

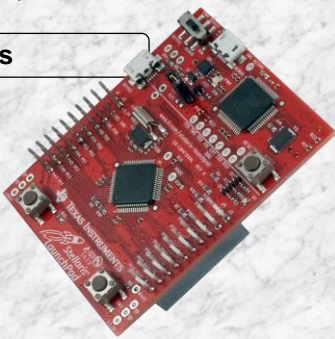
Interrupts and the Timers

Introduction

This chapter will introduce you to the use of interrupts on the ARM® Cortex-M4® and the general purpose timer module (GPTM). The lab will use the timer to generate interrupts. We will write a timer interrupt service routine (ISR) that will blink the LED.

Agenda

- Introduction to ARM® Cortex™-M4F and Peripherals
- Code Composer Studio
- Introduction to StellarisWare,
Initialization and GPIO
- Interrupts and the Timers**
- ADC12
- Hibernation Module
- USB
- Memory
- Floating-Point
- BoosterPacks and grLib
- Synchronous Serial Interface
- UART
- μDMA



NVIC...

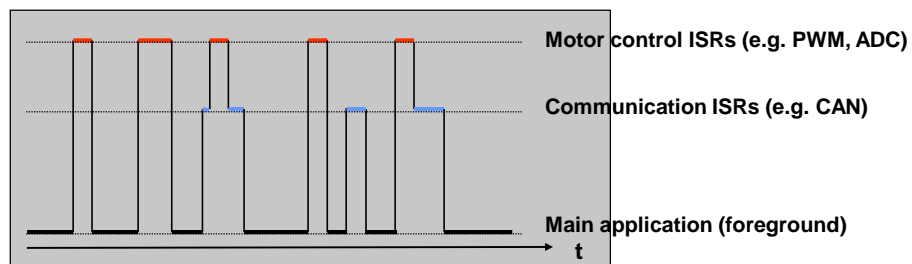
Chapter Topics

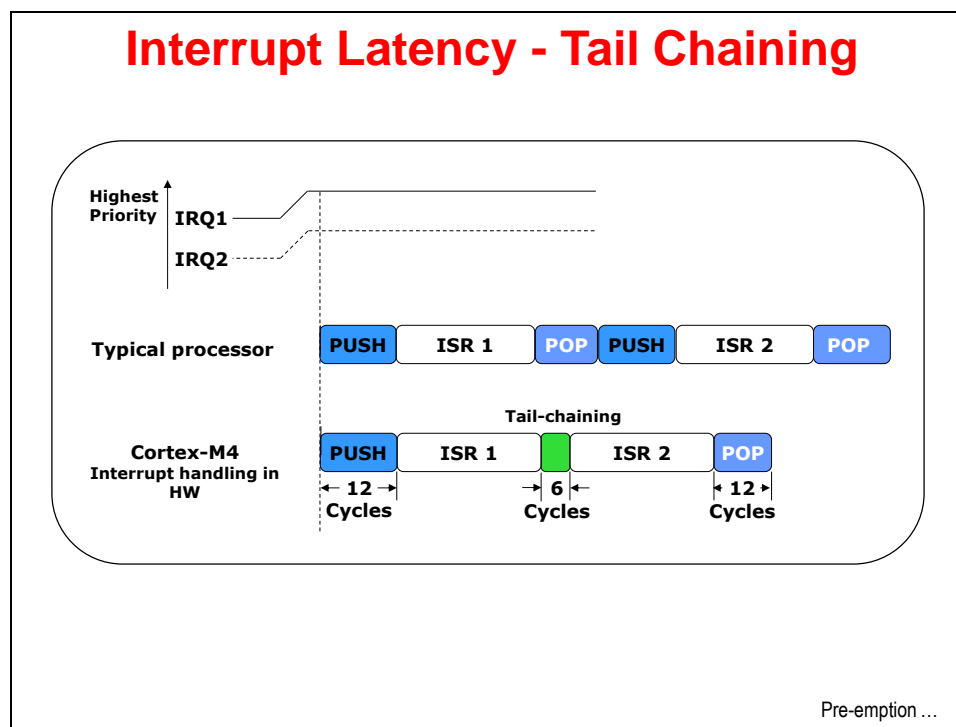
Interrupts and the Timers	4-1
<i>Chapter Topics.....</i>	<i>4-2</i>
<i>Cortex-M4 NVIC.....</i>	<i>4-3</i>
<i>Cortex-M4 Interrupt Handling and Vectors.....</i>	<i>4-7</i>
<i>Genral Purpose Timer Module</i>	<i>4-9</i>
<i>Lab 4: Interrupts and the Timer.....</i>	<i>4-10</i>
Objective.....	4-10
Procedure.....	4-11

Cortex-M4 NVIC

Nested Vectored Interrupt Controller (NVIC)

- ◆ Handles exceptions and interrupts
- ◆ 8 programmable priority levels, priority grouping
- ◆ 7 exceptions and 65 Interrupts
- ◆ Automatic state saving and restoring
- ◆ Automatic reading of the vector table entry
- ◆ Pre-emptive/Nested Interrupts
- ◆ Tail-chaining
- ◆ Deterministic: always 12 cycles or 6 with tail-chaining



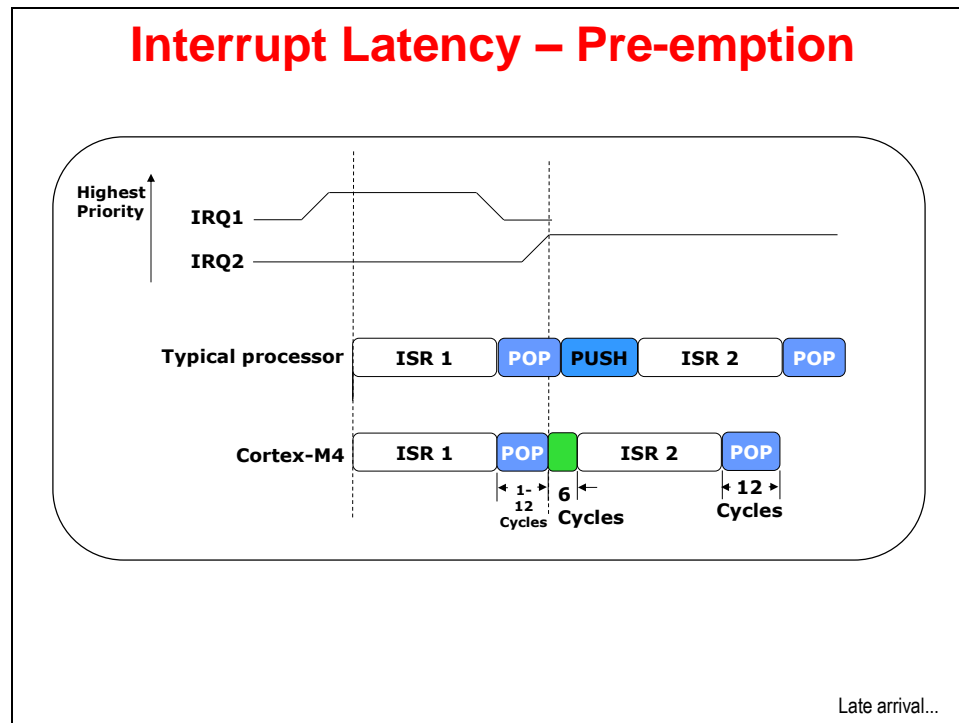


In the above example, two interrupts occur simultaneously.

In most processors, interrupt handling is fairly simple and each interrupt will start a PUSH PROCESSOR STATE – RUN ISR – POP PROCESSOR STATE process. Since IRQ1 was higher priority, the NVIC causes the CPU to run it first. When the interrupt handler (ISR) for the first interrupt is complete, the NVIC sees a second interrupt pending, and runs that ISR. This is quite wasteful since the middle POP and PUSH are moving the exact same processor state back and forth to stack memory. If the interrupt handler could have seen that a second interrupt was pending, it could have “tail-chained” into the next ISR, saving power and cycles.

The Stellaris NVIC does exactly this. It takes only 12 cycles to PUSH and POP the processor state. When the NVIC sees a pending ISR during the execution of the current one, it will “tail-chain” the execution using just 6 cycles to complete the process.

If you are depending on interrupts to be run quickly, the Stellaris devices offer a huge advantage here.

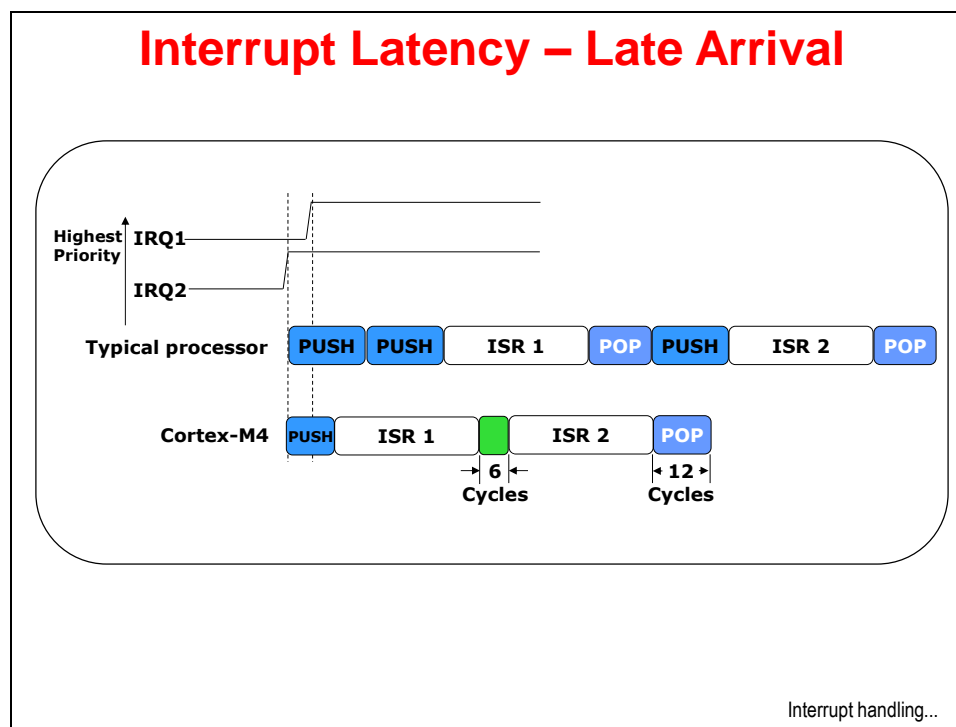


In this example, the processor was in the process of popping the processor status from the stack for the first ISR when a second ISR occurred.

In most processors, the interrupt controller would complete the process before starting the entire PUSH-ISR-POP process over again, wasting precious cycles and power doing so.

The Stellaris NVIC is able to stop the POP process, return the stack pointer to the proper location and “tail-chain” into the next ISR with only 6 cycles.

Again, this is a huge advantage for interrupt handling on Stellaris devices.



In this example, a higher priority interrupt has arrived just after a lower priority one.

In most processors, the interrupt controller is smart enough to recognize the late arrival of a higher priority interrupt and restart the interrupt procedure accordingly.

The Stellaris NVIC takes this one step further. The PUSH is the same process regardless of the ISR, so the Stellaris NVIC simply changes the fetched ISR. In between the ISRs, “tail chaining” is done to save cycles.

Once more, Stellaris devices handle interrupts with lower latency.

Cortex-M4 Interrupt Handling and Vectors

Cortex-M4[®] Interrupt Handling

Interrupt handling is automatic. No instruction overhead.

Entry

- ◆ Automatically pushes registers R0–R3, R12, LR, PSR, and PC onto the stack
- ◆ In parallel, ISR is pre-fetched on the instruction bus. ISR ready to start executing as soon as stack PUSH complete

Exit

- ◆ Processor state is automatically restored from the stack
- ◆ In parallel, interrupted instruction is pre-fetched ready for execution upon completion of stack POP

Exception types...

Cortex-M4[®] Exception Types

Vector Number	Exception Type	Priority	Vector address	Descriptions
1	Reset	-3	0x04	Reset
2	NMI	-2	0x08	Non-Maskable Interrupt
3	Hard Fault	-1	0x0C	Error during exception processing
4	Memory Management Fault	Programmable	0x10	MPU violation
5	Bus Fault	Programmable	0x14	Bus error (Prefetch or data abort)
6	Usage Fault	Programmable	0x18	Exceptions due to program errors
7-10	Reserved	-	0x1C - 0x28	
11	SVCall	Programmable	0x2C	SVC instruction
12	Debug Monitor	Programmable	0x30	Exception for debug
13	Reserved	-	0x34	
14	PendSV	Programmable	0x38	
15	SysTick	Programmable	0x3C	System Tick Timer
16 and above	Interrupts	Programmable	0x40	External interrupts (Peripherals)

Vector Table...

Cortex-M4® Vector Table

- ◆ After reset, vector table is located at address 0
- ◆ Each entry contains the address of the function to be executed
- ◆ The value in address 0x00 is used as starting address of the Main Stack Pointer (MSP)
- ◆ Vector table can be relocated by writing to the VTABLE register (must be aligned on a 1KB boundary)
- ◆ Open startup_ccs.c to see vector table coding

Exception number	IRQ number	Offset	Vector
154	138	0x0268	IRQ131
...
18	2	0x004C	IRQ2
17	1	0x0048	IRQ1
16	0	0x0044	IRQ0
15	-1	0x0040	Systick
14	-2	0x003C	PendSV
13		0x0038	Reserved
12			Reserved for Debug
11	-5		SVCall
10		0x002C	
9			Reserved
8			
7			
6	-10		Usage fault
5	-11	0x0018	Bus fault
4	-12	0x0014	Memory management fault
3	-13	0x0010	Hard fault
2	-14	0x000C	NMI
1		0x0008	Reset
		0x0004	Initial SP value
		0x0000	

GPTM...

General Purpose Timer Module

General Purpose Timer Module

- ◆ Six 16/32-bit and Six 32/64-bit general purpose timers
- ◆ Twelve 16/32-bit and Twelve 32/64-bit capture/compare/PWM pins
- ◆ **Timer modes:**
 - One-shot
 - Periodic
 - Input edge count or time capture with 16-bit prescaler
 - PWM generation (separated only)
 - Real-Time Clock (concatenated only)
- ◆ **Count up or down**
- ◆ **Simple PWM (no deadband generation)**
- ◆ **Support for timer synchronization, daisy-chains, and stalling during debugging**
- ◆ **May trigger ADC samples or DMA transfers**



Lab...

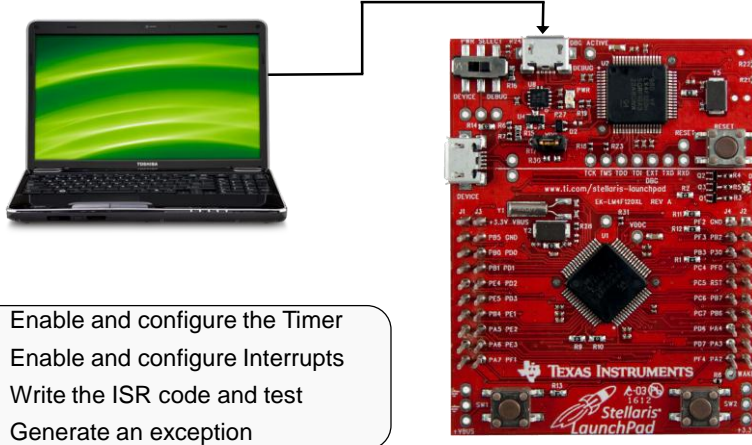
Lab 4: Interrupts and the Timer

Objective

In this lab we'll set up the timer to generate interrupts, and then write the code that responds to the interrupt ... flashing the LED. We'll also experiment with generating an exception, by attempting to configure a peripheral before it's been enabled.

Lab 4: Interrupts and the GP Timer

USB Emulation Connection



The diagram illustrates the setup for the lab. On the left is a laptop with a green screen. A line connects the laptop to a red Texas Instruments Stellaris LaunchPad on the right. The LaunchPad is labeled 'EK-LM4F120XL REV A' and 'www.ti.com/stellaris-launchpad'. It features a central microcontroller, various pins, and a USB connector. The text 'USB Emulation Connection' is positioned above the connection line.

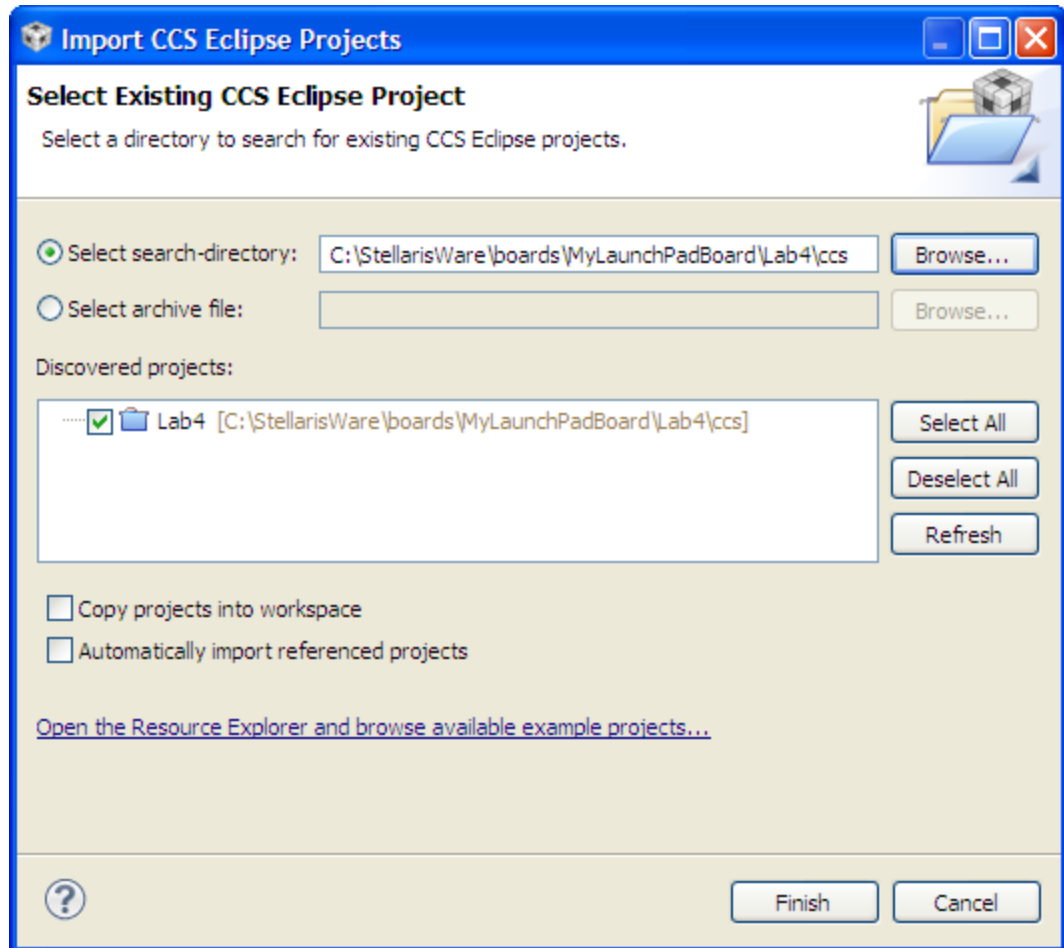
- ◆ Enable and configure the Timer
- ◆ Enable and configure Interrupts
- ◆ Write the ISR code and test
- ◆ Generate an exception

Agenda ...

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.**



Header Files

2. Expand the lab by clicking the + or ▾ 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 StellarisWare APIs :

```
#include "inc/hw_ints.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"
```

hw_ints.h : Macros that define the interrupt assignment on Stellaris devices (NVIC)

hw_memmap.h : Macros defining the memory map of the Stellaris device. This includes defines such as peripheral base address locations, e.g., `GPIO_PORTF_BASE`

hw_types.h : Defines common types and macros such as `tBoolean` and `HWREG(x)`

sysctl.h : Defines and macros for System Control API of driverLib. This includes API functions such as `SysCtlClockSet` and `SysCtlClockGet`.

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

gpio.h : Defines and macros for GPIO API of driverLib. This includes API functions such as `GPIOPinTypePWM` and `GPIOPinWrite`.

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

Main() Function

- We're going to compute our timer delays using the variable `Period`. Create `main()` along with an unsigned-long variable (that's why the variable is called `ulPeriod`) for this computation. Leave a line for spacing and type (or cut/paste) the following after the previous lines:

```
int main(void)
{
    unsigned long ulPeriod;
}
```

Clock Setup

- Configure the system clock to run at 40MHz (like in Lab3) 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

- Like the previous lab, we need to enable the GPIO peripheral and set 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

- Again, before calling any peripheral specific `driverLib` function we must enable the clock to that peripheral (RCGCn register). 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_32_BIT_PER);
```

Calculate Delay

7. To toggle a GPIO at 10Hz and a 50% duty cycle, you need to generate an interrupt at $\frac{1}{2}$ 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 $\frac{1}{2}$ 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:

```
ulPeriod = (SysCtlClockGet() / 10) / 2;
TimerLoadSet(TIMER0_BASE, TIMER_A, ulPeriod - 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 for all interrupts. `IntEnable` enables the specific vector associated with the Timer. `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 following 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)` since the toggling of the GPIO will happen in the interrupt 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 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, remember how we corrected it in the previous lab.

12. Click the Save button to save your work. Your code should look something like this:

```
#include "inc/hw_ints.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)
{
    unsigned long ulPeriod;

    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_32_BIT_PER);

    ulPeriod = (SysCtlClockGet() / 10) / 2;
    TimerLoadSet(TIMER0_BASE, TIMER_A, ulPeriod - 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);
    }
}
```


Startup Code

- Open `startup_ccs.c` for editing. This file contains the vector table that we discussed during the presentation. 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 will 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:

```
37 // External declaration for the reset handler that is to be called when the
38 // processor is started
39 //
40 //*****
41 extern void _c_int00(void);
42 extern void Timer0IntHandler(void);
43
44 //*****
```

Click the Save button.

Compile, Download and Run The Code

14. Compile and download your application by clicking the Debug button  on the menu bar. 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 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:

```
13 SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_XTAL_16MHZ|SYSCTL_OSC_MAIN);
14
15 // SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
16 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, then 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 ... we never disabled it or power cycled the part.
17. Click the Terminate button  to return to the editing perspective. Remove/reinstall the micro-USB cable on the LaunchPad board to cycle the power. This will return the peripheral registers to their default power-up states.

The code that you just downloaded is 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. Nothing much should appear to be happening. 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. Remove the comment and compile, download and run your code again to make sure everything works properly. When you're done, click the Terminate button  to return to the Editing perspective and close the Lab4 project. Minimize CCS.
20. **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.



You're done.

