ECE/CSE 474:
Intro to Embedded Systems

Lab 2: General-Purpose Timers and Interrupts

Introduction

In the first lab, you delayed CPU operation by occupying its attention with non-useful tasks. This, however, puts an undue burden on your CPU and makes your program quite inefficient. In this lab, you will be modifying your work from Lab 1 to implement timers and interrupts. Timers and interrupts are effective alternatives to delaying system operation, reducing the amount of unnecessary operations the CPU is forced to execute.

By the end of this lab, you will:

- **Have a basic understanding of and experience using general-purpose timers and interrupts.**
- **Be familiar with the TM4C1294NCPDT Vector Table and know how to modify it.**
- **Understand how interrupts can be used and how to write Interrupt Service Routines.**
- **Become even more competent at extracting data from the TM4C's datasheet.**

Task 1: General-Purpose Timers

Timers

The TM4C1294NCPDT General-Purpose Timer Module (GPTM) provides programmable timers that can be used to count or to drive events such as I/O, communication, or analog-to-digital conversions. The GPTM is divided into eight 16/32-bit GPTM blocks. Each GPTM block provides two timers (referred to as Timer A and Timer B) that can be configured to operate independently, or they can be configured to be concatenated. When concatenated, they operate as one 32-bit timer (for the 16/32-bit blocks).

There are also other timers available on the Tiva C Series microcontrollers such as the System Timer (SysTick) and the PWM timer which can be used when required by various applications.

For this lab, you will focus on configuring and using General Purpose Timer 0, Subtimer A. There are various modes that this timer can be configured in; refer to section 13.3.3 of the datasheet for more information. The first step is to use the timer to blink an LED periodically every second, the best configuration for this type of operation is the "periodic timer" mode.

Initialization and Configuration

Section 13.4 of the datasheet details how to initialize the GPTMs. For this section of the lab, you need to perform the following steps. For all of these steps use the 16/32-bit Timer 0:

1. **Enable the appropriate TIMERn bit in the RCGCTIMER register.**
2. **Disable the timer using the GPTMCTL register.**
3. **Write 0x0000.0000 to the GPTMCFG register.**
4. **Select 32-bit mode using the GPTMCFG register.**
5. **Configure the TAMR bit of the GPTMTAMR register to be in periodic timer mode.**
6. **Configure the TACDIR bit of the GPTMTAMR register to count down.**

7. Load the value 16,000,000 into the GPTMTAILR to achieve a 1 Hz blink rate using the 16 MHz oscillator.
8. If using interrupts, configure the GPTMIMR register. This is not necessary for task 1.
9. Enable the timer using the GPTMCTL register.

GPTM Polling

To make use of this timer, poll the TATORIS bit of the GPTMRIS register. This will let you know when the timer has counted down and reached 0. Once this happens, clear the bit by writing to the GPTMICR register. Then, you're able to return to polling to see when the timer has reached 0 again.

Debugging

It's important that you use the features offered by the IAR debugger to debug your programs. Here is a video tutorial that should help you: https://youtu.be/e4R2BRpKjC8

Task 1 Assignments

a. Turn off and on the user LEDs in a periodic pattern.
   Repeat the assigned task from lab 1 where you were asked to turn off and on the onboard LEDs in a periodic pattern. Your sequence should turn LEDs off or on at a rate of 1 Hz by polling a GPTM.
b. Timed Traffic Light
   Repeat the traffic light assignment from lab 1. Use timers to make the following modifications:
   - Each time a button is pressed, the system responds if and only if the user holds the button for at least two seconds. The system should follow this rule to process all button presses.
   - Upon reset, the system should be off (i.e., all LEDs should be turned off). When the On/Off button is pressed, the system will be turned on and start in the *stop* state. After a delay of 5 seconds, the system will move to the *go* state. After another 5 seconds, the system will return to the *stop* state. Until a button is pressed, this should repeat indefinitely. If the On/Off button is pressed again, the system will be turned off, which means that all LEDs will be turned off.
   - After the Pedestrian button is pressed and held by the user for 2 seconds while the system is in the *go* state, the system should immediately transition to the *warn* state, and remain there for 5 seconds before moving to the *stop* state.

Important note: You should restructure your program into functions and sub-functions. Functions should be as short as possible and complete smaller tasks, keeping code in your main function to a minimum. A suggestion is to create a function to initialize the GPIO and another to initialize the timer. Both functions could then be called from the main function. Another function that might be helpful is one to turn off and on LEDs. This may seem bothersome, but it is a smart coding practice. It will be helpful when your lab requirements get more complicated.

Task 2: Interrupts

The Vector Table

The vector table is a table of pointers to routines that are written to handle interrupts. The vector table for the ARM Cortex chip found on the TM4C is the main focus of this section. There is a standard default vector table that is included each time you compile and upload your code onto the board, but you can also create and modify your own.

The vector table of the TM4C1294 LaunchPad is found in Figure 2-6 of the datasheet (page 119). The vector table contains the numbers and addresses of each interrupt. The vector table is located at address 0 by default (it can be moved, but that's not the goal here). When you use the debugger in IAR, you can see the vector table contents in the disassembly, from address 0x0 to 0x3c, shown in Figure 1.

Disassembly

```
          0x0: 0x00          DC8       0                       ; '.'
Region$$Table$$Base:
Region$$Table$$Limit:
          0x1: 0x10          DC8       16                      ; '.'
          0x2: 0x2000        DC16      8192                    ; '. '
          0x4: 0x00000219    DC32      __iar_program_start
          0x8: 0x000001cb    DC32      NMI_Handler
          0xc: 0x00000203    DC32      HardFault_Handler
         0x10: 0x00000229    DC32      MemManage_Handler
         0x14: 0x0000022b    DC32      BusFault_Handler
         0x18: 0x0000022d    DC32      UsageFault_Handler
         0x1c: 0x00000000    DC32      0x0 (0)
         0x20: 0x00000000    DC32      0x0 (0)
         0x24: 0x00000000    DC32      0x0 (0)
         0x28: 0x00000000    DC32      0x0 (0)
         0x2c: 0x0000022f    DC32      SVC_Handler
         0x30: 0x00000231    DC32      DebugMon_Handler
         0x34: 0x00000000    DC32      0x0 (0)
         0x38: 0x00000233    DC32      PendSV_Handler
         0x3c: 0x00000235    DC32      SysTick_Handler
```

Figure 1: The vector table of the TM4C1294 shown in the IAR debugger disassembly

The vector table is implemented in a file named cstartup_M.c. The default vector table is located in the IAR installation directory. You will probably find it here:

C:\Program Files (x86)\IAR Systems\Embedded Workbench 8.0\arm\src\lib\thumb

In this file, you'll find the vector table defined as an array:

```
const intvec_elem __vector_table[] =
{
  { .__ptr = __sfe( "CSTACK" ) },
  __iar_program_start,
```

```
  NMI_Handler,
  HardFault_Handler,
  MemManage_Handler,
  BusFault_Handler,
  UsageFault_Handler,
  0,
  0,
  0,
  0,
  SVC_Handler,
  DebugMon_Handler,
  0,
  PendSV_Handler,
  SysTick_Handler

};
```

The order of the array elements matches Figure 2-6 in the datasheet, with the "Reserved" addresses represented as 0s.

The file also contains the implementation of each of the vector table handlers. These are commonly referred to as Interrupt Service Routines (ISRs). The function prototypes for the ISRs are shown as follows:

```
extern void __iar_program_start( void );

extern void NMI_Handler( void );
extern void HardFault_Handler( void );
extern void MemManage_Handler( void );
extern void BusFault_Handler( void );
extern void UsageFault_Handler( void );
extern void SVC_Handler( void );
extern void DebugMon_Handler( void );
extern void PendSV_Handler( void );
extern void SysTick_Handler( void );
```

The objective is to modify the cstartup_M.c file to add Interrupt Requests (IRQ) to the vector table and implement their corresponding ISRs.

Important: Do NOT modify the original cstartup_M.c file. Instead, make a copy of it in the directory you're using for lab 2. Then, add the file to your project in IAR. Note that the file is read-only by default, so you need to remove the read-only status of your copy.

Modifying the vector table

The objective is to create an ISR to service the timer you created in Task 1. Table 2-9 in the datasheet (page 116) lists the interrupts of the TM4C1294NCPDT. Timer 0A is interrupt number 19, corresponding to vector number 35. Therefore, the vector table needs to be appended with the new Timer ISR, which, in the example below, is called Timer0A_Handler:

```
  SysTick_Handler,
```

```
  0,
  0,
  0,
  0,
  0,
  0,
  0,
  0,
  0,
  0,
  0,
  0,
  0,
  0,
  0,
  0,
  0,
  Timer0A_Handler
```

Moreover, a prototype of the ISR needs to be added to the vector table, following the declarations of the other prototypes:

```
extern void Timer0A_Handler( void );
```

And, finally, to implement the ISR:

```
#pragma call_graph_root = "interrupt"
__weak void Timer0A_Handler ( void ) { while (1) {} }
```

Now, with your modified vector table added to your project, download and debug your program to the board. You'll notice in the disassembly window that the vector table has been extended, and the new timer handler is located at the bottom, as seen in Figure 2.

Disassembly

```
          0x0: 0x00          DC8       0                       ; '.'
Region$$Table$$Base:
Region$$Table$$Limit:
          0x1: 0x10          DC8       16                      ; '.'
          0x2: 0x2000        DC16      8192                    ; '. '
          0x4: 0x00000295    DC32      __iar_program_start
          0x8: 0x00000247    DC32      NMI_Handler
          0xc: 0x0000027f    DC32      HardFault_Handler
         0x10: 0x000002a5    DC32      MemManage_Handler
         0x14: 0x000002a7    DC32      BusFault_Handler
         0x18: 0x000002a9    DC32      UsageFault_Handler
         0x1c: 0x00000000    DC32      0x0 (0)
         0x20: 0x00000000    DC32      0x0 (0)
         0x24: 0x00000000    DC32      0x0 (0)
         0x28: 0x00000000    DC32      0x0 (0)
         0x2c: 0x000002ab    DC32      SVC_Handler
         0x30: 0x000002ad    DC32      DebugMon_Handler
         0x34: 0x00000000    DC32      0x0 (0)
         0x38: 0x000002af    DC32      PendSV_Handler
         0x3c: 0x000002b1    DC32      SysTick_Handler
         0x40: 0x00000000    DC32      0x0 (0)
         0x44: 0x00000000    DC32      0x0 (0)
         0x48: 0x00000000    DC32      0x0 (0)
         0x4c: 0x00000000    DC32      0x0 (0)
         0x50: 0x00000000    DC32      0x0 (0)
         0x54: 0x00000000    DC32      0x0 (0)
         0x58: 0x00000000    DC32      0x0 (0)
         0x5c: 0x00000000    DC32      0x0 (0)
         0x60: 0x00000000    DC32      0x0 (0)
         0x64: 0x00000000    DC32      0x0 (0)
         0x68: 0x00000000    DC32      0x0 (0)
         0x6c: 0x00000000    DC32      0x0 (0)
         0x70: 0x00000000    DC32      0x0 (0)
         0x74: 0x00000000    DC32      0x0 (0)
         0x78: 0x00000000    DC32      0x0 (0)
         0x7c: 0x00000000    DC32      0x0 (0)
         0x80: 0x00000000    DC32      0x0 (0)
         0x84: 0x00000000    DC32      0x0 (0)
         0x88: 0x00000000    DC32      0x0 (0)
         0x8c: 0x000001ab    DC32      Timer0A_Handler
```

Figure 2: Modified vector table, shown in IAR debugger disassembly

Timer Configuration

To configure Timer 0A to work with your new ISR, you will need to configure the GPTMIMR and EN0 registers. Remember that Timer 0A corresponds to interrupt number 19.

Latency in Clearing an Interrupt Flag

The datasheet contains the following notes on page 113-114 (section 2.5): "After a write to clear an interrupt source, it may take several processor cycles for the NVIC to see the interrupt

source deassert. Thus if the interrupt clear is done as the last action in an interrupt handler, it is possible for the interrupt handler to complete while the NVIC sees the interrupt as still asserted, causing the interrupt handler to be re-entered errantly. This situation can be avoided by either clearing the interrupt source at the beginning of the interrupt handler or by performing a read or write after the write to clear the interrupt source (and flush the write buffer)."

Task 2 Assignments

a. **Turn off and on the user LEDs in a periodic pattern.**
   Repeat the first assigned task from Task 1, but now use interrupts. Hint: after initializing the timer and LEDs, the ISR should handle led behavior – your code doesn't need to do anything else.
b. **Controlling the timer from a user switch (GPIO Interrupt)**
   Using timers, create a program to blink LED1 at a rate of 1 Hz. With interrupts in place, let SW1 and SW2 buttons interrupt the program. When SW1 is pressed, the timer should stop counting down; instead, LED2 should be turned on. When SW2 is pressed, the timer should start counting down again, returning LED1 back to its blinking behavior. This step consequently turns off LED2
c. **Timed Traffic Light**
   Repeat the second assigned task from Task 1 using timer interrupts.

Lab Demonstration and Submission Requirements

- **Submit demo videos of Task1 and Task2 on their respective due dates. The videos should show the expected results described in Task 1 and Task 2 on the TIVA Board. Feel free to add narration or text in the video. Each video must be less than 90-seconds long. To receive full credits, your videos must thoroughly demonstrate all the required functionalities. Before submitting, refer to the rubric shown on the "Demo Video" Canvas Assignment.**
- **Write a very brief Lab Report, as framed by the "Lab Report Guide" on Canvas -> Files -> Labs -> Lab Report and Code. Make sure to include a drawing of your FSM design for this lab.**
- **Revise the style of your Lab Code, as framed by the "Programming Style Guide" on Canvas -> Files -> Labs -> Lab Report and Code.**
- **Submit your Lab Report as a pdf, and submit your Lab Code (.c and .h files). Please do not submit compressed ZIP folders or other files such as the workspace files (.eww) and project files (.ewp). Before submitting, refer to the "Lab Report and Code Rubric" on Canvas -> Files -> Labs -> Lab Report and Code.**
- **On Padlet, write about a problem you had in the lab and the fix to it, and share a tip or trick you learned while working on the lab. You can also share an aha moment that you discovered while working on the lab. Avoid duplicating comments made by your classmates. NO videos for this Padlet task, please use textual comments. The link to the Padlet can be found on Canvas -> Assignments -> Lab2: Tips and Tricks. Please note that the *Grace Period* does not apply to Padlet submission.**