EEET2490 – Embedded System: OS and Interfacing

Timer and Interrupt

Objectives

Continuing development of our Bare Metal OS, in this lab, we will continue by creating drivers for timers and interrupts. This involves defining the address of the timer and interrupt peripherals, as well as any other relevant registers to enable the setting of interrupt sources, handling interrupts, setting timer count values, and managing timers. It's important to review the reference documentation to fully understand these activities.

Timers and interrupts are critical features in embedded systems, essential for managing precise time-related tasks and responding to asynchronous events. This report provides an in-depth exploration of the implementation and use of timers and interrupts on the Raspberry Pi platform, focusing on their functionality and practical applications in embedded systems.

Hardware Platform

The tutorial series is written for Raspberry Pi 3/4 B+ board. The main difference is physical peripheral base address, which is 0x3F000000 in Raspberry 3, and 0xFE000000 in Raspberry 4.

Software Requirements

Make sure that you have the following tools working properly:

- **Eclipse/VCode** IDE
- **GnuWin32** make tool
- **aarch64-none-elf** compiler
- **QEMU** emulation tool

Lab Technical Materials and References

Following are the main documents for datasheet and technical references:

- **BCM2837 ARM Peripherals - Pi 3**
- **BCM2711 ARM Peripherals - Pi 4**
- **ARM Programmer Guide** (for ARMv8-A)
- **Summary Note of Raspberry Pi Mailbox**
- **Mailbox property interface - Message TAGS**

All documents are available on Canvas at **Board Documents** [1] page and Mailbox topic module.

Lab Activities

Create Subfolders: Within your working directory, create irq and sys_timer subfolders to store the source code and header files for interrupts and timers, respectively. Reuse code from previous projects (e.g., UART1, UART0, MBOX), including the Makefile and other files like gpio.h, Link.ld, and boot.S. Maintain consistency with previous task organization for ease of development.

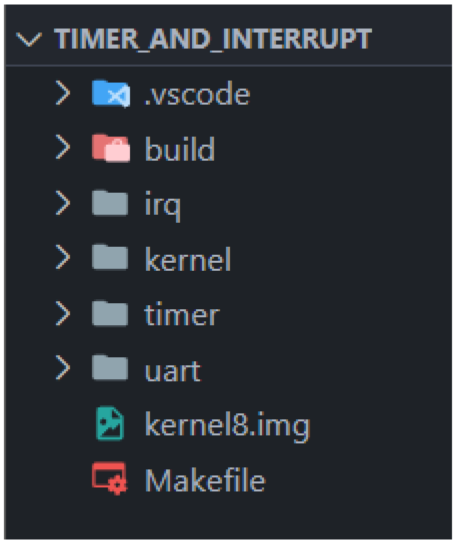


Figure 1 Folder layout

Timer

1. Header file for Timer (sys_timer.h)

Define the system timer registers based on the reference documentation. The TIMER_BASE address is 0x3F003000, and the system clock frequency (CLK_HZ) is set to 1MHz, representing 1 microsecond per tick. Define the function prototypes for initializing and handling the timers.

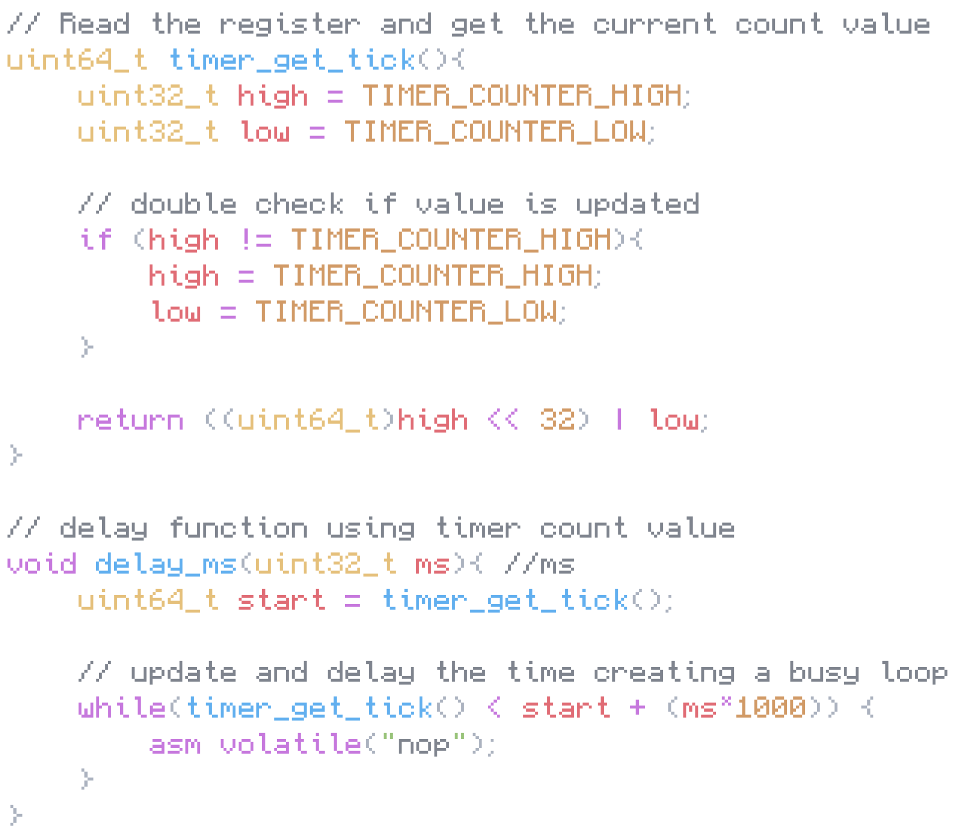
```
//--------------------------------sys_timer.h--------------------------------//
#ifndef SYS_TIMER_H
#define SYS_TIMER_H

#include "../uart/uart1.h"
#include "../uart/uart0.h"
#include "../kernel/gpio.h"
#include "../irq/irq.h"

#define CLK_HZ 1000000
#define TIMER_BASE 0x3F003000

// Timer registers
#define TIMER_CONTROL_STATUS (*(volatile unsigned int *)(TIMER_BASE + 0x0)) //
R/W 3:0
#define TIMER_COUNTER_LOW (*(volatile unsigned int *)(TIMER_BASE + 0x4)) // R
#define TIMER_COUNTER_HIGH (*(volatile unsigned int *)(TIMER_BASE + 0x8)) // R
#define TIMER_COMPARE_0 (*(volatile unsigned int *)(TIMER_BASE + 0xC)) // R/W
#define TIMER_COMPARE_1 (*(volatile unsigned int *)(TIMER_BASE + 0x10)) // R/W
#define TIMER_COMPARE_2 (*(volatile unsigned int *)(TIMER_BASE + 0x14)) // R/W
#define TIMER_COMPARE_3 (*(volatile unsigned int *)(TIMER_BASE + 0x18)) // R/W

// Timer control/status bits
#define SYS_TIMER_MATCH_0_INT 0
#define SYS_TIMER_MATCH_1_INT 1
#define SYS_TIMER_MATCH_2_INT 2
#define SYS_TIMER_MATCH_3_INT 3
```

```
// Function prototypes
void timer_init();
void handle_timer_1();
void handle_timer_3();
uint64_t timer_get_tick();
void delay_ms(uint32_t ms);

#endif
```

2. C Source file for Timer (sys_timer.c)

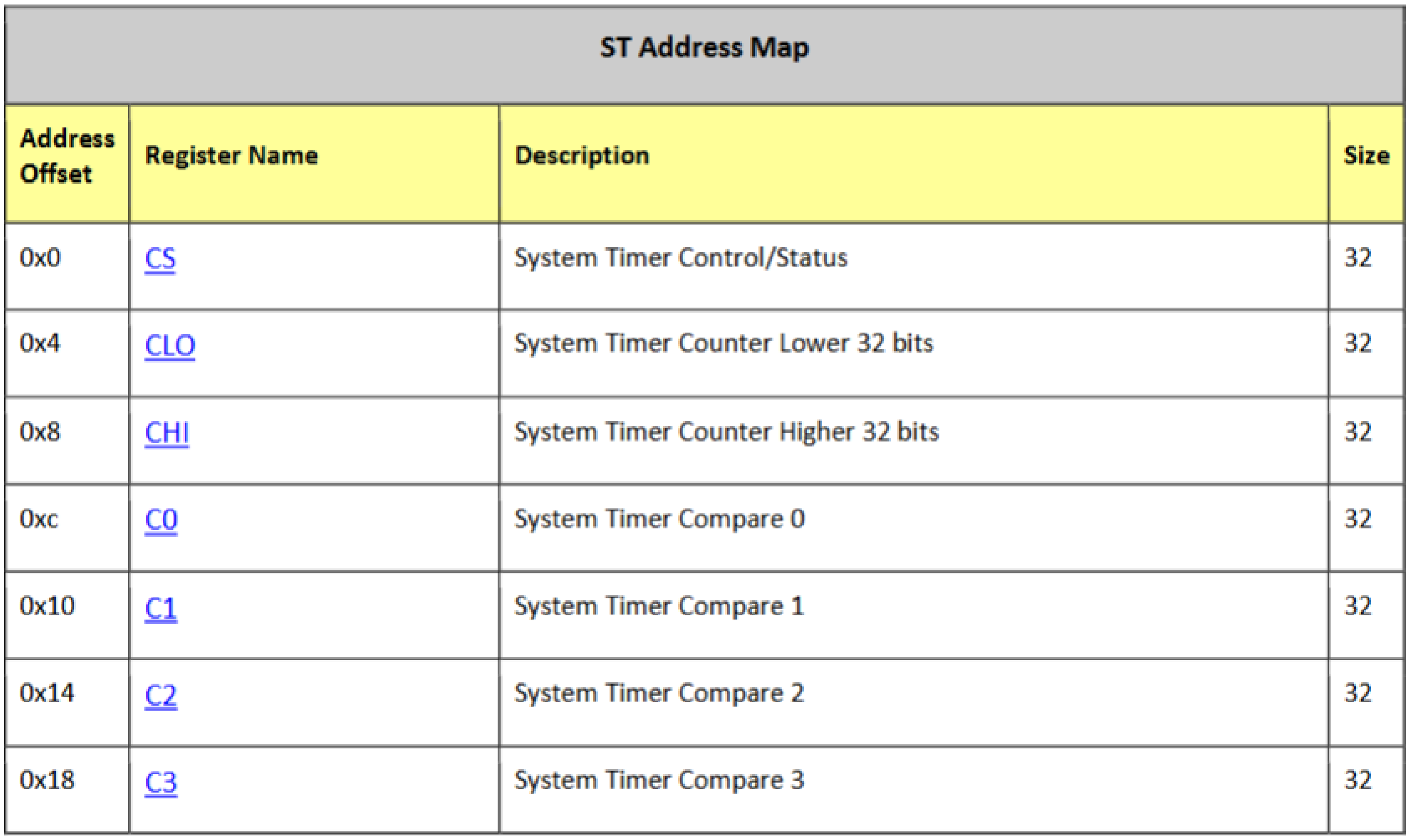
Create the content for the timer C source file as below:

```
//--------------------------------sys_timer.c----------------------------------//
#include "sys_timer.h"


const uint32_t interval_1 = CLK_HZ;
uint32_t cur_val_1 = 0;

const uint32_t interval_3 = CLK_HZ/4;
uint32_t cur_val_3 = 0;

void timer_init() {
    // clear timer status
    TIMER_CONTROL_STATUS &= ~(0xF<<0); //clear status of timer

    // update next value for timer 1
    cur_val_1 = TIMER_COUNTER_LOW;
    cur_val_1 += interval_1;
    TIMER_COMPARE_1 += cur_val_1;

    // update next value for timer 3
    cur_val_3 = TIMER_COUNTER_LOW;
    cur_val_3 += interval_3;
    TIMER_COMPARE_3 += cur_val_3;
}

void handle_timer_1() {
    cur_val_1 += interval_1;
    TIMER_COMPARE_1 = cur_val_1;
    TIMER_CONTROL_STATUS |= (1 << SYS_TIMER_MATCH_1_INT);
    uart_puts("Timer 1.\n");
}

void handle_timer_3() {
    cur_val_3 += interval_3;
    TIMER_COMPARE_3 = cur_val_3;
    TIMER_CONTROL_STATUS |= (1 << SYS_TIMER_MATCH_3_INT);
    uart_puts("Timer 3.\n");
}
```



```
// Read the register and get the current count value
uint64_t timer_get_tick(){
    uint32_t high = TIMER_COUNTER_HIGH;
    uint32_t low = TIMER_COUNTER_LOW;

    // double check if value is updated
    if (high != TIMER_COUNTER_HIGH){
        high = TIMER_COUNTER_HIGH;
        low = TIMER_COUNTER_LOW;
    }

    return ((uint64_t)high << 32) | low;
}

// delay function using timer count value
void delay_ms(uint32_t ms){ //ms
    uint64_t start = timer_get_tick();

    // update and delay the time creating a busy loop
    while(timer_get_tick() < start + (ms*1000)) {
        asm volatile("nop");
    }
}
```

<u>EXPLANATION:</u>



ST Address Map			
Address Offset	**Register Name**	**Description**	**Size**
0x0	CS	System Timer Control/Status	32
0x4	CLO	System Timer Counter Lower 32 bits	32
0x8	CHI	System Timer Counter Higher 32 bits	32
0xc	C0	System Timer Compare 0	32
0x10	C1	System Timer Compare 1	32
0x14	C2	System Timer Compare 2	32
0x18	C3	System Timer Compare 3	32

Figure 2 System Timer Register Table [1]

The system timer peripheral constitutes a vital component of the system, featuring four **32-bit** compare channels and a **64-bit** free-running counter [1]. This counter operates continuously until it exhausts its maximum capacity. Each of the four timer channels possesses its dedicated compare register, cross-checked against the 32 least significant bits of the free-running counter. Upon synchronization between the counter value and the compare register, the corresponding channel bit receives activation from the peripheral. Subsequently, this signal undergoes transmission to the interrupt controller, where it undergoes requisite offsetting to synchronize with the next timer tick.
Various registers are provided for configuring the timer. There's a Control Register for checking the status of each timer channel. The 64-bit free running counter is divided into two 32-bit registers to store the count value, and there are four timer compare registers used to set the counter value for each channel.
Although there are four timer channels labeled **0-3**, only timers **1** and **3** can be configured. Configuring timers **0** and **2** can potentially interfere with the Pi's operation and is not recommended.
Functions Overview:

- The **timer_init()** function is designed to set the count value for each timer used in practice. Before setting the count value, this function clears the **status/control register**. Since the Pi only supports a free running counter that starts counting since the system powers on, there's no exact value to set for the timer. Instead, the timer is offset by the desired amount from the current count value. For timers **1** and **3**, their count values are adjusted to generate interrupts every 1 second and every 0.25 seconds respectively, based on a system frequency of **1MHz**.
- The **handle_timer_1()** and **handle_timer_3()** functions are similar but handle events from different timers. They respond to interrupts generated by timers 1 and 3 respectively.
- The **timer_get_tick()** function retrieves the current count value of the free running counter and returns it as an unsigned 64-bit value. This function supports the **delay_ms()** function.
- The **delay_ms()** function allows users to set a desired delay in the main program. It reads the current count value and creates a loop until the counter value reaches the specified delay time.

Interrupt

1. Header for Interrupt (irq.h)

Define the interrupt register addresses based on the reference documents. ARM processors recognize three main types of interrupts:

ARM-specific peripheral interrupts: Handled by ARM cores.
GPU peripheral interrupts: Managed by the GPU.
Special Event Interrupts: Triggered by specific events.

Each interrupt type has corresponding enable and pending bits. ARM processors do not prioritize interrupts, so handling them relies on the internal system mechanisms.

```
//-----------------------------------irq.h-----------------------------------//
#ifndef IRQ_H
#define IRQ_H

#include "../timer/sys_timer.h"

#define IRQ_BASE            0x3F00B000

// Registers
#define IRQ_BASIC_PENDING   (*(volatile unsigned int*)(IRQ_BASE+0x200)) // R
#define IRQ_PENDING_1       (*(volatile unsigned int*)(IRQ_BASE+0x204)) // R
#define IRQ_PENDING_2       (*(volatile unsigned int*)(IRQ_BASE+0x208)) // R

// FIQ control
```

```
#define FIQ_CONTROL          (*(volatile unsigned int*)(IRQ_BASE+0x20C)) // R/W
6:0

// IRQ enable
#define ENABLE_IRQS_1       (*(volatile unsigned int*)(IRQ_BASE+0x210)) // Enable
interrupt 31:0
#define ENABLE_IRQS_2       (*(volatile unsigned int*)(IRQ_BASE+0x214)) // Enable
interrupt 63:32
#define ENABLE_BASIC_IRQS   (*(volatile unsigned int*)(IRQ_BASE+0x218))

// IRQ disable
#define DISABLE_IRQS_1      (*(volatile unsigned int*)(IRQ_BASE+0x21C))
#define DISABLE_IRQS_2      (*(volatile unsigned int*)(IRQ_BASE+0x220))
#define DISABLE_BASIC_IRQS (*(volatile unsigned int*)(IRQ_BASE+0x224))

// ARM peripherals interrupt
// System Timer Interrupts
#define SYS_TIMER_MATCH_1_INT    1
#define SYS_TIMER_MATCH_3_INT    3
#define USB_CONTROLLER_INT       9
#define AUX_INT                  29
#define I2C_SPI_SLV_INT          43
#define PWA0_INT                 45
#define PWA1_INT                 46
#define SMI_INT                  48
#define GPIO_INT_0               49
#define GPIO_INT_1               50
#define GPIO_INT_2               51
#define GPIO_INT_3               52
#define I2C_INT                  53
#define SPI_INT                  54
#define PCM_INT                  55
#define UART_INT                 57

// Function prototypes
void enable_irq();
void disable_irq();
void handle_irq();

#endif /* IRQ_H */
```

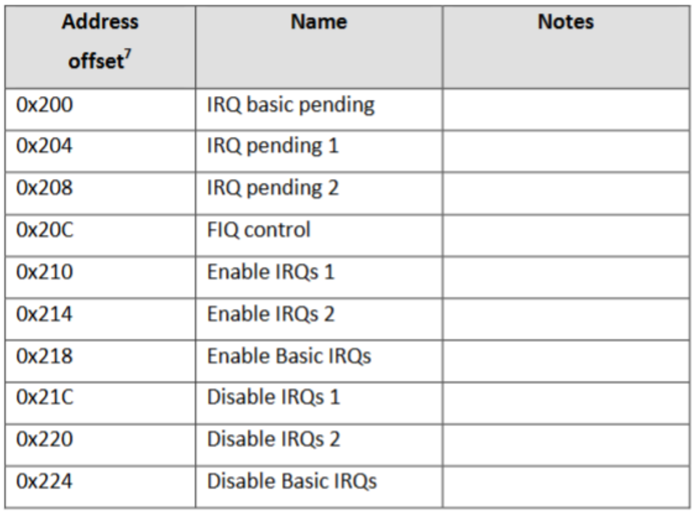
2. C Source file for Interrupt (irq.c)

```
//------------------------------------irq.c-------------------------------------//
#include "irq.h"

// Enable Interrupt for timer 1
void enable_irq() { //enable for timer 1

    /*
     * Remove the comment for the following code to enable the interrupt for
timer 1, 3 and AUX
```

```
    */

    // Enable the interrupt for timer 1, timer 3 and AUX
    // ENABLE_IRQS_1 |= (1 << SYS_TIMER_MATCH_1_INT) | (1 << AUX_INT) | (1 <<
SYS_TIMER_MATCH_3_INT); //interrupt for timer 1

    // Enable interrupt for UART
    // ENABLE_IRQS_1 |= (1 << AUX_INT);


    // Enable interrupt for TIMER 1
    ENABLE_IRQS_1 |= (1 << SYS_TIMER_MATCH_1_INT);


    // Enable interrupt for TIMER 3
    // ENABLE_IRQS_1 |= (1 << SYS_TIMER_MATCH_3_INT);

    // Double check for value in the register
    /*
    uint32_t irq = ENABLE_IRQS_1;
    uart1_puts("\nENABLE_IRQ_1 reg: ");
    uart1_bi(irq);
    uart1_puts("\nCompleted\n");
    */
}

void disable_irq() {
    DISABLE_IRQS_1 |= (1 << SYS_TIMER_MATCH_1_INT) | (1 << AUX_INT) | (1 <<
SYS_TIMER_MATCH_3_INT);
    uint32_t irq = DISABLE_IRQS_1;
    uart_puts("DISABLE_IRQ_1: ");
    uart_bi(irq);
    uart_puts("Completed\n");
}

// Check and IRQ pending register and handle each interrupt accordingly
void handle_irq() {
    uint32_t irq = IRQ_PENDING_1; // read from pending interrupt register
    while (irq) {
        if (irq & (1 << AUX_INT)) {
            irq &= ~(1 << AUX_INT);
            while ((AUX_MU_IIR & 4) == 4) {
                uart_puts("UART receive: ");
                uart_sendc(uart_getc());
                uart_puts("\n");
            }
        }
        // handling timer 1 interrupt
        if (irq & (1 << SYS_TIMER_MATCH_1_INT)) { // check for interrupt from
timer 1
            irq &= ~(1 << SYS_TIMER_MATCH_1_INT); //clear interrupt bit
            handle_timer_1();
        }
        // handling timer 3 interrupt
        if (irq & (1 << SYS_TIMER_MATCH_3_INT)) { // check for interrupt from
timer 3
            irq &= ~(1 << SYS_TIMER_MATCH_3_INT); //clear interrupt bit
            handle_timer_3();
        }
    }
}
```

EXPLANATION:

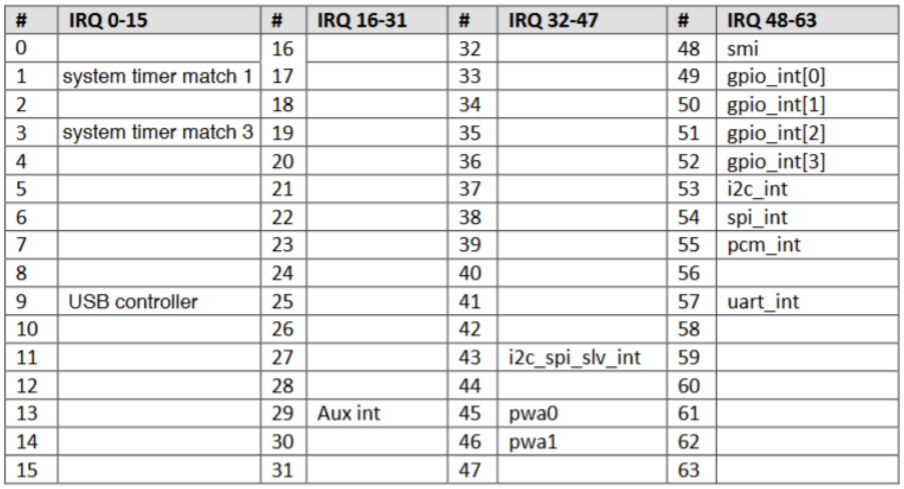
The base address for the ARM interrupt register is 0x7E00B000.
Registers overview:



Address offset[7]	Name	Notes
0x200	IRQ basic pending	
0x204	IRQ pending 1	
0x208	IRQ pending 2	
0x20C	FIQ control	
0x210	Enable IRQs 1	
0x214	Enable IRQs 2	
0x218	Enable Basic IRQs	
0x21C	Disable IRQs 1	
0x220	Disable IRQs 2	
0x224	Disable Basic IRQs	

Figure 3 Interrupt registers [1]

The following is a table which lists all interrupts which can come from the peripherals which can be handled by the ARM.
ARM peripherals interrupts table.



#	IRQ 0-15	#	IRQ 16-31	#	IRQ 32-47	#	IRQ 48-63
0		16		32		48	smi
1	system timer match 1	17		33		49	gpio_int[0]
2		18		34		50	gpio_int[1]
3	system timer match 3	19		35		51	gpio_int[2]
4		20		36		52	gpio_int[3]
5		21		37		53	i2c_int
6		22		38		54	spi_int
7		23		39		55	pcm_int
8		24		40		56	
9	USB controller	25		41		57	uart_int
10		26		42		58	
11		27		43	i2c_spi_slv_int	59	
12		28		44		60	
13		29	Aux int	45	pwa0	61	
14		30		46	pwa1	62	
15		31		47		63	

Figure 4 Timer interrupt registers [1]

The Raspberry Pi employs various interrupt sources stemming from both ARM controller peripherals and GPU peripherals. Interrupts in the ARM processor primarily fall into three categories: those from ARM-specific peripherals, those from GPU peripherals, and Special Event Interrupts. Each interrupt source, whether ARM or GPU, is accompanied by interrupt enable

bits and an interrupt pending bit [1]. The system lacks a priority setting for interrupts, meaning that the order of handling interrupts in the case of concurrent occurrences is contingent upon the system's behavior.
In the provided header file, interrupt addresses are defined based on reference documentation. The base address mentioned in the documentation refers to the Bus address of the device. To properly address these interrupts, we utilize the physical address of the peripherals, which is **0x3F00B000**. Additionally, the header file includes definitions for peripheral interrupts sourced from the ARM interrupt table.
Here's an overview of the functions:

- **enable_irq()**: This function is designed to enable specific interrupt sources. It sets bits in a register according to the ARM peripheral interrupt table provided in the documentation. For instance, to enable the interrupt for the mini UART, we set the AUX INT bit at position 29, as the mini UART is one of the auxiliary peripherals available in the Pi.
- **disable_irq()**: In contrast to **enable_irq()**, this function serves the purpose of disabling any desired interrupt source by setting its respective bit to 1.
- **handle_irq()**: This function is responsible for managing pending interrupts within the system. When an interrupt source is enabled and generates an interrupt, its corresponding bit is set in the pending interrupt register. By examining the bits in this register, we can selectively handle individual interrupts as needed.

Main Program (kernel.c)

In the main program, we simply set the output GPIO 3 and enable interrupt, timer functions.

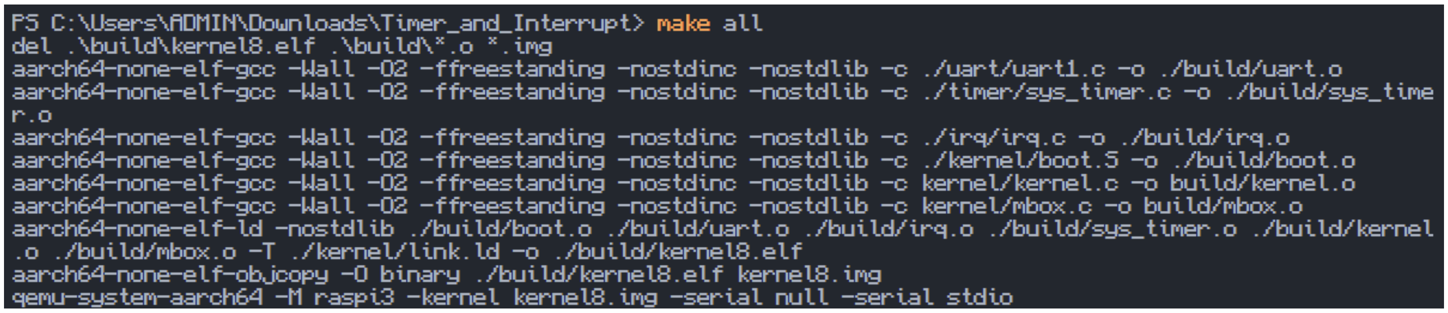
```c
#include "../uart/uart0.h"
#include "../uart/uart1.h"
#include "../timer/sys_timer.h"
#include "../irq/irq.h"
#include "../kernel/mbox.h"

void main() {
    // set GPIO 3 as output
    GPFSEL0 |= 1<<9;

    // set up serial console
    uart_init();
    uart_puts("Hello World!\n");

    //enable IRQ and Timer
    enable_irq();
    timer_init();

    // echo everything back
    while(1) {
        //read each char
        char c = uart_getc();
        //send back
        uart_sendc(c);

        handle_irq();
    }
}
```

EXPLANATION:

The main function enable the interrupt and the timer via the **enable_irq()** and **timer_init()** function, then inside a while loop the **handle_irq()** function is run continuously to handle the interrupt.

Build and Run

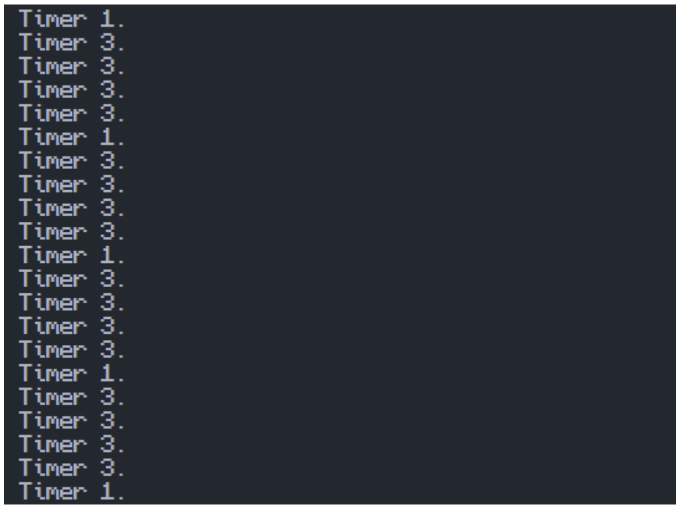
Compile and run the kernel image with QEMU. You should get an output exactly as below if all the steps are done correctly.



```
PS C:\Users\ADMIN\Downloads\Timer_and_Interrupt> make all
del .\build\kernel8.elf .\build\*.o *.img
aarch64-none-elf-gcc -Wall -O2 -ffreestanding -nostdinc -nostdlib -c ./uart/uart1.c -o ./build/uart.o
aarch64-none-elf-gcc -Wall -O2 -ffreestanding -nostdinc -nostdlib -c ./timer/sys_timer.c -o ./build/sys_time
r.o
aarch64-none-elf-gcc -Wall -O2 -ffreestanding -nostdinc -nostdlib -c ./irq/irq.c -o ./build/irq.o
aarch64-none-elf-gcc -Wall -O2 -ffreestanding -nostdinc -nostdlib -c ./kernel/boot.S -o ./build/boot.o
aarch64-none-elf-gcc -Wall -O2 -ffreestanding -nostdinc -nostdlib -c kernel/kernel.c -o build/kernel.o
aarch64-none-elf-gcc -Wall -O2 -ffreestanding -nostdinc -nostdlib -c kernel/mbox.c -o build/mbox.o
aarch64-none-elf-ld -nostdlib ./build/boot.o ./build/uart.o ./build/irq.o ./build/sys_timer.o ./build/kernel
.o ./build/mbox.o -T ./kernel/link.ld -o ./build/kernel8.elf
aarch64-none-elf-objcopy -O binary ./build/kernel8.elf kernel8.img
qemu-system-aarch64 -M raspi3 -kernel kernel8.img -serial null -serial stdio
```

Figure 5 expected output

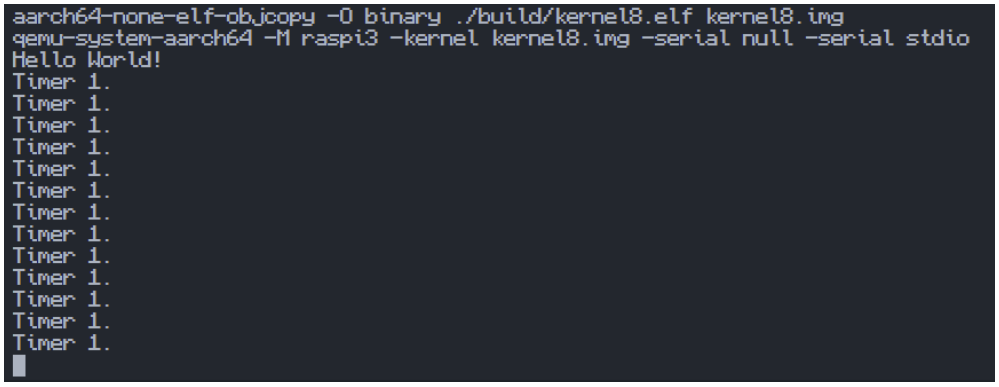
Timer 1 is configured with an offset count of 1,000,000, meaning it triggers every 1 second. Timer 3, on the other hand, is set with an offset count of 250,000, causing it to trigger every 0.25 seconds. Consequently, for every 4 ticks of Timer 3, there will be 1 tick of Timer 1.



```
Timer 1.
Timer 3.
Timer 3.
Timer 3.
Timer 3.
Timer 1.
Timer 3.
Timer 3.
Timer 3.
Timer 3.
Timer 1.
Timer 3.
Timer 3.
Timer 3.
Timer 3.
Timer 1.
Timer 3.
Timer 3.
Timer 3.
Timer 3.
Timer 1.
```

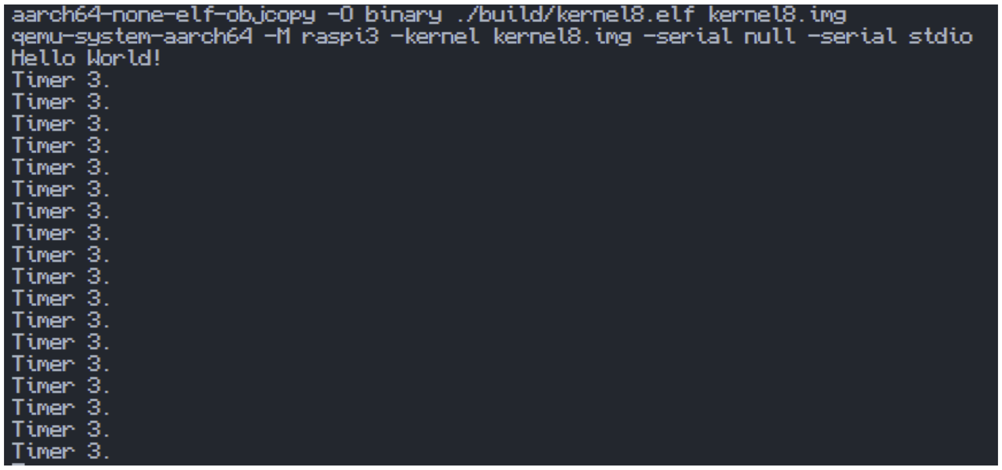
Figure 6 timer 1 and 3 intersect

To better understand how timer interrupts work, you can modify the enable_irq() function to enable interrupts for only Timer 1 or only Timer 3, rather than both at the same time.



```
aarch64-none-elf-objcopy -O binary ./build/kernel8.elf kernel8.img
qemu-system-aarch64 -M raspi3 -kernel kernel8.img -serial null -serial stdio
Hello World!
Timer 1.
Timer 1.
Timer 1.
Timer 1.
Timer 1.
Timer 1.
Timer 1.
Timer 1.
Timer 1.
Timer 1.
Timer 1.
Timer 1.
Timer 1.
```

Figure 7 timer 1



```
aarch64-none-elf-objcopy -O binary ./build/kernel8.elf kernel8.img
qemu-system-aarch64 -M raspi3 -kernel kernel8.img -serial null -serial stdio
Hello World!
Timer 3.
Timer 3.
Timer 3.
Timer 3.
Timer 3.
Timer 3.
Timer 3.
Timer 3.
Timer 3.
Timer 3.
Timer 3.
Timer 3.
Timer 3.
Timer 3.
Timer 3.
Timer 3.
Timer 3.
Timer 3.
```

Figure 8 timer 3

In this lab, we focused on creating essential drivers for timers and interrupts, enhancing our Bare Metal OS on the Raspberry Pi. By defining timer and interrupt peripherals, setting interrupt sources, managing interrupts, and configuring timer counts, we delved into key aspects of embedded systems. These components are crucial for precise time management and handling asynchronous events, forming the backbone of reliable embedded applications.
Implementing the **sys_timer** and irq modules provided practical experience in the hardware-software interaction within embedded systems. The **sys_timer** module enabled us to initialize and control hardware timers, showcasing their use in tasks like delays and periodic interrupts. The **irq** module allowed us to manage interrupts, ensuring the system's responsiveness to various events.
This lab highlighted the importance of timers and interrupts in embedded systems, offering a hands-on approach to their implementation and application. The skills and knowledge gained here are vital for developing efficient and responsive embedded applications on the Raspberry Pi and similar platforms.

Extra activities

From completing the lab activity above, you will already familiarize yourself with the concept of timer and interrupt, to continue working on Task 3 of your group assignments, try to implement how timer and interrupt work in your game (button input delay, animation, jumping, and movement delay, game timer, etc).

References

[1] "BCM2835 ARM Peripherals." [online]. Available: https://datasheets.raspberrypi.com/bcm2835/bcm2835-peripherals.pdf. (Access on May 29th, 2024).

[2] Low Level Devel, "Raspberry Pi Bare Metal Tutorial - Part 8 (Interrupts)," YouTube, Nov. 01, 2020. [online]. Available: https://www.youtube.com/watch?v=nUW1FB_5vqo&list=PLVxiWMqQvhg9FCteL7I0aohj1_YiUx1x8&index=8. (Accessed May 29th , 2024).

[3] Low Level Devel, "Raspberry Pi Bare Metal Tutorial - Part 9 (Timers)," YouTube, Nov. 03, 2020. [online]. Available: https://www.youtube.com/watch?v=2dlBZoLCMSc&list=PLVxiWMqQvhg9FCteL7I0aohj1_YiUx1x8&index=9. (Accessed May 29th , 2024).