003

Interrupt and Exception Handling

Operating Systems

Lab003 Interrupts and Exception Handling

Building a Bare-Metal ARM Application with Timer Interrupt Handling

Objective

The goal of this lab is to introduce students to interrupt-driven programming on ARM architectures, specifically the BeagleBone Black platform. By working with timer interrupts, students will:

- Learn how to configure ARM's timer peripheral for periodic execution
- Understand Interrupt Vector Table (IVT) handling and how to service interrupts in assembly
- Implement interrupt-driven execution instead of relying on busy loops
- Gain insight into how low-level exception handling works in ARM-based systems
- Understand the interaction between hardware peripherals, interrupt controllers, and the CPU

Overview

In this lab, you will extend the existing bare-metal framework by implementing a hardware timer interrupt that triggers every 2 seconds. Instead of relying on manual delay loops, the system will execute code when the timer reaches zero and raises an IRQ (Interrupt Request).

Expected Behavior:

- The main program continuously prints random numbers
- Every 2 seconds, a timer interrupt fires and prints "Tick"
- The timer automatically reloads and continues counting
- The system operates without freezing or crashing

Project Structure

The codebase is structured into three layers, which you must follow:

1. OS Level (Time Setup and dummy program)

Location: OS/os.c (main function)

Purpose: Implements the main logic of the system.

Your Tasks:

- Complete the main function to set up and enable the timer interrupt
- Print system messages before and after enabling interrupts
- Continuously generate and print random numbers (already implemented)
- Monitor whether the interrupt fires correctly every 2 seconds

2. OS Level (Interrupt Handling & Hardware Interface)

Location: OS/os.c and OS/os.h

Purpose: Provides an interface to hardware and manages low-level OS-like functionalities.

Your Tasks:

- Implement a function to configure the timer peripheral (DMTIMER2)
- Implement VIC (Vector Interrupt Controller) setup to enable IRQs
- Implement the actual timer interrupt handler, ensuring it clears the interrupt flag
- Provide UART communication functions for debugging

3. Low-Level Hardware Interface (Exception Handling)

Location: OS/root.s

Purpose: Manages the ARM Exception Vector Table and routes IRQs to the correct handlers.

Your Tasks:

- Set up the exception vector table with proper entries
- Ensure the IRQ vector points to the correct handler
- Preserve CPU registers inside the ISR before modifying them
- Ensure the ISR acknowledges the interrupt and resumes normal execution

Hardware Information

BeagleBone Black Memory Map

- **UART0 Base Address:** 0x44E09000
- **DMTIMER2 Base Address:** 0x48040000
- **INTCPS (Interrupt Controller) Base Address:** 0x48200000
- **CM_PER (Clock Manager) Base Address:** 0x44E00000
- **RAM Start Address:** 0x82000000

Timer Registers (DMTIMER2)

- **TCLR (Timer Control Register):** 0x48040038
    - Bit 0: ST (Start/Stop)
    - Bit 1: AR (Auto-reload)
    - Bit 2: CE (Compare Enable)
- **TCRR (Timer Counter Register):** 0x4804003C
    - Current countdown value
- **TLDR (Timer Load Register):** 0x48040040
    - Value to load when timer reaches zero
- **TIER (Timer Interrupt Enable Register):** 0x4804002C
    - Bit 1: OV (Overflow interrupt enable)
- **TISR (Timer Interrupt Status Register):** 0x48040028
    - Bit 1: OV (Overflow interrupt status - write 1 to clear)

Interrupt Controller Registers (INTCPS)

- **INTC_MIR_CLEAR2:** 0x482000C8
    - Clear mask bit for IRQ 68 (Timer2 interrupt)
- **INTC_CONTROL:** 0x48200048
    - Write 1 to acknowledge interrupt
- **INTC_ILR68:** 0x48200110
    - Interrupt priority and type configuration

Clock Manager Registers

- **CM_PER_TIMER2_CLKCTRL:** 0x44E00080
  - Enable timer clock (write 0x2)

Step-by-Step Implementation Guide

Step 1: Understanding the Codebase

1. **Review the existing structure:**
   - Examine how UART communication works in OS/os.c
   - Understand the vector table setup in OS/root.s
   - Review the build system in build_and_run.sh
2. **Identify what needs to be implemented:**
   - Timer initialization function
   - Interrupt controller configuration
   - IRQ handler in assembly
   - Timer interrupt service routine

Step 2: Implement Timer Initialization (OS/os.c)

Create a function timer_init(void) that:

1. **Enable the timer clock:**

   PUT32(CM_PER_TIMER2_CLKCTRL, 0x2);

2. **Configure the interrupt controller:**
   - Unmask IRQ 68 in INTC_MIR_CLEAR2
   - Set interrupt priority in INTC_ILR68 (write 0x0 for IRQ mode, priority 0)
3. **Stop and reset the timer:**
   - Write 0 to TCLR to stop the timer
   - Clear any pending interrupts by writing 0x7 to TISR
4. **Set the timer load value:**
   - Calculate the value for 2 seconds at 24 MHz: 0xFE91CA00 (approximately 2 seconds)
   - Write to TLDR (Timer Load Register)
   - Write the same value to TCRR (Timer Counter Register)

5. **Enable overflow interrupt:**
    - Write 0x2 to TIER to enable overflow interrupt
6. **Start the timer in auto-reload mode:**
    - Write 0x3 to TCLR (bit 0 = start, bit 1 = auto-reload)

Step 3: Configure VIC for Interrupt Handling (OS/os.c)

The interrupt controller is configured as part of timer initialization, but you may want a separate function enable_irq(void) in OS/root.s that:

1. Clears the I-bit in CPSR to enable IRQ interrupts
2. This allows the CPU to respond to interrupt requests

Step 4: Implement the IRQ Handler (OS/root.s)

Modify the exception vector table and implement the IRQ handler:

1. **Set up the vector table:**
    - Entry at offset 0x18 (IRQ vector) should branch to irq_handler
    - Use .align 5 to ensure 32-byte alignment
2. **Implement irq_handler:**

```
irq_handler:
  push {r0-r12, lr}   @ Save all registers
  bl timer_irq_handler  @ Call C handler
  pop {r0-r12, lr}    @ Restore registers
  subs pc, lr, #4     @ Return from interrupt
```

3. **Ensure proper stack setup:**
    - Set up separate stacks for IRQ mode if needed
    - Or use the same stack but ensure sufficient space

Step 5: Implement Timer Interrupt Service Routine (OS/os.c)

Create timer_irq_handler(void) that:

1. **Clear the timer interrupt flag:**

```
PUT32(TISR, 0x2);  // Clear overflow interrupt
```

2. **Acknowledge the interrupt to the controller:**

   PUT32(INTC_CONTROL, 0x1);

3. **Print "Tick" via UART:**

   os_write("Tick\n");

Step 6: Complete main() in OS/os.c to Initialize and Test

1. **Initialize the system:**
   - Print a startup message
   - Call timer_init() to configure the timer
   - Call enable_irq() to enable interrupts
2. **Main loop:**
   - Random number generation is already implemented
   - The loop continuously prints random numbers
   - A small delay prevents overwhelming the UART
3. **Expected output:**

```
Starting...
Timer initialized
Enabling interrupts...
123
456
789
Tick
234
567
Tick
...
```

Implementation Checklist

Review and understand the existing codebase structure

Implement timer_init() in OS/os.c

Implement enable_irq() in OS/root.s

Set up exception vector table in OS/root.s

Implement irq_handler in OS/root.s

Implement timer_irq_handler() in OS/os.c

Complete main() in OS/os.c to initialize timer and enable interrupts

Test the system and verify "Tick" appears every 2 seconds

Verify timer auto-reloads correctly

Ensure system doesn't crash or hang

Validation & Debugging

Successful Behavior

- The system prints an initialization message
- Random numbers print continuously
- Every 2 seconds, "Tick" appears in the output
- The timer resets automatically after reaching zero
- The system does not freeze or crash

Common Issues and Solutions

Issue	Likely Cause	Solution
Interrupt never fires	VIC is not configured correctly	Verify INTC_MIR_CLEAR2 register is set correctly
System hangs after enabling IRQs	CPU is not acknowledging interrupts	Ensure INTC_CONTROL is written to in the ISR
Timer counts down but does not restart	Timer is not in periodic mode	Ensure bit 1 (AR) is set in TCLR
Multiple interrupt triggers per second	Timer is not properly clearing the interrupt	Ensure TISR = 0x2 is written after servicing the IRQ
No output at all	UART not initialized or wrong base address	Verify UART base address matches BeagleBone Black
Random numbers stop printing	Stack corruption or register corruption	Check register saving/restoring in ISR

Debugging Tips

1. **Add debug prints:**
    - Print messages before and after enabling interrupts
    - Print timer register values to verify configuration
    - Print interrupt status registers
2. **Test incrementally:**
    - First, get the timer countdown working (without interrupts)
    - Then, verify the IRQ fires (add prints in handler)
    - Finally, ensure proper cleanup and return
3. **Check register values:**
    - If an interrupt is not firing, print INTC_MIR_CLEAR2 and TIER registers
    - Verify timer is actually counting: read TCRR periodically
    - Check that TISR shows the interrupt pending

Deliverables

1. Updated Source Files

- OS/os.c: Contains main program, timer setup, interrupt handlers, and UART functions
- OS/os.h: Header file with function declarations
- OS/root.s: Handles IRQ vectoring and exception handling

2. Documentation

- **Code comments:** Explain how the timer interrupt works
- **Function documentation:** Describe each modified/added function
- **Architecture explanation:** Explain how the system handles IRQs from hardware to handler

Expected Outcomes

By the end of this lab, students will:

- Understand how hardware timers work in ARM-based systems
- Implement interrupt-driven execution instead of busy loops
- Successfully route IRQs from the interrupt controller to the CPU
- Develop a deeper understanding of low-level exception handling in ARM

- Understand the interaction between hardware peripherals and the CPU
- Learn to debug interrupt-related issues

Tips for Success

1. **Test Incrementally:**
    - First, get the timer countdown working
    - Then, verify the IRQ fires
    - Finally, ensure the system acknowledges and clears interrupts correctly
2. **Use Debugging Prints:**
    - Add print statements before and after enabling IRQs
    - Print VIC IRQ status before and after clearing the interrupt
    - Print timer register values to diagnose issues
3. **Check Register Values:**
    - If an interrupt is not firing, print the VIC & Timer registers to diagnose the issue
    - Verify bit patterns match expected values
4. **Understand the Flow:**
    - Device (Timer) → Interrupt Controller (INTCPS) → CPU (IRQ exception) → Vector Table → Handler
    - Each step must be configured correctly for interrupts to work
5. **Be Careful with Registers:**
    - Always save all registers in the ISR
    - Restore them exactly as they were
    - Use the correct return instruction (subs pc, lr, #4)

Build and Run Instructions

1. **Build the project:**

    ./build_and_run.sh

2. **Or manually:**

```
# Assemble root.s
arm-none-eabi-as -o bin/root.o OS/root.s

# Compile os.c
arm-none-eabi-gcc -c -mcpu=cortex-a8 -mfpu=neon -mfloat-abi=hard \
  -I OS -Wall -O2 -nostdlib -nostartfiles -ffreestanding \
```

```
    -o bin/os.o OS/os.c

# Link
arm-none-eabi-gcc -nostartfiles -T linker.ld \
    -mcpu=cortex-a8 -mfpu=neon -mfloat-abi=hard \
    -o bin/program.elf bin/root.o bin/os.o

# Convert to binary
arm-none-eabi-objcopy -O binary bin/program.elf bin/program.bin
```

Additional Resources

ARM Interrupt handlers

IRQ Exception

Cortex-A8 (GES)