Lab 4: Interrupts and the Timer

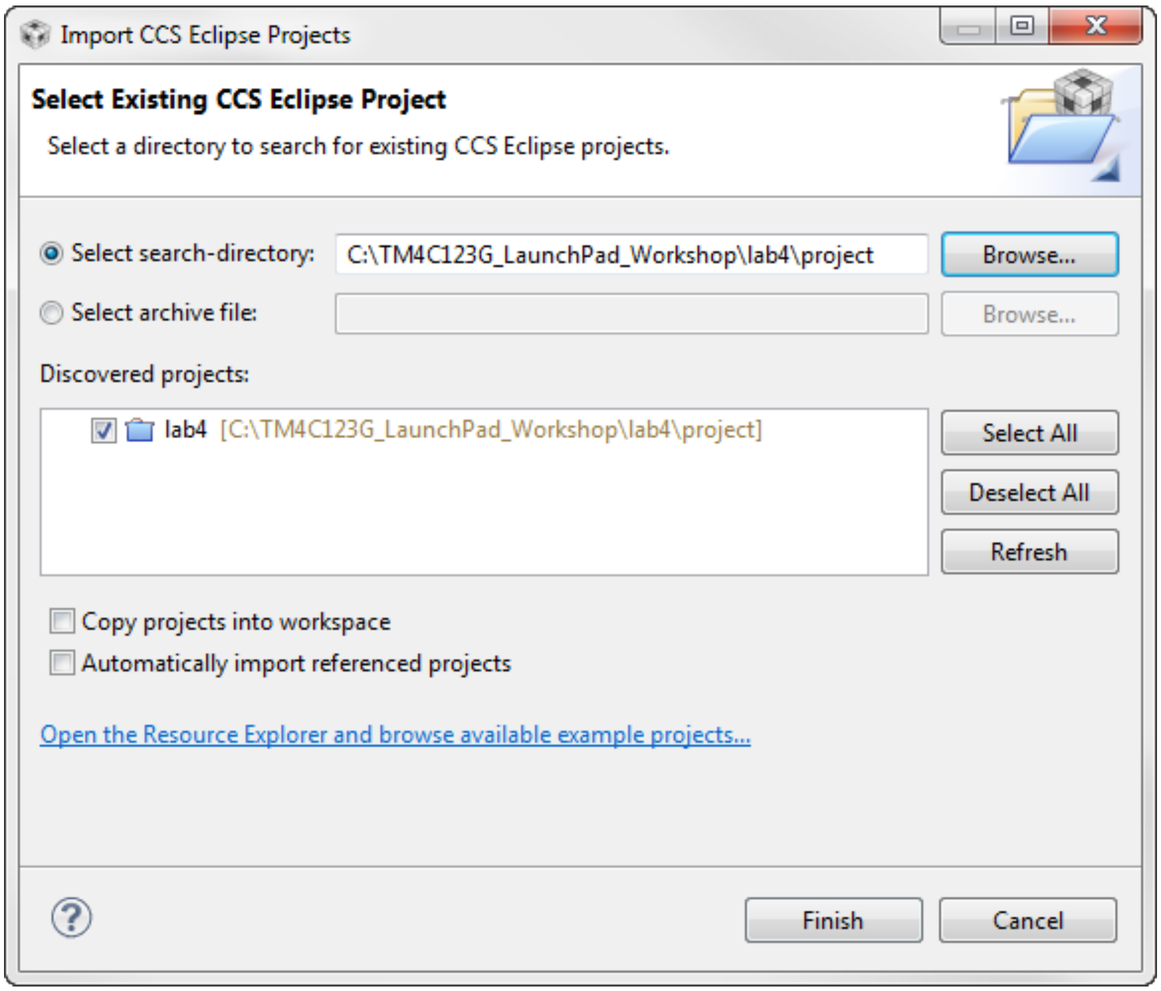
Procedure

Import Lab4 Project

1. We have already created the Lab4 project for you with an empty `main.c`, a startup file and all necessary project and build options set.

   ► Maximize Code Composer and click Project → Import Existing CCS Eclipse Project. Make the settings show below and click Finish.

   Make sure that the "Copy projects into workspace" checkbox is unchecked.


Import CCS Eclipse Projects

Select Existing CCS Eclipse Project

Select a directory to search for existing CCS Eclipse projects.

(●) Select search-directory: C:\TM4C123G_LaunchPad_Workshop\lab4\project

( ) Select archive file:

Discovered projects:

[✓] lab4 [C:\TM4C123G_LaunchPad_Workshop\lab4\project]

[ ] Copy projects into workspace

[ ] Automatically import referenced projects

Open the Resource Explorer and browse available example projects...

► Close the `lab3` project by right-clicking on `lab3` in the Project Explorer pane and selecting *Close Project*.

Header Files

2. ► Expand the lab by clicking the ▷ to the left of lab4 in the Project Explorer pane. Open `main.c` for editing by double-clicking on it.

   ► Type (or copy/paste) the following seven lines into `main.c` to include the header files needed to access the TivaWare APIs :

```
#include <stdint.h>
#include <stdbool.h>
#include "inc/tm4c123gh6pm.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/sysctl.h"
#include "driverlib/interrupt.h"
#include "driverlib/gpio.h"
#include "driverlib/timer.h"
```

   Several new include headers are needed to support the hardware we'll be using in this code:

   `tm4c123gh6pm.h`: Definitions for the interrupt and register assignments on the Tiva C Series device on the LaunchPad board

   `interrupt.h` : Defines and macros for NVIC Controller (Interrupt) API of `driverLib`. This includes API functions such as `IntEnable` and `IntPrioritySet`.

   `timer.h` : Defines and macros for Timer API of `driverLib`. This includes API functions such as `TimerConfigure` and `TimerLoadSet`.

   Note that there are no question marks shown in the editor pane beside your include statements. The paths have already been set up for you in the imported project.

Main() Function

3. We're going to compute our timer delays using the variable ui32`Period`. Create main() along with an unsigned 32-bit integer (that's why the variable is called ui32`Period`) for this computation.

   ► Leave a line for spacing and type (or cut/paste) the following after the previous lines:

```
int main(void)
```

```
{
        uint32_t ui32Period;
}
```

Clock Setup

4. Configure the system clock to run at 40MHz () with the following call.

   ► Leave a blank line for spacing and enter this line of code inside `main()`:

```
SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_XTAL_16MHZ|SYSCTL_OSC_MAIN);
```

GPIO Configuration

5. Like the previous lab, we need to enable the GPIO peripheral and configure the pins connected to the LEDs as outputs.

   ► Leave a line for spacing and add these lines after the last ones:

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3);
```

Timer Configuration

6. Again, before calling any peripheral specific `driverLib` function we must enable the clock to that peripheral. If you fail to do this, it will result in a Fault ISR (address fault).

   The second statement configures Timer 0 as a 32-bit timer in periodic mode. Note that when Timer 0 is configured as a 32-bit timer, it combines the two 16-bit timers Timer 0A and Timer 0B. See the General Purpose Timer chapter of the device datasheet for more information. `TIMER0_BASE` is the start of the timer registers for Timer0 in, you guessed it, the peripheral section of the memory map.

   ► Add a line for spacing and type the following lines of code after the previous ones:

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);
TimerConfigure(TIMER0_BASE, TIMER_CFG_PERIODIC);
```

Calculate Delay

7. To toggle a GPIO at 10Hz and a 50% duty cycle, you need to generate an interrupt at ½ of the desired period. First, calculate the number of clock cycles required for a 10Hz period by calling `SysCtlClockGet()` and dividing it by your desired frequency. Then divide that by two, since we want a count that is ½ of that for the interrupt.

   This calculated period is then loaded into the Timer's Interval Load register using the `TimerLoadSet` function of the driverLib Timer API. Note that you have to subtract one from the timer period since the interrupt fires at the zero count.

► Add a line for spacing and add the following lines of code after the previous ones:

```
Ui32Period = (SysCtlClockGet() / 10) / 2;
TimerLoadSet(TIMER0_BASE, TIMER_A, ui32Period -1);
```

Interrupt Enable

8. Next, we have to enable the interrupt … not only in the timer module, but also in the NVIC (the Nested Vector Interrupt Controller, the Cortex M4's interrupt controller). `IntMasterEnable()` is the master interrupt enable API for all interrupts. `IntEnable` enables the specific vector associated with Timer0A. `TimerIntEnable`, enables a specific event within the timer to generate an interrupt. In this case we are enabling an interrupt to be generated on a timeout of Timer 0A.

   ► Add a line for spacing and type the next three lines of code after the previous ones:

```
IntEnable(INT_TIMER0A);
TimerIntEnable(TIMER0_BASE, TIMER_TIMA_TIMEOUT);
IntMasterEnable();
```

Timer Enable

9. Finally we can enable the timer. This will start the timer and interrupts will begin triggering on the timeouts.

   ► Add a line for spacing and type the following line of code after the previous ones:

```
TimerEnable(TIMER0_BASE, TIMER_A);
```

Main Loop

10. The main loop of the code is simply an empty `while(1)` loop since the toggling of the GPIO will happen in the interrupt service routine.

    ► Add a line for spacing and add the following lines of code after the previous ones:

```
while(1)
{
}
```

Timer Interrupt Handler

11. Since this application is interrupt driven, we must add an interrupt handler or ISR for the Timer. In the interrupt handler, we must first clear the interrupt source and then toggle the GPIO pin based on the current state. Just in case your last program left any of the LEDs on, the first `GPIOPinWrite()` call turns off all three LEDs. Writing a 4 to pin 2 lights the blue LED.

► Add a line for spacing and add the following lines of code **after** the final closing brace of `main()`.

```
void Timer0IntHandler(void)

{
 // Clear the timer interrupt
 TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT);

 // Read the current state of the GPIO pin and
 // write back the opposite state

 if(GPIOPinRead(GPIO_PORTF_BASE, GPIO_PIN_2))
 {
  GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 0);
 }

 else
 {
  GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, 4);
 }
}
```

► If your indentation looks wrong, select all the code by pressing Ctrl-A, right-click on the selected code and pick *Source* → *Correct Indentation*.

► Click the *Save* button to save your work.

Your code should look something like this:

```
#include <stdint.h>
#include <stdbool.h>
#include "inc/tm4c123gh6pm.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/sysctl.h"
#include "driverlib/interrupt.h"
#include "driverlib/gpio.h"
#include "driverlib/timer.h"

int main(void)
{
	uint32_t ui32Period;

	SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_XTAL_16MHZ|SYSCTL_OSC_MAIN);

	SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
	GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3);

	SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);
	TimerConfigure(TIMER0_BASE, TIMER_CFG_PERIODIC);

	ui32Period = (SysCtlClockGet() / 10) / 2;
	TimerLoadSet(TIMER0_BASE, TIMER_A, ui32Period -1);

	IntEnable(INT_TIMER0A);
	TimerIntEnable(TIMER0_BASE, TIMER_TIMA_TIMEOUT);
	IntMasterEnable();

	TimerEnable(TIMER0_BASE, TIMER_A);

	while(1)
```

```
        {
        }
}

void Timer0IntHandler(void)
{
        // Clear the timer interrupt
        TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT);

        // Read the current state of the GPIO pin and
        // write back the opposite state
        if(GPIOPinRead(GPIO_PORTF_BASE, GPIO_PIN_2))
        {
                GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 0);
        }
        else
        {
                GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, 4);
        }
}
```

If you're having problems, this code is contained in `main.txt` in your project folder.

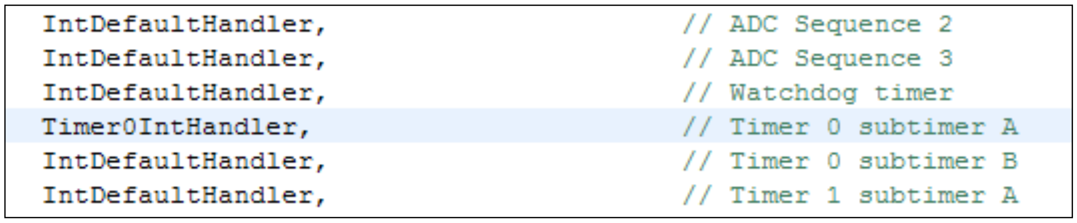
Startup Code

12. ► Open `startup_ccs.c` for editing. This file contains the vector table that we discussed during the presentation. We included the file in the project that you imported.

► Browse the file and look for the `Timer 0 subtimer A` vector.

When that timer interrupt occurs, the NVIC will look in this vector location for the address of the ISR (interrupt service routine). That address is where the next code fetch will happen.

► You need to **carefully** find the appropriate vector position and replace `IntDefaultHandler` with the name of your Interrupt handler (We suggest that you copy/paste this). In this case you will add `Timer0IntHandler` to the position with the comment "`Timer 0 subtimer A`" as shown below:



```
    IntDefaultHandler,                      // ADC Sequence 2
    IntDefaultHandler,                      // ADC Sequence 3
    IntDefaultHandler,                      // Watchdog timer
    Timer0IntHandler,                       // Timer 0 subtimer A
    IntDefaultHandler,                      // Timer 0 subtimer B
    IntDefaultHandler,                      // Timer 1 subtimer A
```

You also need to declare this function at the top of this file as external. This is necessary for the compiler to resolve this symbol.

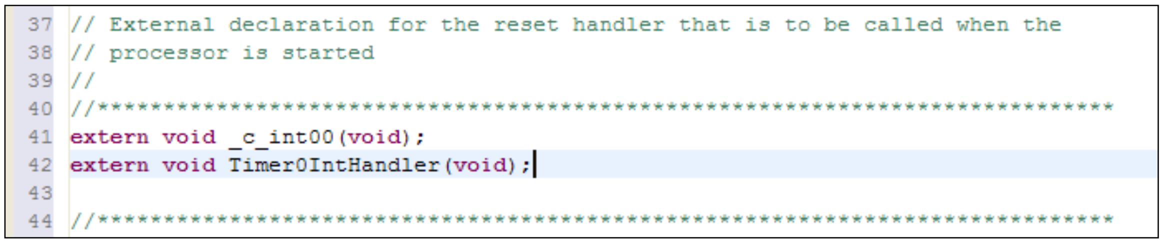
► Find the line containing:

```
extern void _c_int00(void);
```

► and add:

`extern void Timer0IntHandler(void);`

right below it as shown below:



```
// External declaration for the reset handler that is to be called when the
// processor is started
//
//*****************************************************************************
extern void _c_int00(void);
extern void Timer0IntHandler(void);

//*****************************************************************************
```

By the way, the `IntDefaultHandler` handler will catch any “unintentional” interrupts that may occur. Since this handler is also a `while(1)` loop, you might want to consider changing it for your production system.

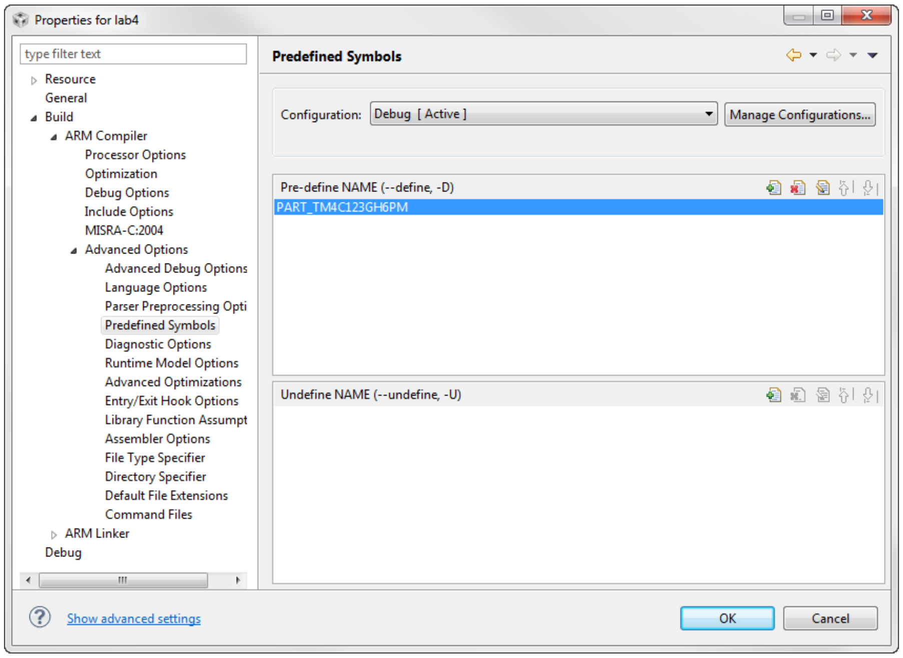
► Click the *Save* button.

Pre-defined Name

13. In order for the compiler to find the correct interrupt mapping it needs to know exactly which part is being used. We do that through a build option called a *pre-defined name*.

► Right-click on lab4 in your Project Explorer and select *Properties*.

► Under *Build* → *ARM Compiler* → *Advanced Options* → *Predefined Symbols*, click the add button for top pane and add `PART_TM4C123GH6PM` as the *pre-define NAME* as shown below.



Properties for lab4

Predefined Symbols

Configuration: Debug [Active]

Pre-define NAME (--define, -D)

PART_TM4C123GH6PM

Undefine NAME (--undefine, -U)

This property will already be set when you import the following labs.

► Click OK.

Compile, Download and Run The Code

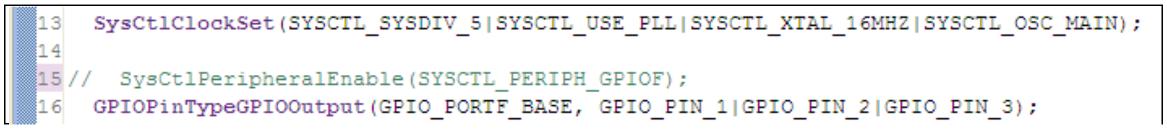
14. ► Click the Debug button on the menu bar to compile and download your application. If you have any issues, correct them, and then click the Debug button again. (You were careful about that interrupt vector placement, weren't you?) After a successful build, the CCS Debug perspective will appear.

    ► Click the Resume button to run the program that was downloaded to the flash memory of your device. The blue LED should be flashing quickly on your LaunchPad board.

    When you're done, ► click the Terminate button to return to the Editing perspective.

Exceptions

15. ► Find the line of code that enables the GPIO peripheral and comment it out as shown below:



```
  SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_XTAL_16MHZ|SYSCTL_OSC_MAIN);

//  SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
  GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3);
```

    Now our code will be accessing the peripheral without the peripheral clock being enabled. This should generate an exception.

16. ► Compile and download your application by clicking the Debug button on the menu bar. Save your changes when you're prompted. ► Click the Resume button to run the program. **What?!** The program seems to run just fine doesn't it? The blue LED is flashing. The problem is that we enabled the peripheral in our earlier run of the code … and we never disabled it or power cycled the part.

17. ► Click the Terminate button to return to the editing perspective. ► Cycle the power on the board using the power switch. This will return the peripheral registers to their default power-up states.

    The code with the enable line commented out is now running, but note that the blue LED isn't flashing now.

18. ► Compile and download your application by clicking the Debug button on the menu bar, then click the Resume button to run the program. Again, the blue LED should not be blinking.

► Click the Suspend button to stop execution. You should see that execution has trapped inside the `FaultISR()` interrupt routine. All of the exception ISRs trap in while(1) loops in the provided code. That probably isn't the behavior you want in your production code.

19. ► Back in `main.c`, uncomment the line enabling the GPIO port. ► Compile, download and run your code to make sure everything works properly. When you're done, ► click the Terminate button to return to the Editing perspective

20. ► Close the lab4 project. Minimize CCS.

    Homework Idea: Investigate the Pulse-Width Modulation capabilities of the general purpose timer. Program the timer to blink the LED faster than your eye can see, usually above 30Hz and use the pulse width to vary the apparent intensity. Write a loop to make the intensity vary periodically.

Follow the submission guideline to be awarded points for this Lab.

Task00: Execute the supplied code, no submission required.

Task 01: Change the toggle of the GPIO at 2 Hz using Timer0 with 75% duty cycle and verify the waveform generated.

Task 02: Include a GPIO Interrupt to Task 02 from switch SW2 to turn ON and the LED for 1.5 sec. Use a Timer1 to calculate the 1.5 sec delay. The toggle of the GPIO is suspended when executing the interrupt.

Follow the submission guideline to be awarded points for this Lab.

Submit the following for all Labs:

1. In the document, for each task submit the modified or included code (only) with highlights and justifications of the modifications. Also include the comments.
2. Use the previously create a Github repository with a random name (no CPE/403, Lastname, Firstname). Place all labs under the root folder TIVAC, sub-folder named LABXX, with one document and one video link file for each lab, place modified c files named as LabXX-TYY.c.
3. If multiple c files or other libraries are used, create a folder LabXX-TYY and place these files inside the folder.
4. The folder should have a) Word document (see template), b) source code file(s) with startup_ccs.c and other include files, c) text file with youtube video links (see template).