Chandler Janzen
Rev 3

CME331 – Lab 3

Microcontrollers
Interrupts and 7-Segment Displays

Safety: In this lab, voltages used are less than 15 volts and this is not normally dangerous to humans. However, you should assemble or modify a circuit when power is disconnected and do not touch a live circuit if you have a cut or break in the skin.
Preparing for emergencies protects our lives and property. A classroom emergency posting is located in each classroom and lab near the main door of the room. Students are advised to review and be familiar with this classroom emergency posting and the College of Engineering Emergency Response Plan (ERP) The building must be evacuated when an alarm sounds for more than 10 seconds.

Objectives: By the end of this lab, you should have learned the following:

1. How to configure a hardware based interrupt.
2. How to configure and use a hardware based timer with interrupts.
3. How to use a timer when it is configured to use a hardware based interrupt.
4. Good coding practices related to interrupt service routines (ISRs).
5. How to adjust the interrupt vector table to point to a new ISR function.
6. How to debounce buttons and increase code functionality using hardware based timers.
7. How to use a 7-Segment display driven by latches.

Expectation: Getting the lab done is good. Getting expected results is better. Learning how to work with MCUs properly so that your code and work is modular, expandable, reusable and robust is best. Although these labs are to be done individually, you should envision yourself working on a small portion of a larger project and design team.

Code Submission: Always create your CCS projects with your NSID(s) in the project name (your NSID should be in the 'tree' on the left of CCS). Generic naming of projects (such as 'Lab 1' may result in a grade of 0. Always submit the **entire** CCS project folder (which should already include your NSID in the folder name) with relevant outputs electronically in a zip file (named <NSID>-<Lab#-Part#>) for each part (if applicable) through Canvas/BlackBoard no later than a week after your lab. Enhance the readability of your code by adding explanatory comments wherever appropriate. Always include your name, NSID, and date at the top of each code file. If you are unsure about what is expected, ask your instructor.

Working with a Partner: If you've worked with a partner, please indicate your partner's Name and NSID at the top of your PDF Document, as well as at the top of any source files (as a comment). Only one submission per pair is required.

Late Policy: Each lab is due one week after the scheduled lab time. Every student is granted 2 days late submission without penalty and 10% per day after; however, all submissions must be in prior to the start of the next lab.

1 Introduction

In the previous lab you learned how to make a software based timer and how to use a hardware based timer without an interrupt. Software based timers need to be managed by the program, and are difficult to keep accurate and consistent. Hardware timers increase the functionality of our code an allow us to keep a more accurate time base in our designs, but are still subject to being checked at the correct time. In this lab you will explore how to use an internal hardware based timer configured with interrupts to allow for more real-time processing. The concept of hardware based interrupts will be introduced through the use of the timers as well as the switches. We will also explore how to use a 7-segment display.

2 Background

For your convenience, here is a link to the MCU Datasheet: TM4C123GH6PM Microcontroller Datasheet

2.1 Hardware Interrupts

A hardware interrupt occurs when the MCU is configured to generate an interrupt signal when a specific condition is met, and the interrupt is triggered when the condition is actually met. The trigger for an interrupt will cause the MCU to call a special block of code (in this case, it will be a special function that we define). This function call occurs by the hardware in the MCU - it is not called by the software that you write. When the condition for the interrupt is met and the interrupt is triggered, the function is called; this is immediate and does not follow the conventional line-by-line execution order that we are used to following. Here is a ordered list of events that occur to generate an interrupt:

1. The MCU is configured to generate a hardware interrupt at the NVIC controller (from the supported list).
2. The hardware peripheral is configured to generate the interrupt signal based on a condition.
3. The condition for the interrupt is met.
4. The hardware acknowledges that the condition is met by setting a bit in a status register (called a *flag*) and signals the interrupt controller to trigger an interrupt (Interrupt Request (IRQ)).
5. The NVIC controller stores the current address of the line that the software is at in the program.
6. The MCU jumps to the interrupt subroutine function that is specified in the *interrupt vector table*.
7. Before the function finishes, the user must clear the flag/(status register) generated by the interrupt (usually done in software).
8. When the function finishes, the program returns to the next line of code to be ran (from the stored address above).

For example, a timer can be configured to count down to zero. In the configuration for the timer, a hardware interrupt can be enabled when the count for the timer reaches zero (this is the condition for the interrupt). From this, the function specified in the interrupt vector table is immediately called when the timer reaches zero.

Every MCU supports different hardware interrupts. Some MCUs allow for external interrupts to trigger from any GPIO PIN, while other MCUs have select dedicated pins that are allowed to trigger an interrupt. Similarly, most peripherals can be configured to generate an interrupt, such as timers, ADCs, communication peripherals, etc. Every MCU has a limited amount of interrupts available. The TM4C123GH6PM has 78 different interrupts that can be configured and enabled. See page 103-106 for a list of supported interrupts. These interrupts are divided into two categores: exception interrupts and peripheral-based interrupts.

Exception interrupts give the user a chance to recover from an error when it occurs. For example, what should the MCU do when it attempts to divide by zero? Some MCUs will ignore this error. Others will reset the MCU. In either case, if an interrupt is enabled on this error, the error can be dealt with by the code written in the function that is called when the interrupt is triggered.

Peripheral interrupts will trigger when the condition is met; this will vary depending on the peripheral. A GPIO pin can trigger an interrupt every time the level changes from low to high, or from high to low. They can be configured to trigger the interrupt anytime the pin is high, or any time the pin is low. All of these options are specified by the configuration registers. When the condition is met, a function is called by the hardware in the MCU - but which one?

When your project is created in CCS, a file called **tm4c123gh6pm_startup_ccs.c** is created. Look at that file, and see that some things are automatically created for you. Here is a breakdown of that file:

Example 2.1 (*tm4c123gh6pm_startup_ccs.c Explanation*)

```
//*****************************************************************************
//
// Forward declaration of the default fault handlers.
//
//*****************************************************************************
void ResetISR(void);
static void NmiSR(void);
static void FaultISR(void);
static void IntDefaultHandler(void);
```

Example 2.1 shows declarations of any functions that are defined *in the current file*. This is useful for exception handlers which are already defined at the bottom of this file.

Example 2.2 (*tm4c123gh6pm_startup_ccs.c Explanation*)

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

//*****************************************************************************
//
// Linker variable that marks the top of the stack.
//
//*****************************************************************************
extern uint32_t __STACK_TOP;

//*****************************************************************************
//
// External declarations for the interrupt handlers used by the application.
//
//*****************************************************************************
// To be added by user
```

Example 2.2 shows function declarations for functions that are *defined in another source file*. In C, the **extern** keyword tells the compiler that the definition for that function/variable is located in another file. This is useful for interrupts for our peripherals. For example, the function that we want called when the timer reaches zero might be defined in our main source file (main.c). We also need to reference that function in this file (tm4c123gh6pm_startup_ccs.c). The **extern** keyword allows us to do that: declare a function for use in the current file that is defined in another source file.

Example 2.3 (*tm4c123gh6pm_startup_ccs.c Explanation*)

```
//*****************************************************************************
//
// The vector table. Note that the proper constructs must be placed on this to
// ensure that it ends up at physical address 0x0000.0000 or at the start of
// the program if located at a start address other than 0.
//
//*****************************************************************************
#pragma DATA_SECTION(g_pfnVectors, ".intvecs")
void (* const g_pfnVectors[])(void) =
{
    (void (*)(void))((uint32_t)&__STACK_TOP),
                                        // The initial stack pointer
    ResetISR, // The reset handler
    NmiSR, // The NMI handler
    FaultISR, // The hard fault handler
    IntDefaultHandler, // The MPU fault handler
    IntDefaultHandler, // The bus fault handler
    IntDefaultHandler, // The usage fault handler
    0, // Reserved
    0, // Reserved
    0, // Reserved
    0, // Reserved
    IntDefaultHandler, // SVCall handler
    IntDefaultHandler, // Debug monitor handler
    0, // Reserved
    IntDefaultHandler, // The PendSV handler
    IntDefaultHandler, // The SysTick handler
    IntDefaultHandler, // GPIO Port A
    IntDefaultHandler, // GPIO Port B
            .
            .
            .
};
```

Example 2.4 Shows the interrupt vector table. This is the list of possible interrupts, and to which function the interrupt should call when the interrupt condition is met. In the current state of this file, most of the interrupts are told to jump to a function called 'IntDefaultHandler'. This function is defined at the bottom of the file, and contains a single infinite blocking while loop. This is likely not useful to us, so any interrupts that we are using should have a new function defined and the interrupt vector table needs to be adjust to jump to that function. **Note: The order of this table is important. Do not add or remove lines from this table.** Follow the comments to adjust the correct interrupt.

Example 2.4 (*tm4c123gh6pm_startup_ccs.c Explanation*)

```
//*****************************************************************************
//
// This is the code that gets called when the processor receives an unexpected
// interrupt. This simply enters an infinite loop, preserving the system state
// for examination by a debugger.
//
//*****************************************************************************
static void
IntDefaultHandler(void)
{
    //
    // Go into an infinite loop.
```

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

The creation of the interrupt function is no different than any other function that you would write; in fact, you can use any function that you've written. There are no rules or limits on what the name of the function is, and there is no limitations on the content. Having said this, here are the **strict guidelines** that you should follow:

1. Name the function such that it is easily identifiable as an interrupt function. Common keywords that would be included in the name include 'handler', 'ISR' (Interrupt Subroutine), and/or 'interrupt'.

2. The contents of the function need to be kept **short**. This refers to execution time, not amount of code present. Common actions that would be included in an ISR include setting variables, setting flags, or adjusting variables. All of the heavy processing should be done outside of the ISR. The reason? The longer your code stays in an ISR, the more likely that some other interrupt source will attempt to trigger an interrupt (during another interrupt, hence the name 'Nested Vectored Interrupt Controller'). When an interrupts occurs, information is stored into special memory called the 'stack'. This is a small allocation of memory, and if too many interrupts are called before finishing, the stack with overflow and cause the MCU to crash. **Never put blocking code in an ISR.**

3. Remember to clear the interrupt flag in the status register at the end of the interrupt. Forgetting to do this can cause erratic behavior - some MCUs will automatically retrigger the ISR; others will not allow for the interrupt to ever happen again. **Note: SysTick is an exception to this:** SysTick will automatically clear the ISR flag once the function has finished. If you are following examples using SysTick, you may notice that the ISR function never clears the ISR flag - don't get confused by this, as it's done automatically for **SysTick only**.

When two interrupts are called at the same time, the MCU needs to know which one to prioritize. To assist with this, you can specify the priority of an ISR using the interrupt priority registers outlined on page 152. Each interrupt can be assigned a value from 0 to 7. Lower values are given higher priority. If two interrupts are have the same priority, the first one triggered takes precedence.

3 Configuring the Nested Vector Interrupt Controller (NVIC)

In order for the interrupt controller to actually trigger the interrupt, we need to enable that functionality in the NVIC. Each interrupt is given a number in the NVIC as shown on page 104.

From the chart we can see that TIMER0A is assigned the vector number 35 and the interrupt number 19. We need this information to set the correct bit in the NVIC registers on page 141/142. Thus, to enable interrupts in the NVIC for TIMER0A, we would have to set bit 19 in the NVIC_SET_EN0 register.

Related to enabling the NVIC, we also should assign a priority to the interrupt. The interrupt priority is used to establish a prioritized order for when an interrupt is called from within another interrupt; should the new interrupt take priority or should the existing one finish before starting the new one? Interrupt priority registers are discussed on page 152. Note that each interrupt contains a 4-bit field to assign a priority value (from 0 to 7). Note: Lower values have higher priority.

4 Configuring Interrupts for Timer 0

Page 722 outlines the procedure for initializing TIMER 0 for periodic mode. Steps 6 and 7 outline the additional steps needed to configure the interrupts for TIMER 0. The actual condition that allows the timer to trigger the interrupt can be selected in the IMR (Interrupt Mask) register. A common point of confusion is related to the differences between Bit 4 (Interrupt on Match) and Bit 0 (Interrupt on Overflow). The difference is related to the intention of the user. Do you intend to have the interrpt trigger when it reaches a certain value, but continue counting past it? If you don't want it to continue past the matched value and you want the interrupt to trigger on overflow (which is based on the value in the ILR register); otherwise, you would set the interrupt to trigger on a match to the value assigned the corresponding

MATCH register (but it will still continue to count to the ILR register before overflowing).

Note that when using the interrupts, RIS (Raw Interrupt Status) will indicate if the timer has overflowed, but MIS (Masked Interrupt Status) will indicate if that interrupt status bit is actually sent to the interrupt controller (IRQ sent to NVIC). This implies that RIS will be set on overflow even if interrupts are not used (which can be useful in our code..), but MIS will be only be set when interrupts are actually enabled via the IMR.

5 Configuring Interrupts for GPIO PORT F

(Optional)

Similar to TIMER0, you will need to enable the corresponding bit in the NVIC using the NVIC_SET_ENn Registers and optional priority registers. Then you will need to follow the instructions on page: 654 for enabling interrupts for the port and configuring the interrupts. Some common triggers include positive edge, negative edge, or level sensitive.

Also similar to TIMER0, PORTF offers both an RIS and an MIS. As the name implies, the GPIOMIS register only shows interrupt conditions that are allowed to be passed to the interrupt controller. The GPIORIS register indicates that a GPIO pin meets the conditions for an interrupt, but has not necessarily been sent to the interrupt controller.

6 7-Segment Displays

As you should know by now, the micrcontroller is great at controlling devices, but does a poor job of actually doing the work. Each 7-segment display consists of 8 LEDs. Thus, 4 displays equates to 32 LEDs. This poses two problems: first, that is too many GPIO pins to use if we had a dedicated LED per GPIO pin. Second, that's a lot of current to sink or source, so we'll need something that provides the driving capabilities required for the 7-segment display.

Let's look at the schematic for the board that contains the 7-segment displays:

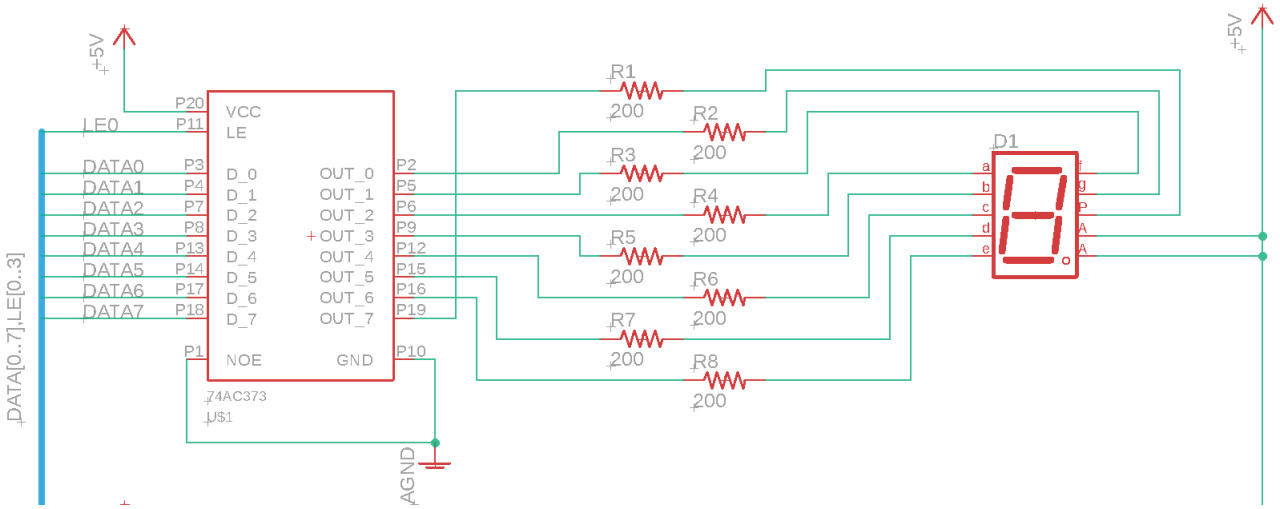
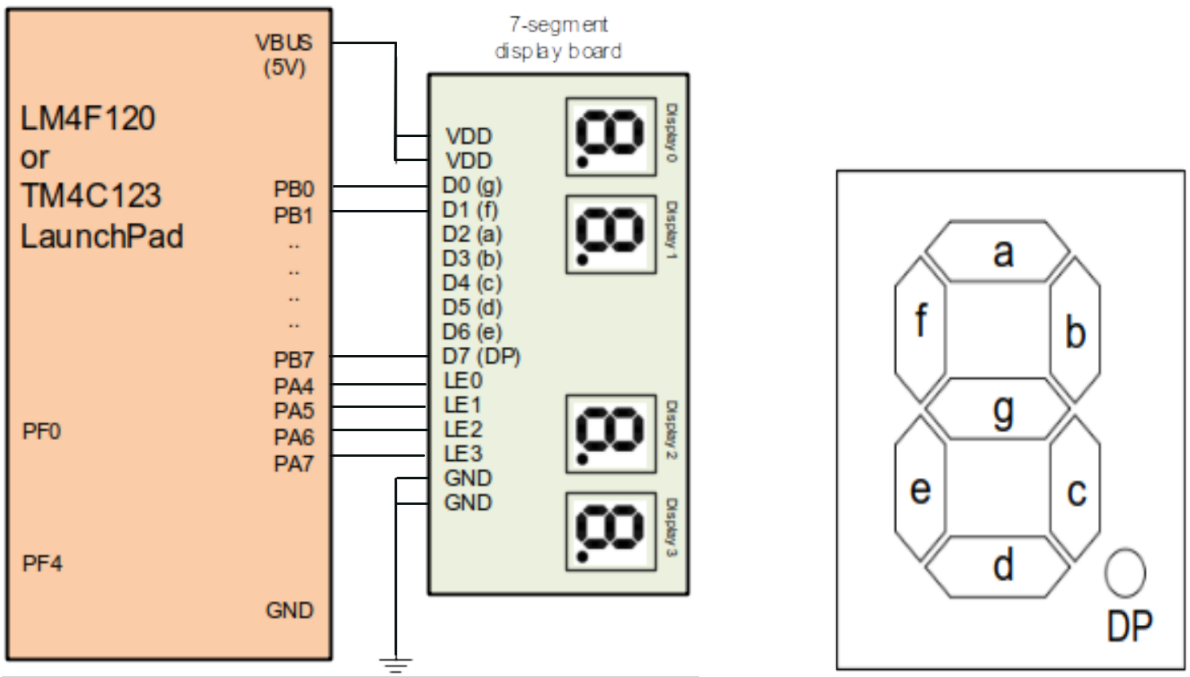


Figure 1: Schematic of the 7-Segment Display Board

From this, you can see that each LED is driven from an 74AC373 latch circuit. The input to the latch circuit is then driven by the MCU with the data driven from PORTB and the latch enable is driven from PORTA.

The procedure to use the Seven Segment Board is as follows:

1. Configure Port A and Port B to be outputs to drive the 7-segment display board.
2. Set the data port to the desired 7-segment value for one of the displays.



(a) Connections from the Launchpad to the 7-Segment Board

(b) 7-Segment Connection Map

Figure 2: Using the 7-Segment Display Board

3. Pulse the latch enable for that display. At 16Mhz, one clock cycle is long enough..
4. Set the data port for the next 7-segment display.
5. Pulse the latch enable for that display.
6. Repeat as necessary.

7 Procedure

1. Create a new project called NSID_Lab_3. Refer to Lab 1 if you need help.
2. Initialize the tri-color LED and the two switches.
3. Initialize Timer0 to count up in 32 bit mode as we did in Lab 2. Make the following adjustments (see Appendix A for more details):
   (a) Enable interrupts in the NVIC controller for Timer 0.
   (b) Create an interrupt subroutine (function definition) for Timer 0 in your main.c file.
   (c) Modify the Startup_CCS.C file to include the function declaration (as an extern declaration)
   (d) Modify the vector table in the Startup_CCS.C file to point Timer0A's interrupt to the defined ISR function.
   (e) Set the timer to overflow/count up to a value that corresponds with a 500ms timer (adjust the TAILR register)
   (f) Enable the bit in Timer0's IMR register to enable interrupts on TIMER A Overflow.
4. Inside the ISR function, toggle the Red LED and *remember to clear the interrupt flag!*
5. Verify your timer with interrupts is working. *If done correctly, your while(1) loop will be empty, but the red LED will be flashing at a rate of 1Hz. You can also verify the interrupt is triggering properly by setting a breakpoint in the ISR function and verifying that the microcontroller is calling the function.*
6. Initialize PORTB and PORTA for use with the 7-segment display.
7. Test the 7-segment displays by writing the value '2431' to it.
8. (Optional) Adjust your solution to include interrupts for the switches on PORTF. Create an ISR function for the switches.
9. (Optional) Adjust your switches to trigger interrupts on the negative edge (get this working without worrying about debouncing first). Update the vector table to include a new function for the PORTF button ISR.
10. Debounce your switches using any method you are familiar with. One possible option is to introduce using another timer.

Exercise 7.1 (*Using the 7-segment displays*) *

1. Write a program that does the following (one program that does all of the following):
   Note: Use of at least one interrupt is required.
   (a) Flash the LED at a rate of approximately **one** hertz.
   (b) Each time SW1 is pressed, change the color of the LED in the following order: (Red->Blue->Green repeat)
   (c) Make a counter on PORTB that counts at a rate of **one** hertz. Write the value of this counter to the 7-segment displays.
   (d) Make SW2 a Start/Stop button. Each time it's pressed, stop counting on PORTB. Press again to resume counting.
2. Make a PDF that has any information that will benefit the marker in regards to the coding question, and answers to the following questions:
   (a) i. How much current can the latch drive?
   ii. Does this current come from the microcontroller?
   iii. How much current does the 8 segments of a 7-segment display draw? (Assume the LED Forward Voltage Drop is 2.6V)
   (b) How long does it take to call the function below (in clock cycles):

```
float square ( float x ) {
    float p ;
    p = x * x ;
    return ( p ) ;
}
```

Hint: Use a hardware based timer and the debugger

(c) The FaultISR() function is already defined in the file **tm4c123gh6pm_startup_ccs.c**. What is the purpose of this function and for what conditions does it get triggered?

3. Go back to the start of this manual and verify that you have met the objectives of the lab.

8 Hand-in Instructions

Make a folder called 'Documents' inside of the folder that is your solution to Exercise 7.1 (that should be named NSID_Lab_3). Place your PDF with the answers to your questions in the Documents Folder. Zip the entire contents of the folder named NSID_Lab_3, and hand-in the zip file to canvas/blackboard before the due date. The due date is one week from the day of the lab (at midnight). See the note on the first page about working with a partner.

A Appendix: Summary of Steps to Configure Interrupts

For each interrupt that you want to enable, you need to follow these steps:

1. Configure the vector table. That's done in the startup.ccs file. See the lab manual for an explanation. In particular, you'll need to:

   a Add extern function declaration (if defining the function in a different file)

   b Add the NAME ONLY of the function in the vector table

   c Add the definition for the function. To keep it simple, you'll likely want to do this in main.c (and it's a really good idea to indicate to yourself it is an interrupt function by including ISR, handler, or some other such indicator).

2. Configure the NVIC Controller. This allows the code to jump to the ISR when the interrupt condition has been met. Each peripheral needs their interrupt enabled in the NVIC Controller. Since there are more than 70 different interrupts, the enables are spread across many different registers. Here are list of the symbol definitions that you would need to include all interrupts:

   ```
   //Symbol Definitions that we'll need to configure the NVIC interrupt controller
   #define NVIC_EN0_R (*((volatile unsigned long *)0xE000E100))
   #define NVIC_EN1_R (*((volatile unsigned long *)0xE000E104))
   #define NVIC_EN2_R (*((volatile unsigned long *)0xE000E108))
   #define NVIC_EN3_R (*((volatile unsigned long *)0xE000E10C))
   #define NVIC_EN4_R (*((volatile unsigned long *)0xE000E110))
   ```

   In our initialization of our microcontroller, we'll need to set the correct bit:

   ```
   //Enable the NVIC register to get interrupts to jump to vector.
   //Each bit represents a different peripheral/interrupt.. Page 104, 142
   NVIC_EN0_R |= 0x00080000; //Timer 0A is interrupt vector 35
                          //and interrupt number 19 (bit in interrupt registers)
   ```

3. Enable interrupts for the peripheral in the configuration registers of the peripheral. For the GPIO timers, the interrupts are enabled by setting the condition for interrupt int the IMR register. For this lab, it makes sense to trigger an interrupt on timeout, so look at bit 0.

   For the GPIO pins (such as the pin connected to the switch..), you can enable interrupts by looking at the interrupt mask register (GPIOIM - page 667).

4. Clear the ISR flag prior to enabling the peripheral. Also clear the ISR flag at the end of your ISR function. There are built-in registers that the datasheet asks us to use to clear the ISR flags.

   Here is how you clear the interrupt status flag for TIMER0 using the built-in function:

   ```
   TIMER0_ICR_R |= 0x01; //clears the timeout interrupt flag. See page 754
   ```

Troubleshooting:
When using the debugger, there are several registers to look at to debug/verify your code:

1. Verify the TIMER0_TAV register is changing values and counting the correct direction.

2. The TIMER0_RIS register tells you the condition has been met. For example, bit 0 should go high on TIMER0 overflow. This tells you that the timer value got to the TAILR value (count up mode).

3. The TIMER0_MIS register tells you if the interrupt request has been sent to NVIC.. this register only gets set when you configure the IMR register correctly.

4. Try setting a breakpoint in the ISR function. See that the code reaches the ISR. This will verify that the vector table is configured properly and the the NVIC register is enabled.