Department of Electrical and Computer Engineering

ENCS4110 | Computer Design LAB

Experiment No. 8: Timers Interrupts

Objectives

The main objective of this experiment is to introduce students to the internal timers/counters and to learn how to setup nested vector interrupt controller (NVIC). Students will learn how to configure the timers to make internal interrupt.

In this experiment, we will discuss TM4C123 Timer interrupt programming on ARM Cortex M4 microcontrollers. Firstly, we will discuss applications of timer interrupt with one example. We will use a TM4C123 Tiva C LaunchPad for demonstration purposes. It comes with a TM4C123GH6PM microcontroller. The general-purpose timer module TimerA module will be configured, and initialized to generate a 1s periodic interrupt. That means an interrupt handler routine of the TimerA module will execute after every one second. We can perform the operations that we want to perform inside the exception handler function.

General Purpose Timer Interrupt

Programmable general-purpose timer modules (GPTM) of TM4C123 microcontroller can be used to count external events as a counter or as a timer. For example, we want to measure an analog signal with the ADC of TM4C123 microcontroller after every one second. By using GPTM, we can easily achieve this functionality. In order to do so, first configure the timer interrupt of TM4C123 to generate an interrupt after every one second. Second, inside the interrupt service routine of the timer, sample the analog signal value with ADC and turn off ADC sampling before returning from the interrupt service routine. Similarly, general-purpose timer modules have many applications in embedded systems.

Applications

Few important applications are:

- External event measurement (Example HC-SR04 with TM4C123)
- Pulse width measurement
- Frequency measurement
- Motor RPM Measurement TM4C123

How to configure Timer Interrupt of TM4C123

TM4C123 microcontroller provides two timer blocks such as Timer A and Timer B. Each block has six 16/32 bits GPTM and six 32/64 bits GPTM. We will use the TimerA block in this experiment.

Procedure

First, create a project in Keil uvision by selecting TM4C123GH6PM microcontroller

Include the header file of TM4C123GH6PM microcontroller, which contains a register definition file of all peripherals such as timers.

#include "TM4C123.h" // Device header file for Tiva Series Microcontroller

Timer1A Interrupt with One Second Delay

In this section, we create a timer interrupt of 1Hz. That means the interrupt will occur after every one second. Because the time period is inverse of the frequency. Whenever an interrupt occurs, we will toggle an LED inside the corresponding interrupt service routine of TimerA module. In short, the interrupt service routine will execute every one second and toggle the onboard LED of TM4C123 Tiva C LaunchPad.

PF2*

TM4C123 Tiva C LaunchPad has an onboard RGB LED. Blue LED is connected with the PF3 pin of PORTF. We will toggle this LED inside the interrupt service routine of TimerA. First, let us define a symbol name for this PF3 pin using #define preprocessor directive.

#define Blue (1<<2) // PF3 pin of TM4C123 Tiva Launchpad, Blue LED

Initialize Timer1A registers for one second delay

RCGCTIMER

First, enable the clock to timer block 1 using RCGTIMER register. Setting 1st bit of RCGTIMER register enables the clock to 16/32 bit general purpose timer module 1 in run mode. This line sets the bit1 of the RCGTIMER register to 1.

```
SYSCTL->RCGCTIMER |= (1<<1);  /*enable clock Timer1 subtimer A in run mode */
```

Before initialization, disable the Timer1 output by clearing GPTMCTL register.

```
TIMER1->CTL = 0; /* disable timer1 output */
```

The first three bits of the GPTMCFG register are used to select the mode of operation of timer blocks either in concatenated mode (32 or 64 bits mode) or individual or split timers (16 or 32 bit mode). We will be using a 16/32-bit timer in 16-bits configuration mode. Hence, writing 0x4 to this register selects the 16-bit configuration mode. (Check datasheet 726 of TM4C123GH6PM)

```
TIMER1->CFG = 0x4; /*select 16-bit configuration option, CFG=0x00 for 32-bit option */
```

Timer modules can be used in three modes such as one-shot, periodic, and capture mode. We want the timer interrupt to occur periodically after a specific time. Therefore, we will use the periodic mode. In order to select the periodic mode of timer1, set the GPTMTAMR register to 0x02 like this (refer to page 732 datasheet):

```
TIMER1->TMAR = 0x02; /*select periodic down counter mode of timer1, TMAR=0x01 for one-shot mode */
```

We are using timer1 in 16-bit configuration. The maximum delay a 16-bit timer can generate according to 16MHz operating frequency of TM4C123 microcontroller is given by this equation:

2^16 = 65536

16MHz = 16000000

Maximum delay = 65536/16000000 = 4.096 millisecond

Hence, the maximum delay that we can generate with timer1 in 16-bit configuration is 4.096 millisecond. Now the question is how to increase delay size? There are two ways to increase the timer delay size: either increase the size of timer (32-bit or 64-bit) or use a prescaler value. Because we have already selected the 16-bit timer1A configuration. Therefore, we can use the prescaler option.

Prescaler Configuration

Prescaler adds additional bits to the size of the timer. GPTM TimerA Prescale (GPTMTAPR) register is used to add the required Prescaler value to the timer. TimerA in 16-bit has an 8-bit

Prescaler value. Prescaler basically scales down the frequency to the timer module. Hence, an 8-bit Prescaler can reduce the frequency (16MHz) by 1-255.

For example, we want to generate a one-second delay, using a Prescaler value of 250 scales down the operating frequency for TimerA block to 64000Hz .i.e. 16000000/250 = 64000Hz = 64KHz. Hence, the time period for TimerA is 1/64000 = 15.625ms. Hence, load the Prescaler register with a value of 250.

```
TIMER1->TAPR = 250-1; /* TimerA prescaler value */
```

When the timer is used in periodic countdown mode, the GPTM TimerA Interval Load (GPTMTAILR) register is used to load the starting value of the counter to the timer.

```
TIMER1->TAILR = 64000 - 1 /* TimerA counter starting count down value */
```

GPTMICR register is used to clear the timeout flag bit of timer.

```
TIMER1->ICR = 0x1; /* TimerA timeout flag bit clears*/
```

After initialization and configuration, enables the Timera module, which we disabled earlier.

```
TIMER1->CTL |= (1<<0);  /* Enable TimerA module */
```

Configure TM4C123 Timer Interrupt

There are two steps to enable the interrupt of peripherals in ARM Cortex-M microcontrollers. Firstly, enable the interrupt from the peripheral level (Timer module in this case) using their interrupt mask register. GPTM Interrupt Mask (GPTMIMR) register enables or disables the interrupt for the general-purpose timer module of TM4C123 microcontroller. Bit0 of GPTMIMR enables the TimerA time-out interrupt mask.

```
TIMER1->IMR |=(1<<0); /*enables TimerA time-out  interrupt mask */
```

Secondly, we must also enable the GPTM interrupt request to a nested vectored interrupt controller using their interrupt enable registers. Interrupt request number of Timer1A is 21 or IRQ21. Hence, it corresponds to NVIC interrupt enable register zero. This line enables the Timer1A to interrupt from NVIC level by setting bit21:

```
NVIC->ISER[0] |= (1<<21); /*enable IRQ21 */
```

You can also find the interrupt number of every exception handler or ISR inside the startup file of TM4C123 microcontroller.

Implementation of Timer Interrupt Handler function

ISRs (Interrupt Service Routines) of all peripheral interrupts and exception routines are defined inside the startup file of TM4C123 Tiva microcontroller and the interrupt vector table is used to

relocate the starting address of each interrupt function. In order to find the name of the handler function of Timer1 and sub-timer A, open the startup file and you will find that the name of the Timer1A handler routine is TIMER1A_Handler(). By using this name of the Timer1A interrupt handler, we can write a c function inside the main code like this:

```
TIMER1A_Handler()

{

// instructions you want to implement

}
```

Types of Interrupt and Exceptions in ARM Cortex-M

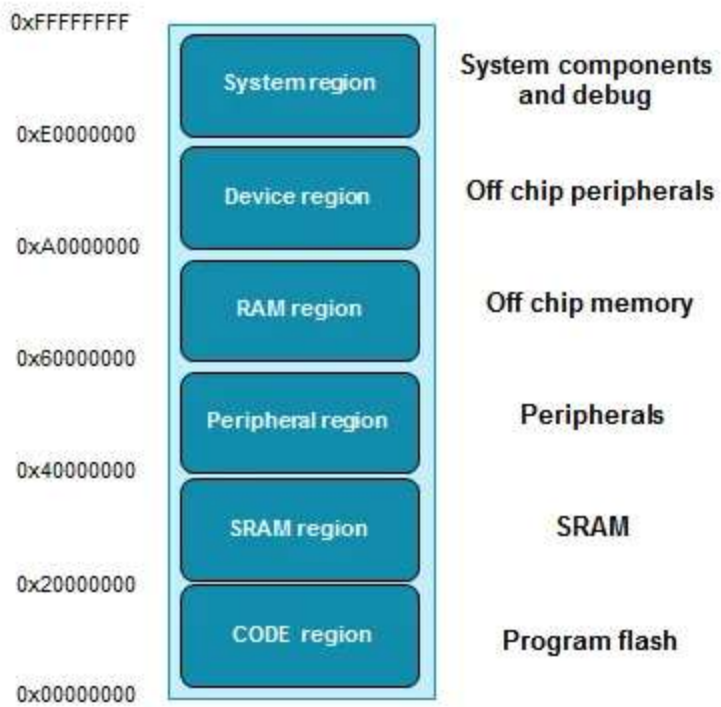
Throughout this experiment, we will use exception and interrupt terms interchangeably. Because, in ARM Cortex-M literature both terms are used to refer to interrupts and exceptions. Although there is a minor difference between interrupt and exception. Interrupts are special types of exceptions which are caused by peripherals or external interrupts, such as, Timers, GPIO, UART, I2C, etc. On the contrary, exceptions are generated by processor or system. For example, In ARM Cortex-M4, the exceptions numbered from 0-15 are known as system exceptions and the peripheral interrupts can be between 1 to 240. However, the available number of peripheral interrupts can be different for different ARM chip manufacturers. For instance, ARM Cortex-M4 based TM4C123 microcontroller supports 16 system exceptions and 78 peripheral interrupts.

What happens when an interrupt occurs?

Whenever an exception or interrupt occurs, the processor uses an interrupt service routine for interrupts and an exception handler for system exceptions. In other words, whenever an interrupt occurs from a particular source, the processor executes a corresponding piece of code (function) or interrupt service routine.

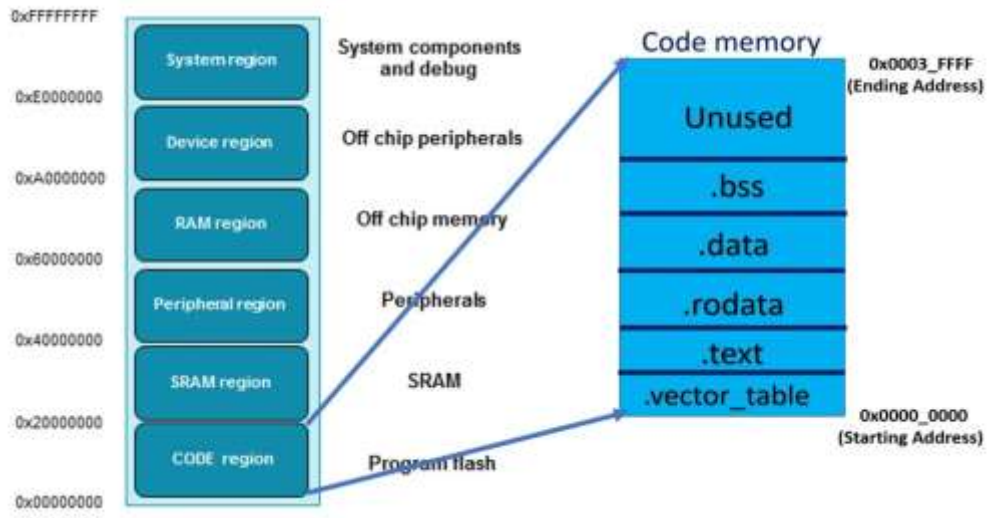
Interrupt Vector Table and Interrupt Processing

However, the question is how does the processor find the addresses of these interrupt service routines or exception handlers? In order to understand this, let us take a review of the memory map of ARM cortex M4 microcontrollers. The address space that ARM MCU supports is 4GB. This memory map is divided into different memory sections, such as, code, RAM, peripheral, external memory, system memory region, as shown in the figure below:

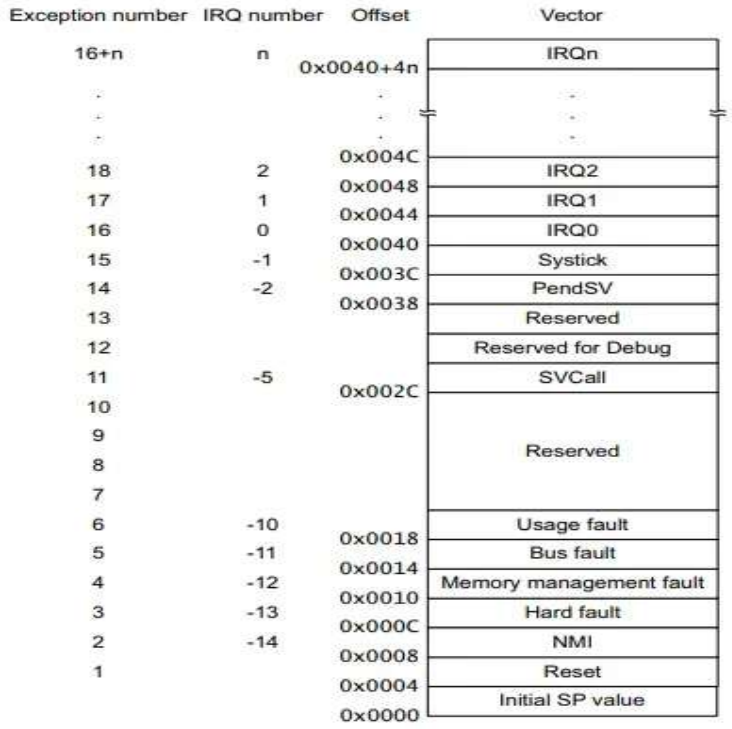


Relocating ISR Address from IVT

Coming back to the main discussion, the code memory region contains an interrupt vector table (IVT). This interrupt vector table consists of a reset address of stack pointer and starting addresses of all exception handlers. In other words, it contains the starting address that points to the respective interrupt and exception routines, such as, reset handler. Hence, the microcontroller uses this starting address to find the code of the interrupt service routine that needs to be executed in response to a particular interrupt.



The figure above shows the interrupt vector table of the ARM Cortex-M4 microcontroller. As you can see, the interrupt vector table is an array of memory addresses. It contains the starting address of all exception handlers. Furthermore, each interrupt/exception also has a unique interrupt number assigned to it. The microcontroller uses interrupt number to find the entry inside the interrupt vector table. The following table shows the interrupt vector table of ARM Cortex M4 based TM4C123GH6PM microcontroller.



Examples

For example, when an interrupt x occurs, the nested vectored interrupt controller uses this interrupt number to find the memory address of the interrupt service routine inside the IVT. After finding the starting address of the exception handler, NVIC sends the request to the ARM microcontroller to start executing the interrupt service routine.

Let us take an example, let us suppose, interrupt number 2 occurs. The NVIC used this formula to find the entry of corresponding IRQx in IVT:

*Entry in IVT = 64 +4 * x;*

for x=2

*Entry in IVT = 64 +4 * 2 = 72*

= 72 or 0x0048

As you can see from the above interrupt vector table, the address of IRQ2 is 0X0048. NVIC will get the starting address of IRQ2 from the memory location 0x0048 and sets the program counter to the starting address of IRQ2. After that processor jumps to this locations to execute ISR.

One important point to note here is that the value of the interrupt number is negative, as well. The interrupt number or IRQn of all system exceptions is in negative. This example shows the relocation of entry in IVT when the bus fault exception occurs:

*Entry in IVT = 64 +4 * x ;*

For bus fault exception x = -11

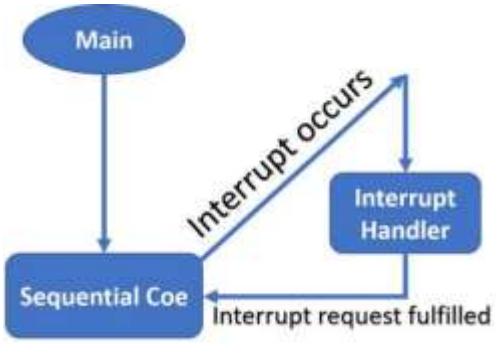
*Entry in IVT = 64 +4 * -11= 20*

= 20 or 0x0014

Steps Executed by ARM Cortex M processor During Interrupt Processing

Until now, we have discussed how ARM microcontroller finds the address of the first instruction of the corresponding interrupt service routine. In this section, we will see the sequence of steps that occurs during interrupt processing, such as, context switching, context saving, registers stacking and unstacking.

Whenever an interrupt occurs, the context switch happens. That means the processor moves from thread mode to the handler mode. As shown in the figure below, the ARM Cortex-M microcontroller keeps executing the main application, but when an interrupt occurs, the processor switches to interrupt service routine. After executing ISR, it switches back to the main application again. Nevertheless, what happens during context switching requires more explanation.



For example, the ARM cortex-M processor executing these instructions inside the main code:

1. LDR R2 [R0] // load value at address [R0] in register R2

2. LDR R1 [R0+4] // load value at address [R0+4] in register R1

3. ADD R3 R2 R1 // add R2+R1 and save the content in R3

4. STR R3 [R0] // Store R3 at address [R0]

The main application is executing the above instructions and the program counter is pointing to instruction 3. At the same time, an interrupt signal arrives.

ARM Cortex-M Context Switching

When an interrupt occurs and its request is approved by the NVIC and processor, the contents of the CPU registers are saved onto the stack. This way of CPU registers preservation helps microcontrollers to resume the interrupted program form the same instruction where it is suspended due to an interrupt service routine. The process of saving the main application registers content onto the stack is known as context switching.

However, it takes time for the microcontroller to save the CPU register's content onto the stack. However, to Harvard architecture of ARM processors (separate instruction and data buses), the processor performs context saving and fetching of starting address of interrupt service routine in parallel.

Interrupts Processing Steps

The microcontroller will perform the following steps:

Stacking

1- First ARM Cortex-M finishes or terminates the instruction under execution, e.g., the instruction number 3 (ADD R3 R2 R1) in the above example. The condition of finishing or terminating the current instruction depends on the value of the interrupt continuable instruction (ICI) field in the xPSR register.

2- After that, it suspend the current main application and save the context of the main application code into the stack. Microcontroller pushes registers R0, R1, R2, R3, R12, LR, Program counter and program status register (PSR) onto the stack.

The order in which stacking take place is PSR, PC, LR, R12, R3, R2, R1 and R0.

Exception Entry

After that, the ARM processor reads the interrupt number from the xPSR register. By using this interrupt number processor finds the entry of the exception handler in the interrupt vector table. Finally, it reads the starting address of the exception handler from the respective entry of IVT.

Now ARM processor updates the values of the stack pointer, linker register (LR), PC (program counter) with new values according to the interrupt service routine. The link register contains the type of interrupt return address.

After that, the interrupt service routine starts to execute and finish its execution.

Exception Return

The last step is to return to the main application code or to exist from the interrupt service routine. To return from ISR, the processor loads the link register (LR) with a special value. The most significant 24-bits of this value are set to 0xFFFFFF and the least significant eight bits provide different ways to return from exception mode. For example, if the least significant 8-bits are F9. The processor will return from exception mode by popping all eight registers from the stack. In addition, it will return to thread mode using MSP as its stack pointer value.

The least significant 8-bits of the value which is loaded to LR register during the exception return process determine which mode to return either remain in exception mode (in case of nested interrupts) or handler mode.

After that, the microcontroller starts executing the main application from the same instruction where it is pre-empted by interrupt service routine.

TM4C123 Timer Interrupt Example Code

This example code of TM4C123 Tiva C Launchpad generates a delay of one second using the Timer1A interrupt handler routine. Inside the main code, we initialize the PF2 pin as a digital output pin

```
/* This is a timer interrupt example code of TM4C123 Tiva C Launchpad */
/* Generates a delay of one second using Timer1A interrupt handler routine */

#include "TM4C123.h"   // Device header file for Tiva Series Microcontroller
#define Blue (1<<2)   // PF3 pin of TM4C123 Tiva Launchpad, Blue LED
void Time1A_1sec_delay(void);
int main(void)
{
        /*Initialize PF3 as a digital output pin */
  SYSCTL->RCGCGPIO |= 0x20; // turn on bus clock for GPIOF
  GPIOF->DIR       |= Blue; // set blue pin as a digital output pin
  GPIOF->DEN       |= Blue;  // Enable PF2 pin as a digital pin
  Time1A_1sec_delay();
        while(1)
        {
                // do nothing wait for the interrupt to occur
        }
}

/* Timer1 subtimer A interrupt service routine */
```

```
TIMER1A_Handler()
{
        if(TIMER1->MIS & 0x1)
                GPIOF->DATA  ^= Blue; /* toggle Blue LED*/
                TIMER1->ICR = 0x1;        /* Timer1A timeout flag bit clears*/
}
void Time1A_1sec_delay(void)
{
SYSCTL->RCGCTIMER |= (1<<1);  /*enable clock Timer1 subtimer A in run mode */
TIMER1->CTL = 0; /* disable timer1 output */
TIMER1->CFG = 0x4; /*select 16-bit configuration option */
TIMER1->TAMR = 0x02; /*select periodic down counter mode of timer1 */
TIMER1->TAPR = 250-1; /* TimerA prescaler value */
TIMER1->TAILR = 64000-1 ; /* TimerA counter starting count down value  */
TIMER1->ICR = 0x1;        /* TimerA timeout flag bit clears*/
TIMER1->IMR |=(1<<0); /*enables TimerA time-out  interrupt mask */
TIMER1->CTL |= 0x01;       /* Enable TimerA module */
NVIC->ISER[0] |= (1<<21); /*enable IRQ21 */
}
```

In-Lab Task:

1. Modify the code above to make the GREEN LED blinks every **500ms**.
2. Modify the code above to make the RED LED blinks every **4s**.
3. Use the onboard LED and another two external LEDs with the TM4C123G board to make one LED flashes every 10 seconds, one flashes every 5 seconds, and one flashes every one second.