleLEDs Using WDT ISR.

7 To Learn More