**Interrupts and Low-Power Modes**

**4.1 Timer’s Continuous Mode with Interrupt**

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Written By: Shayan Taheri

// Learning programming interrupts and using low-power modes.

// Programming Timer\_A and the push buttons with interrupts.

// Engaging the low-power modes 🡪 Reducing the power consumption of the microcontroller while waiting for the interrupt events to occur.

// Timer’s Continuous Mode with Interrupt 🡪 Programming the Timer\_A in the continuous mode for generation of timing events.

// Setting up an interrupt event 🡪 We don’t have to poll the timer’s flag continuously.

// Timer Period 🡪 Raising the timer an interrupt and the hardware invokes a special piece of code to respond.

// Interrupt Basics 🡪 Reviewing the basics of interrupts in the MSP430 architecture.

// Global Interrupt Enable (GIE) 🡪 GIE Bit is the master of ON/OFF switch for all the interrupts in the chip. 🡪 Located in the Status Register (SR) 🡪 Register R2

// Interrupt Enable Bit (xIE) 🡪 Each interrupt event is enabled/disabled individually using its own interrupt enable bit.

// Interrupt Flag Bit (xIFG) 🡪 This bit indicates that the interrupt event has actually occurred. Raised by the hardware and throws an interrupt only when the xIE bit (and global bit GIE) is enabled.

// Interrupt Service Routine (ISR) 🡪 An interrupt occurs 🡪 The responding code is in a special function 🡪 ISR (rather than the main code) 🡪 Usually each interrupt event 🡪 It has a corresponding ISR function 🡪 Multiple interrupt events sometimes share the same responding ISR function 🡪 An interrupt occurs 🡪 The hardware is responsible for finding and launching the proper ISR 🡪 The ISRs are functions that are scattered around the memory.

// Vector Table 🡪 A vector is the starting address of (pointer to) an ISR. The vector table contains the addresses of all the ISRs. The vector table’s format and location are well-known to the hardware. Therefore, the hardware performs a lookup in the vector table to find the ISRs in the memory. The programmer is responsible for filing (linking) the vectors so the vector table contains the addresses of all the ISRs that are used by the code.

// Code that flashes the LEDs 🡪 Writing the code that runs Timer\_A in the continuous mode using ACLK (based on the 32 KHz crystal) and flashes the LEDs when TAR rolls back to zero. An interrupt is raised and the responding ISR toggles the LEDs. Configuring ACLK based on the 32 KHz crystal, use the function config\_ACLK\_to\_32KHz\_crystal() that we saw in an earlier lab.

// The microcontroller has multiple Timer\_A modules in the continuous mode 🡪 ACLK (based on the 32 KHz crystal) 🡪 Flashing the LEDs when TAR rolls back to zero. Every time TAR rolls back to zero 🡪 An interrupt is raised and the responding ISR toggles the LEDs. Configuring ACLK based on the 32 KHz crystal, use the function config\_ACLK\_to\_32KHz\_crystal() that we saw in an earlier lab.

// Configuring ACLK based on the 32 KHz crystal, use the function config\_ACLK\_to\_32KHz\_crystal() that we saw in an earlier lab.

// Microcontroller has multiple Timer\_A modules and we will use the timer module #0. 🡪 The A1 vector is called TIMER0\_A1\_VECTOR. This is where we will link the ISR.

// Configuring the variable TA0CTL 🡪 Containing the bit fields TASSEL (clock select), ID (clock divider), MC (mode), the interrupt flag (TAIFG), and the interrupt enable bit (TAIE).

// Code that Flashes the LEDs 🡪 Writing the code that runs Timer\_A in the continuous mode using ACLK (based on the 32 KHz crystal) and flashes the LEDs when TAR rolls back to zero. Every time TAR rolls back to zero, an interrupt is raised and the responding ISR toggles the LEDs. Configuring ACLK based on the 32 KHz crystal, use the function config\_ACLK\_to\_32KHz\_crystal() that we saw in an earlier lab.

// The microcontroller has multiple Timer\_A modules and we will use the timer module #0, called Timer0\_A. Accordingly, the AI vector is called TIMER0\_A1\_VECTOR.

// The microcontroller has multiple Timer\_A modules and we will use the timer module #0, called Timer0\_A. The A1 vector is called TIMER0\_A1\_VECTOR.

// The timer is configured using the variable TA0CTL. Containing ACLK based on the 32 KHz crystal, use the function config\_ACLK\_to\_32KHz\_crystal() that we saw in an earlier lab.

// Configuring the timer using the variable TA0CTL. Containing the bit fields TASSEL (clock select), ID (clock divider), MC (mode), the interrupt flag (TAIFG), and the interrupt.

// Complete the missing parts of the code 🡪 (A) Configuring the timer similar to how we did in earlier labs; (b) needing to enable the rollback-to-zero interrupt by setting the TAIE bit. Next, call the intrinsic function that sets the GIE bit.

// The ISR is linked to the A1 vector using the pragma line of code. The ISR is marked with the keyword ‘\_\_interrupt’ to indicate so to the compiler. The ISR is now linked to the appropriate vector. Each time an interrupt occurs, the hardware calls this function. Fill the action by toggling the LEDs and clearing the flag (TAIFG).

// The vector’s name, TIMER0\_A1\_VECTOR can be looked up in the header (.h) file included at the top of the code. To be spelled exactly as written in the header file.

// Timer’s Up Mode with Interrupt 🡪 Writing a code that runs the timer in the up mode and raises interrupts periodically.

// Rollback-to-zero (Bits: TAIE/TAIFG in TACTL) 🡪 A1 🡪 Programmer clears the flag (shared vector)

// Channel 1: CCIE/CCIFG in TACCTL1 🡪 A1 (non-shared vector)

// Channel 2: CCIE/CCIFG in TACCTL2 🡪 A1 (non-shared vector)

// Channel 0: CCIE/CCIFG in TACCTL0 🡪 A0 (non-shared vector)

// First 🡪 the interrupt event 🡪 rollback-to-zero

// Timer 🡪 Having three channels of 0, 1, and 2. 🡪 Raising an interrupt!

// Three Channels (0, 1, and 2) 🡪 Each of which can raise an interrupt. Their enable and flag bits are called CCIE/CCIFG but located in three different registers. For Timer0\_A, these are TA0CCTL0, TA0CCTL1, and TA0CCTL2.

// The channel 0 interrupt event occurs when TAR=TACCR0. Let’s recall how the up mode works by looking at the timeline below. TAR counts from 0 up to the value in TACCR0, then rolls back to zero and continues counting.

// Complete the missing parts of the code: (A) Configuring the timer similar to how we did in earlier labs. The only different here is we need to enable the rollback-to-zero interrupt by setting the TAIE bit. Next, call the intrinsic function that sets the GIE bit.

// ISR 🡪 Linking to the A1 vector using the pragma line of code. The ISR is marked with the keyword ‘\_\_interrupt’.

// The vector’s name 🡪 TIMER0\_A1\_VECTOR can be looked up in the header (.h) file included at the top of the code. It should be spelled exactly as written in the header file.

// Timer\_A continuous mode, with interrupt, flashes LEDs

#include <msp430fr6989.h>

#define redLED BIT0 // Red LED at P1.0

#define greenLED BIT7 // Green LED at P9.7

void config\_ACLK\_to\_32KHz\_crystal() {}

void main(void) {

WDTCTL = WDTPW | WDTHOLD; // Stop the Watchdog timer

PM5CTL0 &= ˜LOCKLPM5; // Enable the GPIO pins

P1DIR |= redLED; // Direct pin as output

P9DIR |= greenLED; // Direct pin as output

P1OUT &= ˜redLED; // Turn LED Off

P9OUT &= ˜greenLED; // Turn LED Off

// Configure ACLK to the 32 KHz crystal

config\_ACLK\_to\_32KHz\_crystal();

// Timer\_A configuration (fill the line below)

// Use ACLK, divide by 1, continuous mode, TAR cleared, enable

interrupt for rollback-to-zero event

TA0CTL = TASSEL\_1 | ID\_0 | MC\_2 | TACLR | TAIE;

// Ensure the flag is cleared at the start

TA0CTL &= ˜TAIFG;

// Enable the global interrupt bit (call an intrinsic function)

\_enable\_interrupts();

// Infinite loop... the code waits here between interrupts

for(;;) {}

}

//\*\*\*\*\*\*\* Writing the ISR \*\*\*\*\*\*\*

#pragma vector = TIMER0\_A1\_VECTOR // Link the ISR to the vector

\_\_interrupt void T0A1\_ISR() {

// Toggle both LEDs

P1OUT ^= redLED; //Toggles on/off the red LED

P9OUT ^= greenLED; //Toggles on/off the green LED

// Clear the TAIFG flag

TA0CTL &= ~TAIFG;

}

**4.2 Timer’s Up Mode with Interrupt**

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Written By: Shayan Taheri

// Writing a code that runs the timer in the up mode and raises interrupts periodically.

// Event: Rolling-back-to-zero / Bits: TAIE/TAIFG in TACTL / Vector: A1 / MSP430 Policy: Programmer clears the flag (shared vector).

// Event: Channel 1 / Bits: CCIE/CCIFG in TACCTL1 / MSP430 Policy: Programmer clears the flag.

// Event: Channel 2 / Bits: CCIE/CCIFG in TACCTL2 / MSP430 Policy: Programmer clears the flag.

// Event: Channel 0 / Bits: CCIE/CCIFG in TACCTL0 / MSP430 Policy: Hardware clears the flag.

// First: TAIE/TAIFG 🡪 Located in TACTL. 🡪 TA0CTL: Usage of Timer\_A module #0

// Timer Channels 🡪 0, 1, 2 🡪 Each of which can raise an interrupt.

// Enable and Flag Bits 🡪 CCIE/CCIFG 🡪 Located in three different registers.

// For Timer0\_A 🡪 TA0CCTL0, TA0CCTL1, TA0CCTL2.

// Channel 0 Interrupt 🡪 Occurrence when TAR=TACCR0.

// TAR 🡪 Counting from 0 up to the value in TACCR0. 🡪 Rolling back to zero and continuing counting.

// TAR 🡪 Raising TAIFG and raising an interrupt when enabled.

// TAR=TACCR0 🡪 CCIFG to be raised and raising an interrupt when enabled.

// Setting the Channel 0 🡪 Enabling bit CCIE 🡪 We ensure clearance of flag CCIFG.

// The global interrupt bit (GIE=1) 🡪 The bits CCIE and CCIFG 🡪 Located in the register TA0CCTL0.

// Rolling back to zero event and Channels 1 and 2. 🡪 Sharing the same vector and thus having the same ISR.

// Channel 0 🡪 Having its own non-share vector (A0).

// Having its non-share ISR.

// The hardware clears the interrupt when the ISR is called.

// Timer\_A up mode, with interrupt, flashes LEDs

#include <msp430fr6989.h>

#define redLED BIT0 // Red LED at P1.0

#define greenLED BIT7 // Green LED at P9.7

void main(void) {

WDTCTL = WDTPW | WDTHOLD; // Stop the Watchdog timer

PM5CTL0 &= ˜LOCKLPM5; // Enable the GPIO pins

P1DIR |= redLED; // Direct pin as output

P9DIR |= greenLED; // Direct pin as output

P1OUT &= ˜redLED; // Turn LED Off

P9OUT |= greenLED; // Turn LED On (alternate flashing)

// Configure ACLK to the 32 KHz crystal

config\_ACLK\_to\_32KHz\_crystal();

// Configure Channel 0 for up mode with interrupt

TA0CCR0 = 32768 // Fill to get 1 second @ 32 KHz

TA0CCTL0 |= CCIE // Enable Channel 0 CCIE bit

TA0CCTL0 &= ~CCIFG; // Clear Channel 0 CCIFG bit

// Timer\_A: ACLK, div by 1, up mode, clear TAR (leaves TAIE=0)

TA0CTL = TASSEL\_1 | ID\_0 | MC\_1 | TACLR;

// Enable the global interrupt bit (call an intrinsic function)

\_enable\_interrupts();

for(;;) {}

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#pragma vector = TIMER0\_A0\_VECTOR

\_\_interrupt void T0A0\_ISR() {

// Toggle the LEDs

P1DIR |= redLED; // Red LED pin is set as output

P9DIR |= greenLED; // Green LED is set as output

P1OUT &= ~redLED; // Red LED starts off

P9OUT &= ~greenLED; // Green LED starts off

// Hardware clears the flag (CCIFG in TA0CCTL0)

}

**4.3 Push Button with Interrupt**

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Written By: Shayan Taheri

// Reading the push buttons via interrupts 🡪 Pushing button S1, toggling the red LED, and pushing the button S2, toggling the green LED.

// Wiring the push buttons of the LaunchPad in the active low configuration (they read 0 when pushed).

// Pushing the button when there is a falling edge that is followed by a rising edge.

// Raising an interrupt for the microcontroller on either edge.

// Setting up the falling edge interrupt 🡪 Occurrence of the interrupt as soon as the button is pushed.

// Port 1 🡪 Having eight pins 🡪 Each of which can raise an interrupt individually.

// Each Port 1’s eight pins 🡪 Enabling its built-in register (P1REN is 8-bit) 🡪 Enabling its interrupt (P1IE is 8-bit) and has its own interrupt flag (P1IFG is 8-bit).

// Complete the code by writing 0 or 1 in each configuration variable 🡪 Enabling the global interrupts bit (GIE bit) and write an empty infinite for-loop at the end of main function 🡪 The code waits there between interrupts.

// Remaining task 🡪 Writing the ISR and linking it to its vector in the vector table.

//Port 1 Vector Name 🡪 P1\_VECTOR or PORT\_1\_VECTOR 🡪 Defining the vector names in the header file (msp430fr6989.h).

// Try locating the file in the folder path shown below.

// Port 1 ISR 🡪 Calling if either of the buttons is pushed.

// Inspecting P1IFG 🡪 Containing eight flags.

// [First]: Performing an if-statement to check if button 1 was pushed (bit BUT1 in PP1IFG is true).

// Port 1 ISR 🡪 Sharing among multiple interrupt pins 🡪 Responsibility of the programmer for clearing the flag (which cannot be done by the Hardware).

// Using if-statement for checking for the second button 🡪 If pushed, it toggles the green LED and clears the flag.

// Writing the two if-statements since both buttons may need to be processed simultaneously.

// Port 1 🡪 Eight pins that each can raise an interrupt individually.

// Each Port 1 🡪 Eight pins can enable its built-in resistor (P1REN is 8-bit) that can enable its interrupt (P1IE is 8-bit) and has its own interrupt flag (P1IFG is 8-bit).

// Complete the code by writing 0 or 1 in each configuration variable according to the comments in the code. Then, enable the global interrupts bit (GIE bit) and write an empty infinite for-loop at the end of main function so the code waits there between interrupts.

// Remaining task 🡪 Write the ISR and link it to its vector in the vector table.

// All the eight bits in Port 1 share the same vector and, therefore, have the same ISR. Accordingly, the two push buttons, which are both mapped to Port 1, are serviced by the same ISR.

// What is the name of the Port 1 vector? We suspect it’s something like P1\_VECTOR or PORT\_1\_VECTOR.

// The vector names are defined in the header file, in our case, it is the file msp430fr6989.h.

// The Port 1 ISR 🡪 To be called if either of the buttons is pushed.

// Inspecting P1IFG that contains eight flags.

// [First]: Performing an if-statement. 🡪 Toggling the red LED and the bit BUT1 in P1IFG is cleared.

// A similar if-statement checks for the second button and, if pushed, toggles the green LED and clears the flag.

// Writing two if-statements (rather than if-else) since both buttons may need to be processed simultaneously.

#define BUT1 BIT1 // Button S1 at Port 1.1

#define BUT2 BIT2 // Button S2 at Port 1.2

...

// Configuring buttons with interrupt

P1DIR &= ˜(BUT1|BUT2); // 0: input

P1REN |= (BUT1|BUT2); // 1: enable built-in resistors

P1OUT |= (BUT1|BUT2); // 1: built-in resistor is pulled up to Vcc

P1IE |= (BUT1|BUT2); // 1: enable interrupts

P1IES |= (BUT1|BUT2); // 1: interrupt on falling edge

P1IFG &= ~(BUT1|BUT2); // 0: clear the interrupt flags

#pragma vector = ... // Write the vector name

\_\_interrupt void Port1\_ISR() {

// Detect button 1 (BUT1 in P1IFG is 1)

if ( ... ) {

// Toggle the red LED

P1OUT ^= redLED;

// Clear BUT1 in P1IFG

P1IFG &= ~BUT1;

}

// Detect button 2 (BUT2 in P1IFG is 1)

if ( ... ) {

// Toggle the green LED

P9OUT ^= greenLED;

// Clear BUT2 in P1IFG

P1IFG &= ~BUT2;

}

}

**4.4 Low-Power Modes**

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Written By: Shayan Taheri

// Support of multiple low-power modes in the MSP430 microcontroller.

// Shutting down more and more components that are not needed 🡪 Saving more power!

// Active Mode 🡪 Making all the clocks ON!

// MCLK (Master Clock) 🡪 Driving the CPU and SMCLK (sub-master clock) and ACLK (auxiliary clock) for driving the peripherals.

// Engaging LPM0 🡪 Shutting down MCLK that means the CPU is off!

// Making it possible for peripherals to continue running based on SMCLK and ACLK.

// LPM3 🡪 Shutting down SMCLK and peripherals can run using ACLK.

// Engaging LPM4 disables all the clocks!

// Using LPM4 and the timer requests ACLK. 🡪 ACLK turning ON and staying as long as the timer is using it. This causes prevention of unintentional shutdown of peripherals due to a misconfiguration.

// Interrupts and low-power modes 🡪 Engaging a low-power mode and shutting down the CPU and possibly other clock signals.

// The only way to reactivate the CPU is via an interruption.

// Engaging a low-power mode and shutting down and possibly other clock signals do as well.

// The only way to reactivate the CPU is via an interrupt.

// Engaging a low-power mode, and enabling the interrupts and configuring at least one interrupt event.

// A typical flow of the program 🡪 Configuring an interrupt event and to enter a low-power mode.

// Occurrence of an interrupt 🡪 Engaging the MCU by the active mode automatically (activating the CPU and all the clocks).

// Engaging the low-power modes via four bits in the Status Register (SR) called SCG1, SCG2, CPUOFF, and OSCOFF.

// Dealing with these bits directly can lead to usage of intrinsic function below as well as enabling the global interrupts (GIE bit) because interrupts.

// Using this function when there is no need to call the intrinsic function that enables interrupts. 🡪 \_low\_power\_mode\_x();

// **Engaging Low-Power Modes** 🡪 Revisiting the three codes of this experiment and engaging the appropriate low-power mode for each code.

// Goal: Minimization of the power consumption and choosing the lowest consuming power mode that keeps the code operational.

// Multiple low-power modes 🡪 Supported in the MSP430 microcontroller.

// Shutting down more and more components that are not needed, more power can be saved.

// Active Mode: All the clocks are ON. MCLK (Master Clock) drives the CPU and SMCLK (SubMaster Clock) and ACLK (Auxiliary Clock) drive peripherals.

// Engaging LPM0 shuts down MCLK that means the CPU is OFF.

// There is possibility for peripherals to continue running based on SMCLK (SubMaster) and ACLK (Auxiliary Clock) drive peripherals.

// Engaging LPM0 that can shut down MCLK that means the CPU is OFF.

// There is a possibility that other clock signals can do well as well.

// The only way to reactivate the CPU is through using an interruption.

// A low power mode can be engaged for enabling interrupts and configure at least one interrupt event.

// The typical flow of the program is configuring an interrupt event and to enter a low power mode. Occurrence of an interruption can lead to engaging the active mode with the MCU automatically (activating the CPU and all the clocks) 🡪 This leads to launching the corresponding ISR, and finally returning to the same low-power mode that was engaged earlier.

// A typical program flow 🡪 Configuring an interrupt event and entering low power mode.

// Interrupt Occurrence 🡪 Engaging MCU in the active mode automatically (activating the CPU and all the clocks). 🡪 Launching the corresponding ISR and returning to the same low power mode that was engaged earlier.

// Low Power Modes 🡪 Engaging via four bits in the Status Register (SR) called SCG1, SCG2, CPUOFF, and OSCOFF.

// Instead of dealing with these bits directly, we use the intrinsic function below to engage a low-power mode. The function is able to enable the global interrupts (GIE bit) because interrupts are always used when a low-power mode is engaged. Using this function can lead to calling the intrinsic function that can enable interrupts. 🡪 \_low\_power\_mode\_x();

// Interrupts and low-power modes go hand-in-hand 🡪 Engaging a low-power mode and shutting down the CPU.

// Reactivating the CPU can be done via an interrupt.

// Engaging a low-power mode and enabling the interrupts and configuring at least one interrupt event.

// The typical program flow 🡪 Configuring an interrupt event and entering a low-power mode.

// Occurrence of an interrupt 🡪 Three channels (0, 1, 2) according to which each can raise an interrupt. Their enable and flag bits are called CCIE/CCIFG but located in three different registers. 🡪 Timer: Timer0\_A 🡪 TA0CCTL0, TA0CCTL1, and TA0CCTL2.

// The Channel 0 interrupt event 🡪 Occurrence when TAR=TACCR0.

// The up mode works by looking at the timeline below.

// TAR counts 🡪 From 0 up to the value in TACCR0 🡪 Rolling back to zero and continue counting.

// TAR rolls back to zero 🡪 Knowing that TAIFG and this raises an interrupt when enabled.

// The channel 0 interrupt event 🡪 Occurrence when TAR=TACCR0.

// Recalling how the up mode works by looking at the timeline below.

// TAR counts from 0 up to the value in TACCR0, then rolling back to zero and continue counting.

// TAR rolling back to zero 🡪 TAIFG is raised and this raises an interrupt when enabled.

// TAR=TACCR0 🡪 Raising CCIFG (as marked in the timeline) and raising an interrupt when enabled.

// Raising the flag CCIFG and raising an interrupt when enabled. 🡪 Using the channel 0 interrupt event (CCIFG event).

// TAR rolls back to zero 🡪 TAIFG is raised and this raises an interrupt when enabled.

// Using the Channel 0 interrupt event (CCIFG event).

// Set the Channel 0 enable bit (CCIE=1) 🡪 Initial clearing of the flag

// Enabling the global interrupt bit (GIE=1)

// The bits CCIE and CCIFG located in the register TA0CCTL0

// Rolling back to zero event and Channels 1 and 2.

// Sharing the same vector and having the same ISR.

// The program is responsible for clearing the interrupt flag

// Channel 0 🡪 Having its own non-shared vector!

// Having non-shared ISR

// The hardware clears the interrupt when the ISR is called

// Revisiting the three codes of this experiment and engage the appropriate the low-power mode for each code.

// Revisiting the three codes of this experiment and engaging the appropriate low-power mode for each code. The goal is minimization of the consumed power 🡪 Choosing the lowest consuming power mode that keeps the code operational.

// Doing these changes 🡪 Having the CPU cycle in the infinite for-loop between interrupts 🡪 The CPU is shut down now while we are waiting for the interrupt events.

**(A) How to Handle Part A?**

#include <msp430fr6989.h>

#define redLED BIT0 // Red LEDs pin location is at P1.0

#define greenLED BIT7 // Green LEDs pin location is at P9.7

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Reconfigures ACLK to be rerouted to the 32 KHz crystal on the LaunchPad

void config\_ACLK\_to\_32KHz\_crystal() {

// The default mode of the ACLK is a built-in oscillator at a frequency of 39KHz normally.

// Rerouted the pins to LFXIN/LFXOUT functionality so that the ACLK can be routed to the 32KHz crystal.

// This information can be found using the LaunchPad user's guide (page 29) and the chip's data sheet (page 123).

PJSEL1 &= ~BIT4;

PJSEL0 |= BIT4;

// We need to for the crystal to settle, once it has started.

// Therefore, we will wait until the local and global oscillator fault flags are cleared and remain cleared.

CSCTL0 = CSKEY; // Unlock CS registers, to divert the pins for the crystal functionality.

//Clears the flag and will do so until they remain cleared.

do {

CSCTL5 &= ~LFXTOFFG; // Local oscillator fault flag

SFRIFG1 &= ~OFIFG; // Global oscillator fault flag

} while((CSCTL5 & LFXTOFFG) != 0);

CSCTL0\_H = 0; // Lock CS registers, returns the pins.

return;

}

void main(void) {

WDTCTL = WDTPW | WDTHOLD; // Stops the watchdog timer. We do this so the MCU doesn't reset itself periodically.

PM5CTL0 &= ~LOCKLPM5; // We enable the general purpose I/O pins.

// Configuration and initialization of LEDs

P1DIR |= redLED; // Red LED pin is set as output

P9DIR |= greenLED; // Green LED is set as output

P1OUT &= ~redLED; // Red LED starts off

P9OUT &= ~greenLED; // Green LED starts off

// We configure the Auxiliary Clock to reroute to the 32KHz crystal.

config\_ACLK\_to\_32KHz\_crystal();

// Configuring Timer\_A0 by setting ACLK, frequency division by 1, continuous mode, clear TAR, and interrupt for rollback-to-zero event

TA0CTL = TASSEL\_1 | ID\_0 | MC\_2 | TACLR | TAIE;

// Ensure the flag is cleared at the start

TA0CTL &= ~TAIFG;

// Intrinsic function that starts the low power mode 3 along with enabling global interrupt enable

\_low\_power\_mode\_3();

}

//\*\*\*\*\*\*\* ISR Code \*\*\*\*\*\*\*

#pragma vector = TIMER0\_A1\_VECTOR // Links the vector with the ISR

\_\_interrupt void T0A1\_ISR() {

P1OUT ^= redLED; //Toggles on/off the red LED

P9OUT ^= greenLED; //Toggles on/off the green LED

TA0CTL &= ~TAIFG; // Clears the Timer\_A interrupt flag

}

**(B) How to Handle Part B?**

#include <msp430fr6989.h>

#define redLED BIT0 // Red LEDs pin location is at P1.0

#define greenLED BIT7 // Green LEDs pin location is at P9.7

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Reconfigures ACLK to be rerouted to the 32 KHz crystal on the LaunchPad

void config\_ACLK\_to\_32KHz\_crystal() {

// The default mode of the ACLK is a built-in oscillator at a frequency of 39KHz normally.

// Rerouted the pins to LFXIN/LFXOUT functionality so that the ACLK can be routed to the 32KHz crystal.

// This information can be found using the LaunchPad user's guide (page 29) and the chip's data sheet (page 123).

PJSEL1 &= ~BIT4;

PJSEL0 |= BIT4;

// We need to for the crystal to settle, once it has started.

// Therefore, we will wait until the local and global oscillator fault flags are cleared and remain cleared.

CSCTL0 = CSKEY; // Unlock CS registers, to divert the pins for the crystal functionality.

//Clears the flag and will do so until they remain cleared.

do {

CSCTL5 &= ~LFXTOFFG; // Local oscillator fault flag

SFRIFG1 &= ~OFIFG; // Global oscillator fault flag

} while((CSCTL5 & LFXTOFFG) != 0);

CSCTL0\_H = 0; // Lock CS registers, returns the pins.

return;

}

void main(void) {

WDTCTL = WDTPW | WDTHOLD; // Stops the watchdog timer. We do this so the MCU doesn't reset itself periodically.

PM5CTL0 &= ~LOCKLPM5; // We enable the general purpose I/O pins.

// Configuration and initialization of LEDs

P1DIR |= redLED; // Red LED pin is set as output

P9DIR |= greenLED; // Green LED is set as output

P1OUT &= ~redLED; // Red LED starts off

P9OUT &= ~greenLED; // Green LED starts off

// We configure the Auxiliary Clock to reroute to the 32KHz crystal.

config\_ACLK\_to\_32KHz\_crystal();

// We set the value to 32768 which is a 1 second delay since 32768/32768 = 1.

TA0CCR0 = 32768;

// 16,384 for 0.5 secs

// 3,227 for 0.1 secs

TA0CCTL0 |= CCIE; // Enable Channel 0 interrupt

TA0CCTL0 &= ~CCIFG; // Clear Channel 0 interrupt flag

// Configuring Timer\_A0 by setting ACLK, frequency division by 1, up mode, and clear TAR.

TA0CTL = TASSEL\_1 | ID\_0 | MC\_1 | TACLR;

// Intrinsic function that starts the low power mode 3 along with enabling global interrupt enable

\_low\_power\_mode\_3();

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#pragma vector = TIMER0\_A0\_VECTOR

\_\_interrupt void T0A0\_ISR() {

P1OUT ^= redLED; //Toggles on/off the red LED

P9OUT ^= greenLED; //Toggles on/off the green LED

// The flag is cleared by the hardware.

}

**(C) How to Handle Part C?**

#include <msp430fr6989.h>

#define redLED BIT0 // Red LEDs pin location is at P1.0

#define greenLED BIT7 // Green LEDs pin location is at P9.7

#define BUT1 BIT1 // Button S1 at Port 1.1

#define BUT2 BIT2 // Button S2 at Port 1.2

void main(void) {

WDTCTL = WDTPW | WDTHOLD; // Stops the watchdog timer. We do this so the MCU doesn't reset itself periodically.

PM5CTL0 &= ~LOCKLPM5; // We enable the general purpose I/O pins.

// Configuration and initialization of LEDs

P1DIR |= redLED; // Red LED pin is set as output

P9DIR |= greenLED; // Green LED is set as output

P1OUT &= ~redLED; // Red LED starts off

P9OUT &= ~greenLED; // Green LED starts off

// Configuring buttons with interrupt

P1DIR &= ~(BUT1|BUT2); // Sets the push buttons S1 and S2 as input

P1REN |= (BUT1|BUT2); // Enables the internal resistors of both buttons

P1OUT |= (BUT1|BUT2); // Sets the resistors as pull up to VCC

P1IE |= (BUT1|BUT2); //Enables the interrupt events of both buttons

P1IES |= (BUT1|BUT2); // Configures the interrupt as falling edge

P1IFG &= ~(BUT1|BUT2); // Clears both interrupt flags.

// Intrinsic function that starts the low power mode 4 along with enabling global interrupt enable

\_low\_power\_mode\_4();

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#pragma vector = PORT1\_VECTOR

\_\_interrupt void Port1\_ISR() {

if((P1IFG & BUT1) == BUT1){

P1OUT ^= redLED; //Toggles on/off the red LED

// Clear the flag raised by S1 push button

P1IFG &= ~BUT1;

}

if((P1IFG & BUT2) == BUT2){

P9OUT ^= greenLED;//Toggles on/off the green LED

// Clear the flag raised by S2 push button

P1IFG &= ~BUT2;

}

}