**32324623229696MPS Interrupt Lab Exercise**

**Interrupt and Timer ISRs**

Student's name & ID (1): \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Partner's name & ID (2): \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Your Section number & TA's name \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Notes:**

You must work on this assignment with your partner.

Hand in a printer copy of your software listings for the team.

Hand in a neat copy of your circuit schematics for the team.

These will be returned to you so that they may be used for reference.

------------------------------- do not write below this line -----------------------------

|  |  |  |  |
| --- | --- | --- | --- |
|  | POINTS  (1) (2) | | TA init. |
| Grade for performance verification (50% max.) |  | |  |
| Enhancement (5% max.) |  | |  |
| Grade for answers to TA's questions (20% max.) |  |  |  |
| Grade for documentation and appearance (30% max.) |  | |  |
|  |  |  | TOTAL |

Grader's signature: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Date: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Interrupt and Timer ISRs**

# GOAL

By doing this lab assignment, you will learn:

1. To use IRQs and on-chip Timer interrupts on the STM32F7.
2. To add interrupt service routines to a C program using the GCC compiler.

# PREPARATION

• References: *769 Reference Manual (Register Map).pdf*

Ch. 5 (RCC), 6 (GPIO), 11 (EXTI), 28 (Basic Timers)

*Mastering STM32*

Ch. 7 (Interrupts; skip 7.2.2, 7.4.1, 7.4.3), 11 (Timers; only up to 11.2.5, and 11.4 up to 11.4.1)

## SOFTWARE DESIGN

• Do a top-down design of your program. *Provide a flowchart or equivalent.*

• Write C programs for each module/subroutine. Recall that interrupt service routines are *not* subroutines. *Provide planning documentation for both flowchart and tests. Specify test inputs and expected output or other indictors of correct operation.*

• After all the submodules are operating correctly, they may be integrated to form a higher-level module. Integrate your modules to and compile a program that is free from syntax errors.

# PROGRAMMING TASKS FOR THE STM32

1. Write a simple C program that responds to an External Interrupt 0 (/EXTI0 wired through a pushbutton switch on the protoboard to ground). The interrupt handler should set a global variable that can be seen by the main routine. The main routine should handle any I/O, i.e. output to the terminal, related to the generated interrupt.
2. Write two simple C programs that respond to timer interrupts to display elapsed time in tenths of a second using two different operational modes of the programmable STM32 timers (different timers, different scaled clocks, up or down mode, etc.). One of the modes may approximate 0.1 sec by rounding but the other must be exact to the accuracy of the processor’s clock frequency. Responses through I/O to the console may be off by milliseconds due to serial line delays, but the internal interrupts should be generated every 0.1 seconds. Be prepared to explain the differences in operation between the two programs.
3. Combine the C programs from 1 and the exact 0.1 sec timer in 2 to test reaction time. Flash the screen after a random delay and measure the time it takes for the subject to react and ground the IRQ pin. (Use the pull-downs on inputs to avoid false interrupts.) Compute the average of the number of trials or a moving average of the last 5 trials and display it on the screen. Provide a means to reset or clear the program or display; the details of how it’s done are left up to you. Optional features you may choose to add could penalize the operator for reacting *before* the signal from the program.

*Good programmer's tip:* Design the program top-down. Then write the routines bottom-up. Write them one at a time and thoroughly test each one before integrating them. This way you will have isolated any errors to the routine that you are currently writing. Good programmers follow this method.

# INTRODUCTION TO INTERRUPTS - H/W BIT SETTINGS

An interrupt in a microprocessor is a request for service by a process (a background process) that is running outside the direct control of the program. That is, the process is not a subroutine or other subprogram that is called by the program at a point in its execution. For example, a keyboard input is not controlled by the user's program but by someone pressing on the keys. In real-time applications, interrupts are often used to perform periodic functions such as to sample data or to update the error input to a servo loop. Other functions are to measure the elapsed time from a reference starting time or to measure the time interval between external events. In this laboratory exercise, only time-based interrupts are used.

For a (background) timer process to request service, several conditions must be met. The interrupt’s service vector must be enabled in the Nested Vector Interrupt Controller, and the timer’s interrupt enable bit must be set.

## Nested Vector Interrupt Controller Interrupt Service Enable Register (NVIC->ISER)

All interrupts are controlled by the Nested Vector Interrupt Controller’s Interrupt Service Enable Register (ISER). When the bit corresponding to an interrupt function is clear, that interrupt’s service routine will not be called. On reset, this register will be cleared and must be set manually for interrupts to be enabled. The bit number for an interrupt matches the interrupt’s numbers. This allows an interrupt to be enabled using the following line:

NVIC->ISER[TIM6\_DAC\_IRQn/32U] = (uint32\_t) 1U << (TIM6\_DAC\_IRQn % 32U);

Replace *n* in TIM6\_DAC\_IRQ*n* with the desired IRQ number. See **Interrupt Handler Definitions & Numbers** belowfor valid values.

## Timer Interrupt Enable Bit

Each time-based interrupt has its own DMA/Interrupt Enable Register, DIER. This must be configured to allow the overflow interrupt to be triggered. Before this register can be configured, the corresponding timer’s peripheral clock must be activated. Otherwise, any changes to this register will not be written. To enable the peripheral clock, the proper register and bit in the Reset and Clock Control (RCC) register. See Section 5.3 in the *769 Reference Manual (Register Map).pdf* for a complete listing of these registers and bit assignments. Most timers’ peripheral clocks are enabled through the RCC->APB1ENR register. **Note: you must delay for 2 cycles after setting the peripheral clock.** Use asm( “nop” ) or a busy loop to insert these delays. Once the peripheral clock has been enabled, the timer’s DIER must be configured to trigger update interrupts. This is done by setting the UIE bit:

TIM6->DIER |= TIM\_DIER\_UIE;

## Interrupt Request Flag

For an interrupt request to be made, the interrupting process must set its flag. When a 16-bit timer rolls over from 0xFFFF to 0x0000, a Timer Overflow occurs. An interrupt can be enabled to handle the overflow. If both conditions are met, an interrupt request is made to the microprocessor.

# INTRODUCTION TO TIMERS

The STM32F7 has 14 programmable timer systems with three main tiers of functionality. These tiers are Basic Timers, General Purpose Timers, and Advanced Timers. The functional differences between these timer types are summarized in Section 11.1.1 of *Mastering STM32*. Basic Timers are 16-bit timers that are not exposed to an output. General Purpose Timers are broken out, can be 16-bit or 32-bit, and have more advanced control features. The Advanced Timers have even more advanced control options as discussed in *Mastering STM32*. Timers are labelled TIM1 – TIM14, and their functions are summarized in Table 1 and Table 2 below.

## Basic Counting

One of the most common uses of a timer is for it to generate an interrupt after counting a specific number of ticks. This allows for programs to incorporate a sense of time progression. Once interrupts are enabled, a timer’s default interrupt request (IRQ) handler will be called once it reaches overflow value, such as 0xFFFF for a 16-bit timer. The name of this IRQ is usually of the form TIM*n*\_IRQHandler, where *n* is replaced with the number of the timer in question. There are several exceptions, so see the **STM32F7 Interrupt Handler Definitions & Numbers** list below for exact IRQ names.

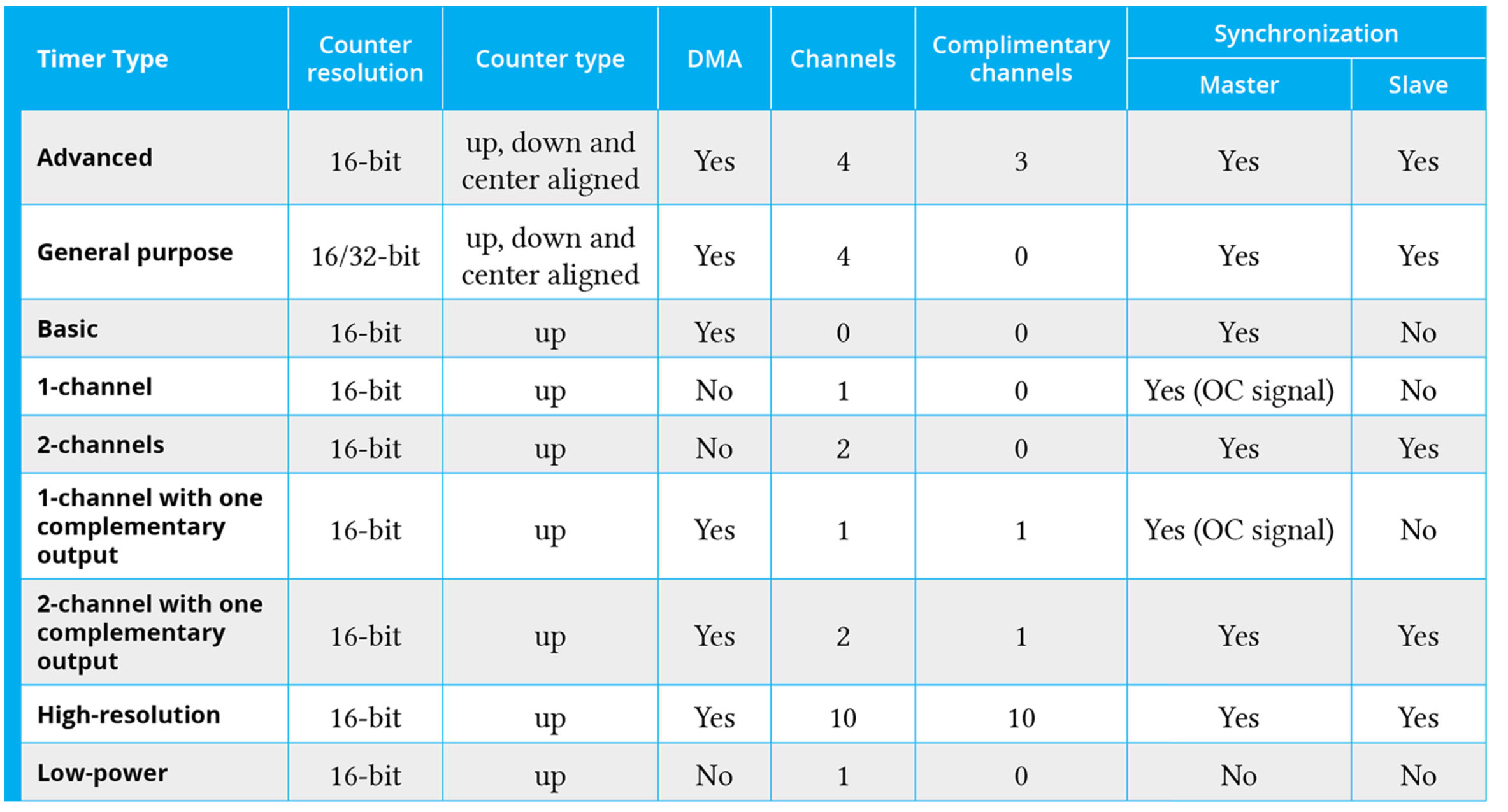
To adjust the time it takes for the timer to reach overflow, it is possible to modify the timer’s prescaler (TIM*n*->PSC) and auto-reload value (TIM*n*->ARR). For example, if we wished to create an interrupt on TIM6 every 0.015 seconds, TIM6->PSC and TIM6->ARR would need to be set in such a way that TIM6 would increment and overflow within 0.015 seconds. When using the default clock (HSI), the STM32F7 runs at 16MHz. With the prescaler set to 1599 the default tick frequency will be 10kHz, or every .0001 seconds. This means that in .015 seconds, 150 ticks will occur. Since TIM6 is a 16-bit timer, and 2^16 = 65,536, we would need to set TIM6->ARR to (65,536 – 150 = 65,386) or 0xFF6A. For convenience, the calculations to achieve this are included below. Note: For this to work the timer must a 16-bit timer and be running (TIM*n*->CR1 |= TIM\_CR1\_CEN). The IRQ handler must also manually clear the interrupt flag in the timer’s status register before returning: TIM*n*->SR = ~TIM\_DIER\_UIE;

**Example 16-bit Overflow Time Calculation**

**Table 1: STM32F7 Timer Summary**

|  |  |  |
| --- | --- | --- |
| **Timers** | **Size** | **Type** |
| TIM1 & TIM8 | 32-bit | Advanced |
| TIM3 & TIM4 | 16-bit | General |
| TIM2 & TIM5 | 32-bit | General |
| TIM6 & TIM7 | 16-bit | Basic |
| TIM10, TIM11, TIM13, & TIM14 | 16-bit | General (1-channel) |
| TIM9 & TIM12 | 16-bit | General (2-channel) |

**Table 2: Timer Type Function Chart (from Section 11.1.1 in Mastering STM32)**



# STM32F7 Interrupt Handler Definitions & Numbers

1. WWDG\_IRQHandler
2. PVD\_IRQHandler
3. TAMP\_STAMP\_IRQHandler
4. RTC\_WKUP\_IRQHandler
5. FLASH\_IRQHandler
6. RCC\_IRQHandler
7. EXTI0\_IRQHandler
8. EXTI1\_IRQHandler
9. EXTI2\_IRQHandler
10. EXTI3\_IRQHandler
11. EXTI4\_IRQHandler
12. DMA1\_Stream0\_IRQHandler
13. DMA1\_Stream1\_IRQHandler
14. DMA1\_Stream2\_IRQHandler
15. DMA1\_Stream3\_IRQHandler
16. DMA1\_Stream4\_IRQHandler
17. DMA1\_Stream5\_IRQHandler
18. DMA1\_Stream6\_IRQHandler
19. ADC\_IRQHandler
20. CAN1\_TX\_IRQHandler
21. CAN1\_RX0\_IRQHandler
22. CAN1\_RX1\_IRQHandler
23. CAN1\_SCE\_IRQHandler
24. EXTI9\_5\_IRQHandler
25. TIM1\_BRK\_TIM9\_IRQHandler
26. TIM1\_UP\_TIM10\_IRQHandler
27. TIM1\_TRG\_COM\_TIM11\_IRQHandler
28. TIM1\_CC\_IRQHandler
29. TIM2\_IRQHandler
30. TIM3\_IRQHandler
31. TIM4\_IRQHandler
32. I2C1\_EV\_IRQHandler
33. I2C1\_ER\_IRQHandler
34. I2C2\_EV\_IRQHandler
35. I2C2\_ER\_IRQHandler
36. SPI1\_IRQHandler
37. SPI2\_IRQHandler
38. USART1\_IRQHandler
39. USART2\_IRQHandler
40. USART3\_IRQHandler
41. EXTI15\_10\_IRQHandler
42. RTC\_Alarm\_IRQHandler
43. OTG\_FS\_WKUP\_IRQHandler
44. TIM8\_BRK\_TIM12\_IRQHandler
45. TIM8\_UP\_TIM13\_IRQHandler
46. TIM8\_TRG\_COM\_TIM14\_IRQHandler
47. TIM8\_CC\_IRQHandler
48. DMA1\_Stream7\_IRQHandler
49. FMC\_IRQHandler
50. SDMMC1\_IRQHandler
51. TIM5\_IRQHandler
52. SPI3\_IRQHandler
53. UART4\_IRQHandler
54. UART5\_IRQHandler
55. TIM6\_DAC\_IRQHandler
56. TIM7\_IRQHandler
57. DMA2\_Stream0\_IRQHandler
58. DMA2\_Stream1\_IRQHandler
59. DMA2\_Stream2\_IRQHandler
60. DMA2\_Stream3\_IRQHandler
61. DMA2\_Stream4\_IRQHandler
62. ETH\_IRQHandler
63. ETH\_WKUP\_IRQHandler
64. CAN2\_TX\_IRQHandler
65. CAN2\_RX0\_IRQHandler
66. CAN2\_RX1\_IRQHandler
67. CAN2\_SCE\_IRQHandler
68. OTG\_FS\_IRQHandler
69. DMA2\_Stream5\_IRQHandler
70. DMA2\_Stream6\_IRQHandler
71. DMA2\_Stream7\_IRQHandler
72. USART6\_IRQHandler
73. I2C3\_EV\_IRQHandler
74. I2C3\_ER\_IRQHandler
75. OTG\_HS\_EP1\_OUT\_IRQHandler
76. OTG\_HS\_EP1\_IN\_IRQHandler
77. OTG\_HS\_WKUP\_IRQHandler
78. OTG\_HS\_IRQHandler
79. DCMI\_IRQHandler
80. RNG\_IRQHandler
81. FPU\_IRQHandler
82. UART7\_IRQHandler
83. UART8\_IRQHandler
84. SPI4\_IRQHandler
85. SPI5\_IRQHandler
86. SPI6\_IRQHandler
87. SAI1\_IRQHandler
88. LTDC\_IRQHandler
89. LTDC\_ER\_IRQHandler
90. DMA2D\_IRQHandler
91. SAI2\_IRQHandler
92. QUADSPI\_IRQHandler
93. LPTIM1\_IRQHandler
94. CEC\_IRQHandler
95. I2C4\_EV\_IRQHandler
96. I2C4\_ER\_IRQHandler
97. SPDIF\_RX\_IRQHandler
98. DSI\_IRQHandler
99. DFSDM1\_FLT0\_IRQHandler
100. DFSDM1\_FLT1\_IRQHandler
101. DFSDM1\_FLT2\_IRQHandler
102. DFSDM1\_FLT3\_IRQHandler
103. SDMMC2\_IRQHandler
104. CAN3\_TX\_IRQHandler
105. CAN3\_RX0\_IRQHandler
106. CAN3\_RX1\_IRQHandler
107. CAN3\_SCE\_IRQHandler
108. JPEG\_IRQHandler
109. MDIOS\_IRQHandler

Note: These names are also defined in “startup\_stm32f769xx.s” in the “startup” folder.

While you may use the physical numbers corresponding to the interrupts, ARM features macro definitions for these values. These are of the form FEATURE\_NAME\_IRQn. For example, the IRQ number for TIM6\_DAC\_IRQHandler can be accessed with the TIM6\_DAC\_IRQn macro.

By default, lower numbered interrupts have higher priority than those with a greater number. This can be adjusted through the NVIC—see Section 7.5 of *Mastering STM32* for details. Whenever you are interacting with multiple interrupts take care to avoid interrupt starvation. Essentially make sure all interrupts have an opportunity to be triggered, and that the highest-level interrupt is not dominating all processing time.

# Sample Code

This sample code is a skeleton solution to the lab. Most of the game logic has been removed; it is up to you to implement it. In addition, the outline below hints at using Timer 6, GPIO pins J5 and J13 as LED outputs, and pin A0 as an external interrupt. Most, but not all registers that must be manipulated are presented in comments. You are free to change all implementation details.//----------------------------------

// Lab 2 - Timer Interrupts – Lab02.c

//----------------------------------

// Objective:

// Build a small game that records user's reaction time.

//

//

//

// -- Imports ---------------

//

#include "Lab02.h"

//

//

// -- Code Body -------------

//

volatile uint8\_t timeUpdated = 0;

volatile uint8\_t buttonPressed = 0;

volatile uint8\_t buttonReleased = 0;

volatile uint32\_t elapsed = 0;

int32\_t randomNumber = 0;

uint32\_t startTime = 0;

float averageScore = 0;

unsigned int iterations = 0;

int main() {

Sys\_Init();

Init\_Timer();

Init\_GPIO();

// Setup random number generation.

// srand(0);

while (1) {

// Main loop code goes here

printf("\033c\033[36m\033[2J");

printf("Blink!\r\n");

HAL\_Delay(1000);

blinkScreen();

HAL\_Delay(1000);

}

}

//

//

// -- Utility Functions ------

//

void blinkScreen(){

printf("\033[30;47m");

// Clear and redraw display (flash it & sound the bell).

printf("\a\033[s\033[2J\033[u");

fflush(stdout);

HAL\_Delay(100);

printf("\033[37;40m");

// Clear and redraw display (flash it).

printf("\033[s\033[2J\033[u");

fflush(stdout);

}

//

//

// -- Init Functions ----------

//

void Init\_Timer() {

// Enable the TIM6 interrupt.

// Looks like HAL hid this little gem, this register isn't mentioned in

// the STM32F7 ARM Reference Manual....

// NVIC->ISER

// Enable TIM6 clock

// RCC->APB1ENR |= RCC\_APB1ENR\_TIM6EN;

// asm ( "nop" );

// asm ( "nop" );

// Set pre-scaler to make a 100kHz ticker.

// TIM6->PSC

// Set the Auto-reload Value for 10Hz overflow

// TIM6->ARR

// Generate update events to auto reload.

// TIM6->EGR

// Enable Update Interrupts.

// TIM6->DIER

// Start the timer.

// TIM6->CR1

}

void Init\_GPIO() {

// Enable GPIO clocks?

// Looks like GPIO reg updates are synced to a base clock.

// for any changes to appear the clocks need to be running.

RCC->AHB1ENR |= RCC\_AHB1ENR\_GPIOAEN;

RCC->AHB1ENR |= RCC\_AHB1ENR\_GPIOJEN;

// Delay after an RCC peripheral clock enabling

asm ("nop");

asm ("nop");

// Set Pin 13/5 to output. (LED1 and LED2)

//GPIOJ->MODER

// GPIO Interrupt

// By default pin 0 will trigger the interrupt,

// so no need to mess with SYSCFG\_EXTICR1.

// Set Pin 0 as input (button) with pull-down.

//GPIOA->PUPDR

// Set interrupt enable for EXTI0.

// NVIC->ISER

// Unmask interrupt.

// EXTI->IMR

// Register for rising edge.

// EXTI->RTSR

// And register for the falling edge.

// EXTI->FTSR

}

//

//

// -- ISRs (IRQs) -------------

//

void TIM6\_DAC\_IRQHandler() {

// Clear Interrupt Bit

TIM6->SR = ~TIM\_DIER\_UIE;

// Other code here:

}

void EXTI0\_IRQHandler() {

// Clear Interrupt Bit (by setting it, weird I know).

EXTI->PR |= EXTI\_PR\_PR0;

// Other code here:

}

//----------------------------------

// Lab 2 - Timer Interrupts – Lab02.h

//----------------------------------

#ifndef \_\_Lab02\_H

#define \_\_Lab02\_H

//#include <stdio.h>

//#include <stdlib.h>

#include "stm32f769xx.h"

#include "stm32f7xx\_hal.h"

#include "init.h"

//

//

// -- Function Defines ------

//

void Init\_Timer();

void Init\_GPIO();

void blinkScreen();

void TIM6\_DAC\_IRQHandler();

void EXTI0\_IRQHandler();

#endif /\* \_\_Lab02\_H \*/