

Workshop 2: Interrupts, Timers, UART and ADC

Getting Started

It is strongly recommended that you work through all tasks in the first instance and ignore the exercises along the way. You can then return to the exercises listed at the end of each task once you have managed to complete a first run of all tasks successfully.

You will need an MSP432P401R Launchpad (henceforth referred to as MSP432) to complete the tasks in this workshop. You will also need a computer that has PlatformIO installed.

Contents

1 Task 1 – Clock Configuration	2
1.1 'For Loop' Delays and Clocks	2
1.1.1 EXERCISE 9	4
1.1.2 EXERCISE 10	4
2 Task 2 – Interrupts	5
2.1 Configure the interrupt	5
2.2 Enable the interrupt	5
2.3 Writing the interrupt function (ISR – Interrupt Service Routine)	7
2.4 EXERCISE 1	8
2.5 EXERCISE 2	8
3 Task 3 - Timers	9
3.1 Task 3a - Timer Basics	9
3.1.1 Configuring the timer's clock	9
3.1.2 Configuring interrupts	10
3.1.3 Starting the timer	10
3.1.4 EXERCISE 4	11
3.1.5 EXERCISE 5	11
3.2 Task 3b – Improving Timer Resolution	12
3.2.1 EXERCISE 6	12
4 Task 4 - UART	13
4.1 Configure the UART module	14
4.2 Enable interrupts	15
4.3 Write the interrupt function (ISR – Interrupt Service Routine)	15
4.4 EXERCISE 7	17
5 Task 5 - ADC	18
5.1 EXERCISE 8	20
5.2 EXERCISE 9	20
5.3 EXERCISE 10	20

1 Task 1 – Clock Configuration

In the next task, we shall demonstrate how we can change the clock settings for this board.

1.1 'For Loop' Delays and Clocks

Learning outcome: Understand how a for loop creates a delay of specific/desired duration.

In the previous workshop, when making the LED blink we used a delay based on a for loop. But how exactly does it generate this delay and how can we determine the duration of this delay?

Each for loop takes a couple of clock cycles. This is because it must decrement the variable used for counting, as well as compare the value against the condition of the loop. When we repeatedly iterate through a for loop, for example, by making the for loop count down from 10000 down to 1, as shown below, we generate a significant delay.

```
for(i=10000; i>0; i--);
```

It should be evident that increasing the number of iterations in the for loop will increase the delay.

This loop can also help to demonstrate the concept of clocks. Rather than changing the number of iterations we could instead slow down the microcontroller's clock speed, thereby causing each iteration to take longer. Hence, for a fixed number of iterations we can adjust the length of the for loop delay by varying the clock speed (or to be more precise, the clock frequency). Once again, it is worth mentioning that this is not the ideal way of implementing a delay, however, it does help us understand the nature of the clock system on a microcontroller device like the MSP432.

There are a number of clocks available within the MSP432, however, here we focus on the Digitally Controlled Oscillator (DCO) which is the default clock after startup.

In order to change the clock's settings, we must modify the contents of the Clock Select (CS) register. This is a large register with multiple elements. In order to change the frequency of the DCO we need to change the bits corresponding to the CTL0 settings. Fortunately, the settings are already defined in the chip's header file (if you look under the section called "CS Bits"). You can find this definition by typing CS_CTL0_DCORSEL_0 in main.c, then right-clicking on word and selecting "Go to Definition".

```
2584 #define CS_CTL0_DCORSEL_0 ((uint32_t)0x00000000) /*!< Nominal DCO Frequency Range (MHz): 1 to 2 */
2585 #define CS_CTL0_DCORSEL_1 ((uint32_t)0x00010000) /*!< Nominal DCO Frequency Range (MHz): 2 to 4 */
2586 #define CS_CTL0_DCORSEL_2 ((uint32_t)0x00020000) /*!< Nominal DCO Frequency Range (MHz): 4 to 8 */
2587 #define CS_CTL0_DCORSEL_3 ((uint32_t)0x00030000) /*!< Nominal DCO Frequency Range (MHz): 8 to 16 */
2588 #define CS_CTL0_DCORSEL_4 ((uint32_t)0x00040000) /*!< Nominal DCO Frequency Range (MHz): 16 to 32 */
2589 #define CS_CTL0_DCORSEL_5 ((uint32_t)0x00050000) /*!< Nominal DCO Frequency Range (MHz): 32 to 64 */
2590 /* CS_CTL0[DCORES] Bits */
```

To understand the frequency options available through the above settings, we need to refer to the MSP432's user guide:

18-16	DCORSEL	RW	1h	<p>DCO frequency range select. Selects frequency range settings for the DCO.</p> <p>000b = Nominal DCO Frequency (MHz): 1.5; Nominal DCO Frequency Range (MHz): 1 to 2</p> <p>001b = Nominal DCO Frequency (MHz): 3; Nominal DCO Frequency Range (MHz): 2 to 4</p> <p>010b = Nominal DCO Frequency (MHz): 6; Nominal DCO Frequency Range (MHz): 4 to 8</p> <p>011b = Nominal DCO Frequency (MHz): 12; Nominal DCO Frequency Range (MHz): 8 to 16</p> <p>100b = Nominal DCO Frequency (MHz): 24; Nominal DCO Frequency Range (MHz): 16 to 32</p> <p>101b = Nominal DCO Frequency (MHz): 48; Nominal DCO Frequency Range (MHz): 32 to 64</p> <p>110b to 111b = Nominal DCO Frequency (MHz): Reserved--defaults to 1.5 when selected; Nominal DCO Frequency Range (MHz): Reserved--defaults to 1 to 2 when selected.</p>
-------	---------	----	----	---

Three bits are available for configuring the DCORSEL register, thus allowing for eight different options. Here, we can see that six options (0 to 5) have been defined, which allow us to set the DCO nominal frequency between 1.5 MHz and 48 MHz.

WARNING: Do not set the DCORSEL to a value that would drive the frequency higher than **24MHz** since this will “lock” your MSP432 board and will render it unusable. To enable this feature, that is, driving the clock higher than 24MHz, a number of other settings need to be configured which are beyond the scope of this workshop.

We can choose a particular DCO clock frequency by setting the CTL0 register to the corresponding bits. Note we are using a different syntax to access the registers in this case, specifically the CMSIS syntax. You could access the register in the typical way, however, CSCTL0 has not been defined in the header file.

```
CS->CTL0 = CS_CTL0_DCORSET_0;
```

If you uploaded this and this alone you would not see a change in clock frequency. This is because the clock registers are write protected - you cannot write to them without unlocking a password protection. The key needed to unlock the clock registers is again defined in the headers as CS_KEY_VAL. In the code below we unlock the clock registers, change the clock frequency and then lock the clock registers back up. It is crucial to lock the registers again as it will protect them from accidental changes.

```
CS->KEY = CS_KEY_VAL; //Unlocks register for write access
CS->CTL0 = CS_CTL0_DCORSEL_0;
CS->KEY = 0;
```

We are now able to write a Blink LED program that changes the clock frequency.

- Create a new project and name it "Ex8-forloopdelay"
- Write a program based on the contents shown in the following screenshot

```
1  #include <msp.h>
2
3  void main(void)
4  {
5      WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;    // stop watchdog timer
6
7      //Set P1.0 as an OUTPUT
8      P1DIR |= (1<<0);
9      P1OUT |= (1<<0);
10
11     CS->KEY = CS_KEY_VAL; //Unlocks register for write access
12     CS->CTL0 = CS_CTL0_DCORSEL_0;
13     CS->KEY = 0;
14
15     int i;
16     while(1)
17     {
18         P1OUT ^= 1<<0;    // toggle P1.0
19         for(i=10000; i>0; i--);    // delay
20     }
21
22     return 0;
23 }
```

1.1.1 EXERCISE 9

Experiment with different clock frequencies and observe how the LED blinking pattern changes accordingly.

WARNING: Do not select a clock frequency higher than 24 MHz since this will “lock” your MSP432 board and will render it unusable.

1.1.2 EXERCISE 10

Can you create a program that toggles one of the LEDs on or off with an accurate 1 second delay?

2 Task 2 – Interrupts

In this task, you will learn how to use interrupts on the MSP432.

Learning outcome: Learn how to toggle an LED using interrupts on Port 1.

NOTE: The terms ‘switch’ and ‘pushbutton’ are used interchangeably in this workshop. They mean the same thing, that is, the physical push buttons available on the MSP432.

We begin by first halting the watchdog timer as per usual. Then we set the LED connected to P1.0 as an output (LED1), and switch 1 on P1.1 as an input. This has been covered before so is not detailed here. The complete code for this functionality is shown below.

Now we need to set up the interrupts. To use interrupts, there are three things that we need to do:

1. Configure the interrupt.
2. Enable the interrupt.
3. Write the interrupt function.

2.1 Configure the interrupt

Depending on the purpose of the interrupt under consideration, there will be different settings and registers to configure. For GPIOs, the only setting required is whether to trigger on a high to low transition (falling edge) or a low to high transition (rising edge). This is defined by the *interrupt select* register (PxIES). As always, each bit corresponds to a particular pin within port X (here being port 1). In this example, we want to trigger on a high to low transition for switch 1 on P1.1.

```
P1IES |= 0x02; // Choose high to low transition to trigger interrupt
```

Bit	Field	Type	Reset	Description
7-0	PxIES	RW	Undefined	Port X interrupt edge select 0b = PxIFG flag is set with a low-to-high transition. 1b = PxIFG flag is set with a high-to-low transition.

2.2 Enable the interrupt

Interrupts must be enabled for the CPU to respond to them. Generally they must be enabled at two places. First, at the peripheral level. For example, the GPIOs have an *interrupt enable* register (P1IE) which is used to enable/disable interrupts for each pin in a port. Setting a bit enables the interrupt while clearing a bit disables it.

Here we will set and thus enable the interrupt for P1.1.

```
P1IE |= 0x02;
```

Bit	Field	Type	Reset	Description
7-0	PxIE	RW	0h	Port X interrupt enable 0b = Corresponding port interrupt disabled 1b = Corresponding port interrupt enabled

The second place where interrupts need to be configured is in the nested interrupt vector controller (NVIC) module. The NVIC module requires us to enable interrupts related to our specific port.

To do this we modify the NVIC_ISER (ISER: Interrupt Set-Enable Register). This spreads over 64 bits with each bit referring to a particular interrupt. So how do we know which bit to set for port 1? We need to know the interrupt number of the port 1 interrupts. Once we identify this interrupt number, this then maps directly to the number of the bits to set in the ISER register.

This information is found in the datasheet (Table 6-39). Shown below is an extract of this table illustrating that the port 1 interrupt is number 35.

INTISR[35]	I/O Port P1	P1IFG.x (x = 0 to 7)
INTISR[36]	I/O Port P2	P2IFG.x (x = 0 to 7)
INTISR[37]	I/O Port P3	P3IFG.x (x = 0 to 7)
INTISR[38]	I/O Port P4	P4IFG.x (x = 0 to 7)
INTISR[39]	I/O Port P5	P5IFG.x (x = 0 to 7)
INTISR[40]	I/O Port P6	P6IFG.x (x = 0 to 7)

The NVIC_ISER consists of 64 bits split into two groups of 32 bits (see TRM for details).

Table 2-25. NVIC Registers

Offset	Acronym	Register Name	Type	Reset	Section
100h	ISER0	Irq 0 to 31 Set Enable Register	read-write	00000000h	Section 2.4.3.1
104h	ISER1	Irq 32 to 63 Set Enable Register	read-write	00000000h	Section 2.4.3.2

Number 35 will correspond to bit position three (remember that the LSB is bit position zero) in the second half of NVIC_ISER. We set this bit with the following command.

```
NVIC->ISER[1] |= 1 << (3);
```

We are almost finished. At this point, the code should look like the one shown in the screenshot below. We have included an infinite while loop so the device performs no operation while waiting for the interrupt. There are better ways to do this by making use of the low power modes (LPM) available on the MSP432.

```
#include "msp.h"

void main(void)
{
    WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;    // stop watchdog timer

    //Choose GPIO functionality on desired pins
    P1SEL0 &= ~0x03;
    P1SEL1 &= ~0x03;

    P1DIR |= 0x01; //LED as output
    P1OUT &= ~0x01; //Turn off

    P1DIR &= ~0x02; //Switch 1 as input
    P1REN |= 0x02; //Enable pullup/down resistor
    P1OUT |= 0x02; //Set to pullup resistor

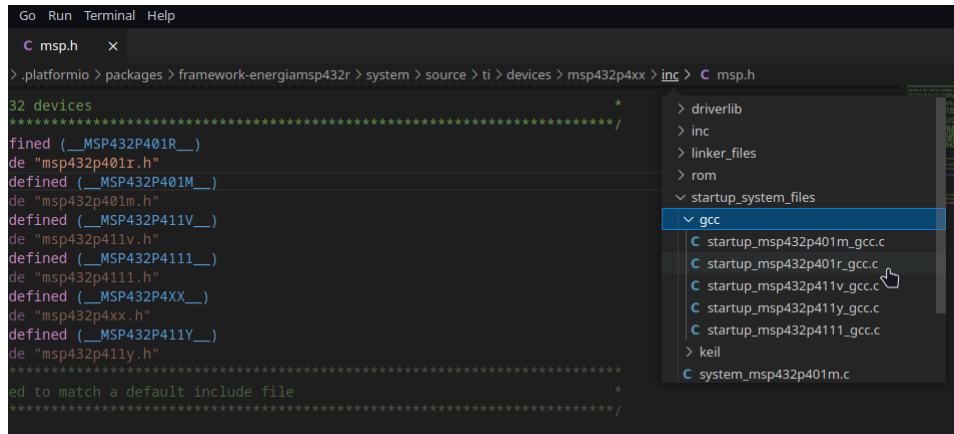
    P1IES |= 0x02; //Choose high to low transition to trigger interrupt
    P1IE |= 0x02;

    NVIC->ISER[1] |= 1 << (3);

    while(1){
        ;
    }
}
```

2.3 Writing the interrupt function (ISR – Interrupt Service Routine)

The third and final task is to write the function that will be called when the interrupt occurs, that is, the interrupt service routine (ISR). We first need to find the correct name to call our function. This has already been defined in the startup file called 'startup_msp432p401r_gcc.c'. You can find this file by going by opening msp.h file (right-click it in main.c and select "Go to Definition"), and navigating to inc > startup_system_files > gcc > startup_msp432p401r_gcc.c with the top navigation bar:



Browsing through the startup file, we find function prototypes for a number of interrupts. Our function must be called: PORT1_IRQHANDLER.

```

100 extern void PORT1_IRQHandler (void) __attribute__((weak,alias("Default_Handler")));
101 extern void PORT2_IRQHandler (void) __attribute__((weak,alias("Default_Handler")));
102 extern void PORT3_IRQHandler (void) __attribute__((weak,alias("Default_Handler")));
103 extern void PORT4_IRQHandler (void) __attribute__((weak,alias("Default_Handler")));
104 extern void PORT5_IRQHandler (void) __attribute__((weak,alias("Default_Handler")));
105 extern void PORT6_IRQHandler (void) __attribute__((weak,alias("Default_Handler")));

```

We should now add our interrupt function into main.c:

```

void PORT1_IRQHandler(void)
{
}

```

When an interrupt is triggered on a pin, it sets the corresponding bit in the *interrupt flag* register. This must be reset when entering the interrupt to avoid the interrupt being called again. The flag register is also useful in that it indicates the pin that caused the interrupt. All port 1 pins will trigger the same interrupt function, hence sometimes you will need to check which pin caused the interrupt.

In this simple example, we will simply clear the interrupt flag of switch 1 and toggle LED1.

NOTE: The following may be unstable because the switches have not been debounced.

```

// Interrupt handler for Port 1
void PORT1_IRQHandler(void)
{
    P1IFG &= ~0x02; // Clear interrupt flag
    P1OUT ^= 0x01; // Toggle LED
}

```

Upload this code to your MSP432 and press switch 1. If your code is correct, the LED should toggle each time you press the switch. The complete code is shown below.

```
#include "msp.h"

void main(void)
{
    WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;    // stop watchdog timer

    //Choose GPIO functionality on desired pins
    P1SEL0 &= ~0x03;
    P1SEL1 &= ~0x03;

    P1DIR |= 0x01; //LED as output
    P1OUT &= ~0x01; //Turn off

    P1DIR &= ~0x02; //Switch 1 as input
    P1REN |= 0x02; //Enable pullup/down resistor
    P1OUT |= 0x02; //Set to pullup resistor

    P1IES |= 0x02; //Choose high to low transition to trigger interrupt
    P1IE |= 0x02;

    NVIC->ISER[1] |= 1 << (3);

    while(1){
        ;
    }
}

void PORT1_IRQHandler(void) // Interrupt handler for port 1
{
    P1IFG &= ~0x02; //Clear interrupt flag
    P1OUT ^= 0x01; //Toggle LED
}
```

2.4 EXERCISE 1

Modify the code to use switch 2 instead of switch 1.

2.5 EXERCISE 2

Change the code so switch 1 is used to turn LED1 on, while switch 2 is used to turn the LED1 off.

3 Task 3 - Timers

The easy way we have used thus far to generate delays is by using 'for' loops. This, however, is an inefficient and a poor method for implementing delays. In this workshop, we will make use of the timers that are built-in the MSP432 already.

There are three types of a timer available on the MSP432; the watchdog timer (WDT), Timer 32 and Timer A. This task will focus on using Timer A. We will be making use of interrupts, so make sure you are comfortable with Task 1 before completing this task.

3.1 Task 3a - Timer Basics

Learning outcome: Learn how to blink an LED using Timer A on the MSP432.

Timer A is one type of timer on the MSP432. All types of timers can be used in the same manner. You simply need to change the name of the register. For example, the *timer A control* register has the generic name TAxCTL. If we want to use a specific timer we replace the 'x' with a number identifying it. In this task, we will use Timer 0 (of the Timer A module) throughout. This means our control register is called TA0CTL.

We will make an LED blink using Timer A. Start by halting the watchdog timer and setting the LED on P1.0 as an output.

```
void main(void)
{
    WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;    // stop watchdog timer

    // GPIO functionality is selected on desired pins
    P1SEL0 &= ~0x01;
    P1SEL1 &= ~0x01;

    P1DIR |= 0x01;
    P1OUT &= ~0x01;
```

To use a timer, there are five things we need to do:

1. Choose the clock source.
2. Choose the clock divider.
3. Select the timer mode.
4. Enable the appropriate interrupts.
5. Write the interrupt function.

3.1.1 Configuring the timer's clock

The settings for timer A are (almost) all stored within the TAxCTL register. Looking at the register in the technical manual (section 19.3.1) we see bits 8 and 9 (TASSEL) are used to determine the clock used by the timer. Which timer we want to use depends on the length of timing we want to implement. For long pauses we tend to use a slow clock, for short pauses, we use a fast clock.

Four choices of clocks are available, as shown in the table below. TAxCLK and INCLK are both driven by an external signal applied to the corresponding pin, while ACLK and SMCLK are internally driven. Let's focus on ACLK and SMCLK. Both these clocks can be configured but we shall use their default values. By default, ACLK is 32.768 kHz and SMCLK is 3 MHz. If we want to make an LED blink, we will use the slower clock.

9-8	TASSEL	RW	0h	Timer_A clock source select 00b = TAxCLK 01b = ACLK 10b = SMCLK 11b = INCLK
-----	--------	----	----	---

Set ACLK as the clock source (32.768kHz) by setting TASSEL = 01b.

```
TA0CTL |= (1 << 8); // Select ACLK as clock source
TA0CTL &= ~(1 << 9);
```

3.1.2 Configuring interrupts

Before we start the timer, we need to setup the interrupts. As described in Task 1, the timer interrupts have to be enabled in two places. First, we enable the interrupt at a local level, here only enabling one of the Timer A timers. This is done using the interrupt enable bit of the TA0CTL register (TAIE = 1).

```
TA0CTL |= (1 << 1); // Enable interrupt
```

Then we need to enable at a global level by enabling Timer interrupts in general. This is done using the NVIC registers as detailed in Task 1. We need to know the interrupt number that corresponds to the Timer A interrupts. This is found in the MSP432's datasheet in Table 6-39. The timer flag is the TAIFG bit in the TA0CTL register. We can see below that this corresponds to interrupt 9, INTISR[9].

INTISR[8]	Timer_A0	TA0CCTL0.CCIFG
INTISR[9]	Timer_A0	TA0CCTLx.CCIFG (x = 1 to 4), TA0CTL.TAIFG
INTISR[10]	Timer_A1	TA1CCTL0.CCIFG
INTISR[11]	Timer_A1	TA1CCTLx.CCIFG (x = 1 to 4), TA1CTL.TAIFG
INTISR[12]	Timer_A2	TA2CCTL0.CCIFG
INTISR[13]	Timer_A2	TA2CCTLx.CCIFG (x = 1 to 4), TA2CTL.TAIFG
INTISR[14]	Timer_A3	TA3CCTL0.CCIFG
INTISR[15]	Timer_A3	TA3CCTLx.CCIFG (x = 1 to 4), TA3CTL.TAIFG

The following command enables this interrupt. Note that because the interrupt number is less than 32, it occurs within the first half of the ISER register. Hence the reason we use **NVIC->ISER[0]** and not **NVIC->ISER[1]**.

```
NVIC->ISER[0] |= 1 << 9;
```

Finally, we need to write our interrupt function. In order to know what to name our function we can look in the startup file. Within this function, we will clear the interrupt flag in the TA0CTL register and then toggle the LED.

```
void TA0_N_IRQHandler(void){
    TA0CTL &= ~(0x01); //Clear interrupt flag
    P1OUT ^= 0x01; //Toggle LED
}
```

3.1.3 Starting the timer

Finally, we start the timer by defining which mode it should use. Again, this is defined in the TA0CTL register, specifically using bits 5 and 4 (MC). By default, the timer is in the stop mode, that is, it is halted. It can also be set to count up to a value in the TAxCCRO register, or to count to the maximum value of a 32-bit register, or count up to the value in TAxCCRO and then back down. We will use the continuous mode and count up to 0FFFFh. When it reaches this value, an interrupt will be triggered.

5-4	MC	RW	0h	Mode control. Setting MCx = 00h when Timer_A is not in use conserves power. 00b = Stop mode: Timer is halted 01b = Up mode: Timer counts up to TAxCCR0 10b = Continuous mode: Timer counts up to 0FFFFh 11b = Up/down mode: Timer counts up to TAxCCR0 then down to 0000h
-----	----	----	----	---

We select this mode by setting bit 5 (MC = 10b).

```
TA0CTL &= ~(1 << 4); // Set to use continuous counting mode
TA0CTL |= (1 << 5);
```

NOTE: This should be one of the very last things you do, since it will start the timer. Changing settings while the timer is active will lead to errors.

Your complete code should look similar to the following screenshot.

```
#include "msp.h"

void main(void)
{
    WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD; // stop watchdog timer

    // GPIO functionality is selected on desired pins
    P1SEL0 &= ~0x01;
    P1SEL1 &= ~0x01;

    P1DIR |= 0x01;
    P1OUT &= ~0x01;

    TA0CTL |= (1<<8); //Select ACLK as clock source
    TA0CTL &= ~(1<<9);

    TA0CTL |= (1<<1); //Enable interrupt for TimerA0
    NVIC->ISER[0] |= 1<<9; //Enable TimerA interrupts globally

    TA0CTL &= ~(1<<4); //Set to use continuous counting mode
    TA0CTL |= (1<<5);

    while(1){;}

}

void TA0_N_IRQHandler(void){
    TA0CTL &= ~(0x01); //Clear interrupt flag
    P1OUT ^= 0x01; //Toggle LED
}
```

Upload this code to your MSP432. You should observe that your LED is blinking at a slow rate.

3.1.4 EXERCISE 4

Change the clock of the timer to SMCLK.

3.1.5 EXERCISE 5

With the SMCLK as the clock source, change the clock divider to slow down the blinking rate of the LED.

Hint: You will need to change bits 6 and 7 of the TA0CTL register.

3.2 Task 3b – Improving Timer Resolution

In Task 2a, you changed the speed of the timer by either changing the clock driving the timer or by dividing the clock. This does not give flexibility in terms of timing resolution. If we wish to generate delays having a wide range of durations, we need to make use of the **up mode** of the timer instead of the continuous mode.

Learning outcome: Learn how to use the different modes of Timer A on the MSP432.

Let's begin with the code required to make the LED blink using SMCLK as the source (TASSEL = 10b).

```
#include "msp.h"

void main(void)
{
    WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;    // stop watchdog timer

    // GPIO functionality is selected on desired pins
    P1SEL0 &= ~0x01;
    P1SEL1 &= ~0x01;

    P1DIR |= 0x01;
    P1OUT &= ~0x01;

    TA0CTL &= ~(1<<8);    //Select SMCLK as clock source
    TA0CTL |= (1<<9);

    TA0CTL |= (1<<1);    //Enable interrupt for TimerA0
    NVIC->ISER[0] |= 1<<9; //Enable TimerA interrupts globally

    TA0CTL &= ~(1<<4);    //Set to use continuous counting mode
    TA0CTL |= (1<<5);

    while(1){;}

}

void TA0_N_IRQHandler(void){
    TA0CTL &= ~(0x01); //Clear interrupt flag
    P1OUT ^= 0x01; //Toggle LED
}
```

This should be identical to the code you developed in Part 2a. When uploaded, the LED should blink very fast but this blinking should still be noticeable.

In order to improve the resolution in terms of timing duration, we will stop using the continuous mode (which if you remember counts until the counting register overflows). Instead, we will use the up mode (MC = 01b) which only counts to the value stored in the TAxCCR0 register. By reducing the number in this register, the interrupt will be triggered sooner and your delay is shortened.

First, we store a value in the TAxCCR0 register (which is the value we wish to count to).

```
TA0CCR0 = 5000;
```

Then, we modify our existing code to the following:

```
TA0CCR0 = 5000;
TA0CTL |= (1<<4);    //Count up to TAxCCR0
TA0CTL &= ~(1<<5);
```

And that's it! The timer should now trigger the interrupt when it reaches the value stored in TAxCCR0. Combining this with an adjustable clock and clock divider means you can implement a wide range of delays.

3.2.1 EXERCISE 6

Implement an accurate 500 ms delay and demonstrate it with a blinking LED (toggling on/off).

4 Task 4 - UART

Learning outcome: In this task, you will learn how to use the UART peripheral to receive characters sent from the computer (keyboard) and echo them back.

NOTE: In previous tasks, we have set and reset bits using the (1<<x) notation. This is not always the neatest notation. Instead, we will start to use aliases defined in the header file. These aliases will correspond to particular bits within a register. Their name is usually identical to the name of the bits given in the register description. Sometimes, however, these bits are not defined, in which case we will resort to our previous approach.

We begin by first halting the watchdog timer as per usual. Then we need to set pins P1.2 and P1.3 to use their UART function by configuring P1SEL1 and P1SEL0. This setting will configure them to act as receiver (RXD) and transmitter (TXD) pins, respectively, as illustrated in Table 6-62 in the datasheet.

PIN NAME (P1.x)	x	FUNCTION	CONTROL BITS OR SIGNALS ⁽¹⁾		
			P1DIR.x	P1SEL1.x	P1SEL0.x
P1.2/UCARXD/UCASOMI	2	P1.2 (I/O)	I: 0; O: 1	0	0
		UCARXD/UCASOMI	X ⁽²⁾	0	1
		N/A	0	1	0
		DVSS	1		
		N/A	0	1	1
		DVSS	1		
P1.3/UCATXD/UCASIMO	3	P1.3 (I/O)	I: 0; O: 1	0	0
		UCATXD/UCASIMO	X ⁽²⁾	0	1
		N/A	0	1	0
		DVSS	1		
		N/A	0	1	1
		DVSS	1		

From Table 6-62 in the datasheet, we observe that P1SEL1.x has to be set to zero while P1SEL0.x has to be set to one. Referring to Table 12-2 in the TRM, we will notice that this configuration of the function select registers will select the primary module function. Hence, we can deduce that the UART functionality for pins P1.2 and P1.3 is a primary module function.

Table 12-2. I/O Function Selection

PxSEL1	PxSEL0	I/O Function
0	0	General purpose I/O is selected
0	1	Primary module function is selected
1	0	Secondary module function is selected
1	1	Tertiary module function is selected

Based on the above, it should be evident that we must set the bits of P1SEL0 (with the bits of P1SEL1 being in their default state of zero).

```
//Set P1.3 & P1.2 as UART pins
P1SEL0 |= BIT2 | BIT3;
```

We then configure the SMCLK to use a frequency of 12 MHz. This clock will be used subsequently to generate a baud rate.

```
//Set SMCLK freq to 12MHz
CS->KEY = CS_KEY_VAL;           // Unlock CS module for register access
CS->CTL0 = CS_CTL0_DCORSEL_3;   // Set DCO to 12MHz (nominal, center of 8-16MHz range)
CS->KEY = 0;                     // Lock CS module from unintended accesses
```

4.1 Configure the UART module

Before making any changes to the UART module, we must first halt it. This is accomplished by setting the UCSWRST bit in the UCA0CTLW0 register.

```
// Configure UART
UCA0CTLW0 |= UCSWRST; // Put eUSCI in reset
```

The UART peripheral has a wide range of settings that can be configured. For simplicity, we shall keep the majority of these at their default values, however, we need to setup the baud rate.

Most settings are contained in the UCA0CTLW0 register. We begin by selecting the clock source, in our case being SMCLK. Looking at the register description in Table 24-8 of the TRM, we observe that we need to set bit 7 (and optionally bit 6), in order to select SMCLK.

7-6	UCSSELx	RW	0h	eUSCI_A clock source select. These bits select the BRCLK source clock. 00b = UCLK 01b = ACLK 10b = SMCLK 11b = SMCLK
-----	---------	----	----	--

Here, we configure both these bits by writing 0x02 to the register.

```
UCA0CTLW0 |= (0x02<<6); // Configure eUSCI clock source for SMCLK
```

We now need to configure a number of settings to select the desired baud rate. We can do this by referring to Table 24-5 in the TRM.

BRCLK	Baud Rate	UCOS16	UCBRx	UCBRFx	UCBRSx	TX Error ⁽²⁾ (%)		RX Error ⁽²⁾ (%)	
						neg	pos	neg	pos
12000000	9600	1	78	2	0x0	0	0	0	0.04

In this case, we wish to run at a baud rate of 9600 using a 12 MHz clock. Hence, we need to carry out the following steps:

1. Set UCOS16.
2. Write 0x78 to UCBR.
3. Write 0x02 to UCBRF.
4. UCBRS may be left as 0 (its default value).

The location of these settings can again be found in the TRM, specifically in the UART register descriptions. UCOS16 is bit 0 of UCA0MCTLW. UCBR is all 16 bits of the UCBRW register. UCBRF is bits 7-4 in the UCA0MCTLW register.

```
UCA0MCTLW |= UCOS16;
UCA0BRW = 78; // 12000000/16/9600
UCA0MCTLW |= (2 << 4);
```

Having finished changing our settings we can enable the UART peripheral.

```
UCA0CTLW0 &= ~UCSWRST; // Initialize eUSCI
```

4.2 Enable interrupts

The next task is to enable interrupts. As always this must be done both at a local level, that is, a specific interrupt for the UART module, and at a global level by enabling UART interrupts in the NVIC registers. Recall that the NVIC interrupt number is found in the datasheet in Table 6-39.

INTISR[16]	eUSCI_A0	UART or SPI mode TX, RX, and Status Flags
------------	----------	---

In this task, we have used the alias for the interrupt number defined in the header.

```
EUSCIA0_IRQn = 16, /* 32 EUSCIA0 Interrupt */
```

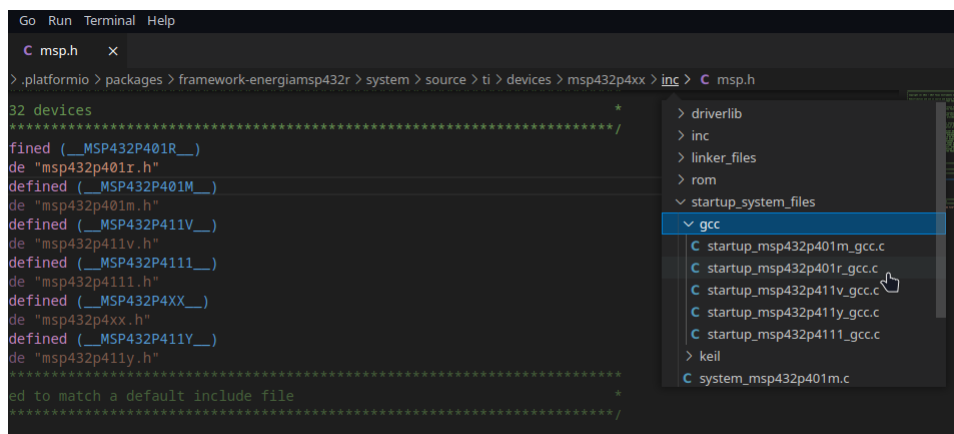
Subsequently, we enter a while loop to wait for incoming UART bytes.

```
//Enable interrupts
UCA0IE |= UCRXIE;
NVIC->ISER[0] |= 1 << (EUSCIA0_IRQn);

while(1){;}
```

4.3 Write the interrupt function (ISR – Interrupt Service Routine)

The final task is to write the function that will be called when the interrupt occurs, that is, the interrupt service routine (ISR). We first need to find the correct name to call our function. This has already been defined in the startup file called 'startup_msp432p401r_gcc.c'. You can find this file by going by opening msp.h file (right-click it in main.c and select "Go to Definition"), and navigating to inc > startup_system_files > gcc > startup_msp432p401r_gcc.c with the top navigation bar:



Browsing through the startup file, we find function prototypes for a number of interrupts. Our function for this peripheral must be called: EUSCIA0_IQRHandler.

```
extern void EUSCIA0_IRQHandler (void) __attribute__((weak,alias("Default_Handler")));
extern void EUSCIA1_IRQHandler (void) __attribute__((weak,alias("Default_Handler")));
extern void EUSCIA2_IRQHandler (void) __attribute__((weak,alias("Default_Handler")));
extern void EUSCIA3_IRQHandler (void) __attribute__((weak,alias("Default_Handler")));
```

Flags for the UART module are stored in the UCA0IFG register. We will check whether the RX flag has been set.

```
void EUSCIA0_IRQHandler(void) {
    //Clear flag
    if (UCA0IFG & UCRXIFG) {
```


Incoming bytes are written to UCA0RXBUF while output bytes are written to UCA0TXBUF. We will store the contents of the RX buffer, that is, the characters that the computer has sent to the microcontroller, in a char variable.

```
char resp = UCA0RXBUF;
```

Before writing our response to the TX buffer we need to check if it is empty. This is done by checking that the UCTXIFG flag is not set. We will wait until this flag is cleared, at which point, we are permitted to write to the TX buffer.

```
// Check if the TX buffer is empty first
while(!(UCA0IFG & UCTXIFG));
// Echo the received character back
UCA0TXBUF = resp;
```

The complete code is shown in the screenshot below.

```
void main(void)
{
    WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;    // stop watchdog timer

    //Set P1.3 & P1.2 as UART pins
    P1SEL0 |= BIT2 | BIT3;

    //Set SMCLK freq to 12MHz
    CS->KEY = CS_KEY_VAL;                          // Unlock CS module for register access
    CS->CTL0 = CS_CTL0_DCORSEL_3;                   // Set DCO to 12MHz (nominal, center of 8-16MHz range)
    CS->KEY = 0;                                    // Lock CS module from unintended accesses


    // Configure UART
    UCA0CTLW0 |= UCSWRST; // Put eUSCI in reset
    UCA0CTLW0 |= (0x02<<6); // Configure eUSCI clock source for SMCLK
    UCA0MCTLW |= UCOS16;
    UCA0BRW = 78; // 12000000/16/9600
    UCA0MCTLW |= (2 << 4);

    UCA0CTLW0 &= ~UCSWRST; // Initialize eUSCI

    //Enable interrupts
    UCA0IE |= UCRXIE;
    NVIC->ISER[0] |= 1 << (EUSCIA0_IRQn);

    while(1){;}
}

void EUSCIA0_IRQHandler(void){
    //Clear flag
    if (UCA0IFG & UCRXIFG){
        char resp = UCA0RXBUF;
        // Check if the TX buffer is empty first
        while(!(UCA0IFG & UCTXIFG));
        // Echo the received character back
        UCA0TXBUF = resp;
    }
}
```

Upload this code to your MSP432. Connect to 'Serial Monitor' by pressing  at the bottom VSCode bar. It will show you a list of available ports to connect serial monitor to. EUSCIA0 port on MSP432 board should correspond to the first port with comment 'XDS110 (03.00.00.05) Embed with CMSIS-DAP'. PlatformIO by default uses 9600 baud rate, so you don't need to configure it; however it can be changed in platformio.ini file in your project if needed. Once successfully connected, use your keyboard to send characters. If your code is correct, the characters you send should be echoed back to you.

4.4 EXERCISE 7

Modify the code above so that when the characters 'r', 'g' or 'b' are sent they toggle the respective colour on LED2. For example writing 'r' to the microcontroller should toggle the red LED whereas 'g' would toggle the green LED, and so forth.

5 Task 5 - ADC

Learning outcome: In this task, you will learn how to use the ADC peripheral to convert an analogue input to digital samples and print the output to the serial console.

NOTE: Once again, we will use the CMSIS notation for the purpose of practice.

We begin by halting the watchdog timer and configuring LED1 as an output.

```
WDT_A->CTL = WDT_A_CTL_PW |          // Stop WDT
              WDT_A_CTL_HOLD;

// GPIO Setup
P1->OUT &= ~BIT0;                      // Clear LED to start
P1->DIR |= BIT0;                       // Set P1.0/LED to output
```

Next, we must configure the pins to use the ADC peripheral. Here, we select pin 5.4 (A1) to use its ADC peripheral. We do this using the PxSEL registers, as we did for the TX/RX pins in the UART task. From the datasheet, we observe that we need to set both PxSEL0 and PxSEL1.

PIN NAME (P1.x)	x	FUNCTION	CONTROL BITS OR SIGNALS ⁽¹⁾		
			P1DIR.x	P1SEL1.x	P1SEL0.x
P5.4/A1	4	P5.4 (I/O)	I: 0; O: 1	0	0
		N/A	0	0	1
		DVSS	1		
		N/A	0	1	0
		DVSS	1		
		A1 ⁽²⁾	X	1	1

Question: What type of module is the ADC peripheral?

Let us now set P5SEL1 and P5SEL0 in our code.

```
P5->SEL1 |= BIT4;                      // Configure P5.4 for ADC
P5->SEL0 |= BIT4;
```

We are now ready to adjust the ADC settings. First we begin by turning the ADC on, this is done by setting the corresponding bit in the CTL0 register. Doing so powers up the ADC and enables us to start using the module.

```
//ADC14 on
ADC14->CTL0 |= ADC14_CTL0_ON;
```

Next, we need to select our sampling mode (sample-and-hold pulse-mode select). Here we select the source of the sampling signal to be the output of the sampling timer (ADC14SHP). See section 22.2.6.2 in the TRM for more details.

```
ADC14->CTL0 |= ADC14_CTL0_SHP;
```

The ADC has a selectable reference voltage. This is changed with the ADC14_MCTLN_INCH bits of the ADC14MCTL0 memory control register (see table 22.4 in the TRM for a list of the ADC14 registers). We select to use the AVCC which is simply the VCC supply of the board. If you want to adjust this, the TRM outlines the choices of voltage references.

```
ADC14->MCTL[0] |= ADC14_MCTLN_INCH_1; // A1 ADC input select; Vref=AVCC
```

And those are all the settings we will configure in this task. The last thing to do is to setup our interrupts. Again, they must be enabled both at a local level, the ADC14IFGR0 register, and the more global level with the NVIC module.

```
//Interrupts
ADC14->IER0 |= ADC14_IER0_IE0;        // Enable ADC conv complete interrupt
NVIC->ISER[0] |= 1 << (ADC14_IRQn);  // Enable ADC interrupt in NVIC module
```

We will write the interrupt function in a moment, but first let us finish our main function. Enabling a sample is done with the following code.

```
ADC14->CTL0 |= ADC14_CTL0_ENC | ADC14_CTL0_SC;
```

However, this function should only be called when the ADC has finished with the previous sample. We can check this status with the interrupt flags. If the interrupt flag is set, there is a sample waiting to be dealt with. Conversely, if the flag is reset, then the sample is complete and we are allowed to request another. So, we will request a sample only when the flag is cleared. This conditional statement will then be placed in a while loop so it runs continuously.

```
while(1){
    // Start sampling/conversion if ready
    if(~(ADC14->IFGR0 & 0x01)){
        ADC14->CTL0 |= ADC14_CTL0_ENC | ADC14_CTL0_SC;
    }
}
```

Your final main code should look similar to the following screenshot.

```
#include "msp.h"

void main(void) {

    WDT_A->CTL = WDT_A_CTL_PW |          // Stop WDT
                WDT_A_CTL_HOLD;

    // GPIO Setup
    P1->OUT &= ~BIT0;                    // Clear LED to start
    P1->DIR |= BIT0;                     // Set P1.0/LED to output
    P5->SEL1 |= BIT4;                    // Configure P5.4 for ADC
    P5->SEL0 |= BIT4;

    //ADC14 on
    ADC14->CTL0 |= ADC14_CTL0_ON;
    ADC14->CTL0 |= ADC14_CTL0_SHP;

    ADC14->MCTL[0] |= ADC14_MCTLN_INCH_1; // A1 ADC input select; Vref=AVCC

    //Interrupts
    ADC14->IER0 |= ADC14_IER0_IE0;      // Enable ADC conv complete interrupt
    NVIC->ISER[0] |= 1 << (ADC14_IRQn); // Enable ADC interrupt in NVIC module

    while(1){
        // Start sampling/conversion if ready
        if(~(ADC14->IFGR0 & 0x01)){
            ADC14->CTL0 |= ADC14_CTL0_ENC | ADC14_CTL0_SC;
        }
    }
}
```

Last but not least, let us write the interrupt function. The name of the interrupt can be found in the startup file generated when you create the project (as per previous tasks). In this function we will check the value of the ADC. If the value is above 0.5 of our reference voltage, we will turn on LED1 else we will turn off LED1.

The value of the ADC is stored in the ADC14MEM0 register.

```
// ADC14 interrupt service routine
void ADC14_IRQHandler(void) {
    if (ADC14->MEM[0] >= 0x7FF)           // ADC12MEM0 = A1 > 0.5AVcc?
        P1->OUT |= BIT0;                 // P1.0 = 1
    else
        P1->OUT &= ~BIT0;                 // P1.0 = 0
}
```

And that's it! You can test the code by either tapping on p5.4 on your launchpad, which should cause the LED to flash. Alternatively, you can use a potentiometer to vary the voltage applied to pin 5.4, which will turn LED1 on or off depending on whether the value is above or below the threshold.

5.1 EXERCISE 8

Set three thresholds that will toggle the three colours on the RGB LED depending on the value of the voltage applied to one of the MSP432's ADC input channels.

5.2 EXERCISE 9

Change the resolution of the conversion result from the default resolution (14-bit) to 10-bit.

5.3 EXERCISE 10

Challenging: Change the brightness of one of the on-board LEDs (using PWM) based on the intensity of light shone upon an LDR connected to one of the ADC input channels.