Tutorial 10: I/O System

Problem 1: Polling and Interrupt

Assume for a processor with a 450-MHz clock it takes 450 clock cycles for a polling operation (call polling routine, accessing the device, and returning). The overhead for an interrupt operation is 525 clock cycles. Hard disk transfers data in 16-byte, chunks and can transfer at 12 MB/second rate.

A. If the processor uses interrupt technique and the interrupt rate is equal to the software polling rate, what percentage of the processor time is tied up in servicing the interrupt by the hard disk during the data transfer?

B. If the processor's clock frequency increases to 475 MHz and the processor uses interrupt technique and the hard disk is only active 2.5% of the time, what percentage of the processor time is tied up in servicing the interrupt by the hard disk?

C. If the processor's clock frequency increases to 800 MHz and the processor uses interrupt technique and the hard disk is only active 3% of the time, what percentage of the processor time is tied up in servicing the interrupt by the hard disk?

D. To reduce interrupt overheads, the hard disk controller is augmented with a 64-byte buffer, and now only interrupts the CPU when the buffer is full. The interrupt rate is equal to the software polling rate. What percentage of the processor (450-MHz) time is tied up in servicing the interrupt by the hard disk during the data transfer?
D. The required interrupt rate can be computed as:

\[
\text{(Interrupt Rate)} = \frac{(12 \text{ MB/sec})}{(64 \text{ Byte})} = 0.1875 \text{ Mega interrupts per second}
\]

\[
\text{(Number of clock cycles per second required for servicing interrupt)} = (0.75 \text{ Mega interrupts per second}) \times (525 \text{ clock cycles}) = 98.4375 \text{ Mega clock cycles per seconds}
\]

\[
\text{(Percentage loading due to polling)} = \frac{(393.75 \text{ Mega clock cycles per seconds})}{(450\text{MHz Processor Clock Frequency})} = 0.21875 = 22\%.
\]

**Problem 2: Handling Interrupts**

Consider ARM DSLMU interrupt controller shown in Figure 1. Write an interrupt service routine that implements a re-entrant interrupt system with 8 priority levels (Input 0 with lowest priority and 7 with highest priority).

![Diagram](image)

*Figure 1: Interrupt Controller on the DSLMU Microcontroller Board*
6. disables interrupts by setting bit 7 (the I bit) of CPSR to one, and
7. loads the address of the interrupt vector (0x00000018) into the Program Counter PC.
At this point, the ARM processor executes the code at the interrupt vector address, which can only be a branch to the IRQ interrupt handler code (irq_handler). Please note that, for the duration of the interrupt (or, more correctly, while in that specific processor mode), the banked registers replace the ordinary ones. While the processor is in IRQ mode, the ordinary registers R13–R14 cannot be seen directly: only the registers R13_irq–R14_irq can be seen. This is shown in Figure 3.

Once the IRQ interrupt handler does what is necessary to handle the interrupt, the handler:

1. moves the contents of LR_irq less some an offset of –4, into PC, and
2. copies SPSR back to CPSR.
Both of these steps can be done in one instruction (sub pc, lr, #4). Doing so has the effect of returning to the original task, running under the original processor mode and with the CPSR as it was originally before the interrupt.

Figure 3: Switching from User Mode to IRQ Mode on a Normal Interrupt Exception

Figure 4 is a simple non-entrant interrupt handler. In this code registers R0 – R12 are saved first. That is because the interrupted code would have certainly used them. We also save the LR (LR_irq) register as they the handler employs a “BL” instruction.

```assembly
irq_handler:
    stmfd sp!, {r0-r12,lr}  ; Save r0 – r12, and lr to the stack (pointed to by SP_irq)
    …
    …
    bl      process_irq        ; Branch to code to handle the interrupt
    …
    …
    ldmfdS sp!, {r0-r12,lr}  ; restore a1-a4 and lr from the stack (pointed to by SP_irq)
    SUBS   pc, lr ,#4        ; Return from IRQ handler restores CPSR
```

Figure 4: Simple non-entrant irq_handler

The code in Figure 4 code is not re-entrant, as the I bit in CPSR is disabled when the handler is entered.
Figure 5 is a simple re-entrant interrupt handler. This is achieved by enabling the I bit in CPSR in the handler code.

```
irq_handler:
.set    ARM_PSR_i, 0x80    ; I bit (bit 7) in CPSR/SPSR
.set    irq_status,  0x18            ; IRQ Status port offset
stmfd   sp!, {r0-r12, lr}  ; Save r0 – r12, and lr to the stack (pointed to by SP_irq)
ldr     r0, =iobase                ; R0 = base of I/O space
ldrb    r1, [r0, #irq_status]      ; check the status (source) of the IRQs
mrs     ip, cpsr           ; Get current value of CPSR into IP (R12)
bic     ip, ip, #(ARM_PSR_i) ; clear I bit, (enable all irq the interrupts)
msr     cpsr, ip        ; Actually make the changes to CPSR
…
…
bl      process_irq        ; Branch to code to handle the interrupt
…
…
mrs     ip, cpsr           ; Get current value of CPSR into IP (R12)
or     ip, ip, #(ARM_PSR_i) ; clear I bit, (disable all irq the interrupts)
msr     cpsr, ip        ; Actually make the changes to CPSR
ldmfdS   sp!, {r0-r12,lr}  ; restore a1-a4 and lr from the stack (pointed to by SP_irq)
SUBS   pc, lr ,#4        ; Return from IRQ handler restores CPSR
```

Figure 5: Simple re-entrant irq_handler

The code in Figure 5 although is rendered re-entrant, suffers from a major flaw. It has no system priority levels. Any interrupt source can interrupt IRQ handler belonging to another interrupt source including its own. As successive interrupts interrupt one another, the IRQ stack eventually overflows.

The code in Figure 6 only enables the I bit for the next interrupt if the newly taken interrupt is at a higher priority than the interrupt currently being processed.

```
irq_handler:
.set    ARM_PSR_i, 0x80    ; I bit (bit 7) in CPSR/SPSR
.set    irq_status,  0x18            ; IRQ Status port offset
stmfd   sp!, {r0-r12, lr}  ; Save r0 – r12, and lr to the stack (pointed to by SP_irq)
mov  r0, #0x80             ; level 7
bl   process_level
mov  r0, #0x40             ; level 6
bl   process_level
mov  r0, #0x20             ; level 5
bl   process_level
mov  r0, #0x10             ; level 4
bl   process_level
mov  r0, #0x8              ; level 3
bl   process_level
mov  r0, #0x4              ; level 2
bl   process_level
mov  r0, #0x2              ; level 1
bl   process_level
mov  r0, #0x1              ; level 0
bl   process_level
ldmfd   sp!, {r0-r12,lr}    ; restore a1-a4 and lr from the stack (pointed to by SP_irq)
subS   pc, lr ,#4        ; Return from IRQ handler restores CPSR
```

```
process_level:
```

process_level:
stmfd   sp!, {r4, r5, lr}  ; Save r4 – r5, and lr to the stack (pointed to by SP_irq)
mov     r5, r0
ldr     r4, =irq_level             ; R0 = storage for irq_level
ldrb    r3, [r4]                   ; check the status (source) of current IRQ level
str     r3, [sp, #-4]!             ; store the current IRQ level on stack; IRQ level
ldr     r1, =iobase                ; R1 = base of I/O space
ldrb    r0, [r1, #irq_status]      ; check the status (source) of the IRQs
and     r2, r0, r5                 ; find priority Level in r5
cmp     r2, r3                     ; compare the current and taken; interrupt levels
strbgt r2, [r4]                   ; save the new level if greater
strgt   r3, [sp, #-4]!             ; store the current IRQ level on stack; IRQ level
ble return
mrs     ip, cpsr           ; Get current value of CPSR into IP (R12)
bic     ip, ip, #(ARM_PSR_i) ; clear I bit, (enable all irq the interrupts)
msr     cpsr, ip           ; Actually make the changes to CPSR
…
…
bl      process_irq        ; Branch to code to handle the interrupt
…
…
mrs     ip, cpsr           ; Get current value of CPSR into IP (R12)
orr     ip, ip, #(ARM_PSR_i) ; clear I bit, (disable all irq the interrupts)
msr     cpsr, ip           ; Actually make the changes to CPSR
str     r3, [sp, #-4]!     ; Restore the previous IRQ level on stack
return:
ldmfd   sp!, {r4, r5, lr}  ; Restore r4 – r5, and lr from the stack (pointed to by SP_irq)
mov  pc, lr
Figure 6: Final re-entrant irq_handler