Exception and Interrupt Handling

Minsoo Ryu

Department of Computer Science and Engineering
Hanyang University

msryu@hanyang.ac.kr

Contents

Introduction

- An interrupt or exception is usually defined as an event that alters the sequence of instructions executed by a processor
  - Such events correspond to electrical signals generated by hardware circuits both inside and outside of the CPU chip

- Definitions of interrupts and exceptions
  - Interrupts are asynchronous events (예정되지 않은) that are generated by other hardware devices at arbitrary times with respect to the CPU clock signals
  - Exceptions are synchronous events (예정됨) that are produced by the CPU control unit while executing instructions and are called synchronous because the control unit issues them only after terminating the execution of an instruction

Definition and Classification

- **Interrupts are issued by**
  - **Interval timers and I/O devices;**
  - **For instance, the arrival of a keystroke from a user sets off an interrupt**

- **Exceptions are caused by**
  - **Programming errors**
    - The kernel handles the exception by delivering to the current process one of the signals familiar to every Unix programmer
  - **Anomalous conditions**
    - The kernel performs all the steps needed to recover from the anomalous condition, such as a page fault or a request (via an int instruction) for a kernel service

Definition and Classification

- **However, different architectures have different meanings of interrupts and exceptions**
  - **Intel 80x86 microprocessor manuals designate**
    - Synchronous interrupts → exceptions
    - Asynchronous interrupts → interrupts
  - **ARM processor manuals designate**
    - The ARM processor has seven exceptions that can halt the normal sequential execution of instructions: Data Abort, Fast Interrupt Request, Prefetch Abort, Software Interrupt, Reset, and Undefined Instruction

Contents

1. Interrupts and Exceptions
2. Intel 80x86 Processors
3. ARM Processors

Intel's Classification of Interrupts

- **Interrupts**
  - **Maskable interrupts**
    - Sent to the INTR pin of the microprocessor
    - They can be disabled by clearing the IF flag of the eflags register
    - All IRQs issued by I/O devices give rise to maskable interrupts
  - **Nonmaskable interrupts**
    - Sent to the NMI (Nonmaskable Interrupts) pin of the microprocessor
    - They are not disabled by clearing the IF flag
    - Only a few critical events, such as hardware failures, give rise to nonmaskable interrupts

Intel's Classification of Exceptions

- **Exceptions**
  - **Processor-detected exceptions**
    - Generated when the CPU detects an anomalous condition while executing an instruction
    - These are divided into three groups, depending on the value of the eip register that is saved on the Kernel Mode stack
  - **Programmed exceptions**
    - Occur at the request of the programmer
    - They are triggered by int or int3 instructions

Programmed Exceptions

- Programmed exceptions are handled by the control unit as traps
  - They are often called software interrupts
- Such exceptions have two common uses
  - To implement system calls
  - To notify a debugger of a specific event

Interrupt and Exception Vectors (1/2)

- Each interrupt or exception is identified by a number ranging from to 255
  - For some unknown reason, Intel calls this 8-bit unsigned number a vector
  - The vectors of nonmaskable interrupts and exceptions are fixed
  - The vectors of maskable interrupts can be altered by programming the Interrupt Controller

Interrupt and Exception Vectors (2/2)

- **Linux uses the following vectors:**
  - **Vectors ranging from 0 to 31 correspond to exceptions and nonmaskable interrupts**
  - **Vectors ranging from 32 to 47 are assigned to maskable interrupts, that is, to interrupts**
  - **The remaining vectors ranging from 48 to 255 may be used to identify software interrupts**
    - Linux uses only one of them, namely the 128 or 0x80 vector, which it uses to implement system calls
    - When an int 0x80 Assembly instruction is executed by a process in User Mode, the CPU switches into Kernel Mode and starts executing the system_call( ) kernel function

IRQs and Interrupts (1/5)

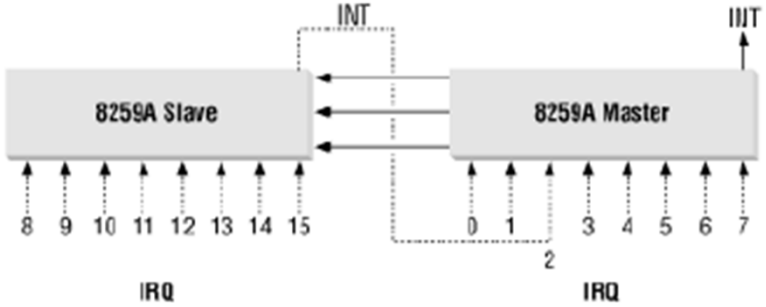
- **Each hardware device controller has an output line designated as an IRQ (Interrupt ReQuest)**
  - **All existing IRQ lines are connected to the input pins of a hardware circuit called the Interrupt Controller, which performs the following actions:**
  - **1. Monitors the IRQ lines, checking for raised signals**
  - **2. If a raised signal occurs on an IRQ line:**
    - a. Converts the raised signal received into a corresponding vector
    - b. Stores the vector in an Interrupt Controller I/O port, thus allowing the CPU to read it via the data bus
    - c. Sends a raised signal to the processor INTR pin—that is, issues an interrupt
    - d. Waits until the CPU acknowledges the interrupt signal by writing into one of the Programmable Interrupt Controllers (PIC) I/O ports; when this occurs, clears the INTR line
  - **3. Goes back to step 1**

IRQs and Interrupts (2/5)

- IRQ lines are sequentially numbered starting from 0
  - The first IRQ line is usually denoted as IRQ0
- Intel's default vector associated with IRQn is n+32
  - As mentioned before, the mapping between IRQs and vectors can be modified by issuing suitable I/O instructions to the Interrupt Controller ports

IRQs and Interrupts (3/5)

- **A typical connection "in cascade" of two Intel 8259A PICs**
  - **Can handle up to 15 different IRQ input lines**
  - **The INT output line of the second PIC is connected to the IRQ2 pin of the first PIC**
    - A signal on that line denotes the fact that an IRQ signal on any one of the lines IRQ8-IRQ15 has occurred
  - **The number of available IRQ lines is thus traditionally limited to 15; however, more recent PIC chips are able to handle many more input lines**



IRQs and Interrupts (4/5)

- Other lines not shown in the figure connect the PICs to the bus
  - Bidirectional lines D0-D7 connect the I/O port to the data bus, while another input line is connected to the control bus and is used for receiving acknowledgment signals from the CPU
- Since the number of available IRQ lines is limited, it may be necessary to share the same line among several different I/O devices
  - When this occurs, all the devices connected to the same line will have to be polled sequentially by the software interrupt handler in order to determine which of them has issued an interrupt request

IRQs and Interrupts (5/5)

- Each IRQ line can be selectively disabled
  - The PIC can be programmed to disable IRQs
  - The PIC can be told to stop issuing interrupts that refer to a given IRQ line or vice versa to enable them
  - Disabled interrupts are not lost; the PIC sends them to the CPU as soon as they are enabled again
  - This feature is used by most interrupt handlers, since it allows them to process IRQs of the same type serially
- Selective enabling/disabling of IRQs is not the same as global masking/unmasking of maskable interrupts
  - When the IF flag of the eflags register is clear, any maskable interrupt issued by the PIC is simply ignored by the CPU
  - The cli and sti assembly instructions, respectively, clear and set that flag

Contents

1. Interrupts and Exceptions
2. Intel 80x86 Processors
3. ARM Processors

Introduction

- The exception handlers are responsible for handling errors, interrupts, and other events generated by the external system

- The ARM processor has seven exceptions that can halt the normal sequential execution of instructions: Data Abort, Fast Interrupt Request, Prefetch Abort, Software Interrupt, Reset, and Undefined Instruction

ARM Processor Exceptions and Modes

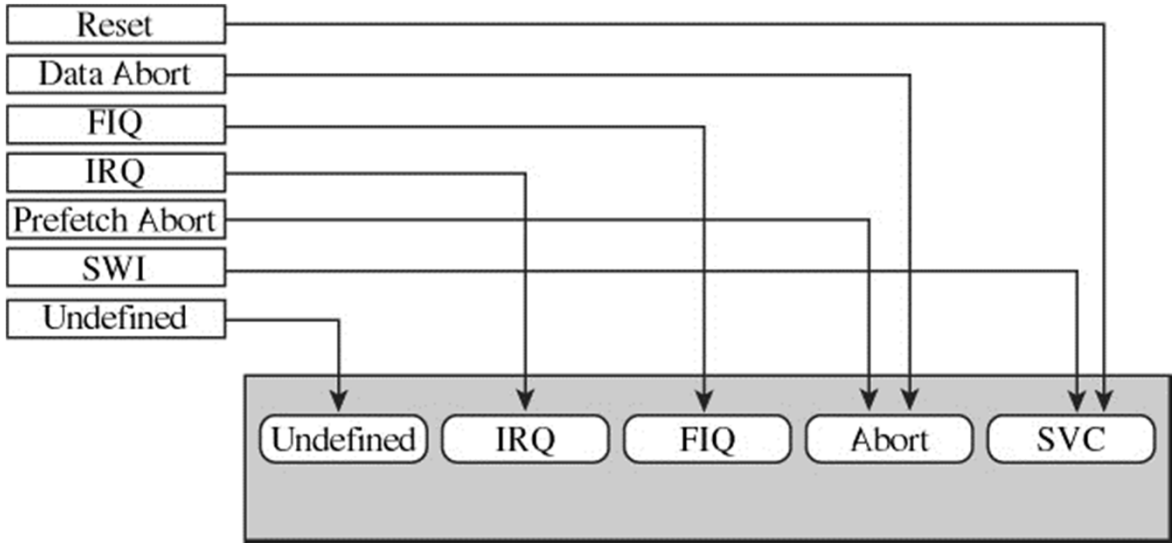
- Each exception causes the core to enter a specific mode
- Any of the ARM processor modes can be entered manually by changing the cpsr
  - User and system mode are the only two modes that are not entered by a corresponding exception, in other words, to enter these modes you must modify the cpsr

ARM processor exceptions and associated modes.

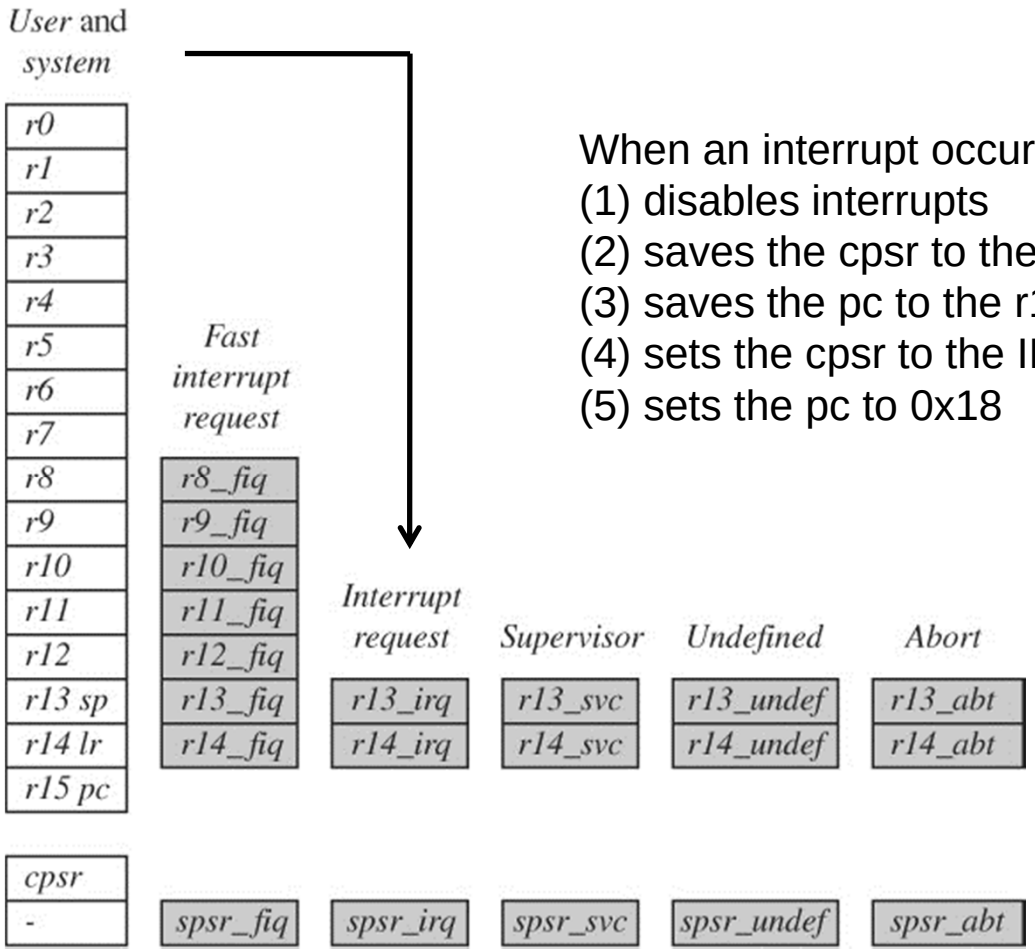
Exception	Mode	Main purpose
Fast Interrupt Request	*FIQ*	fast interrupt request handling
Interrupt Request	*IRQ*	interrupt request handling
SWI and Reset	*SVC*	protected mode for operating systems
Prefetch Abort and Data Abort	*abort*	virtual memory and/or memory protection handling
Undefined Instruction	*undefined*	software emulation of hardware coprocessors

ARM Processor Exceptions and Modes

- When an exception causes a mode change, the core automatically
  - saves the cpsr to the spsr of the exception mode
  - saves the pc to the lr of the exception mode
  - sets the cpsr to the exception mode
  - sets the pc to the address of the exception handler



Register Changes on IRQ Occurrence



When an interrupt occurs, the processor
(1) disables interrupts
(2) saves the cpsr to the spsr_irq
(3) saves the pc to the r14_irq
(4) sets the cpsr to the IRQ mode
(5) sets the pc to 0x18

Vector Table

- **Vector table is a table of addresses that the ARM core branches to when an exception is raised**
- **These addresses commonly contain branch instructions of one of the following forms**
  - **B <address> This branch instruction provides a branch relative from the pc**
  - **LDR pc, [pc, #offset] This load register instruction loads the handler address from memory to the pc**
    - The address is an absolute 32-bit value stored close to the vector table
    - This results in a slight delay due to the extra memory reference
  - **LDR pc, [pc, #-0xff0] This load register instruction loads a specific interrupt service routine address 0xffffff030 to the pc**
    - This is only used when a vector interrupt controller is present (VIC PL190)
  - **MOV pc, #immediate This move instruction copies an immediate value into the pc**

Vector Table

- You can also have other types of instructions in the vector table
  - The FIQ handler might start at address offset +0x1c
  - Thus, the FIQ handler can start immediately at the FIQ vector location, since it is at the end of the vector table

Vector table and processor modes.

Exception	Mode	Vector table offset
Reset	SVC	+0x00
Undefined Instruction	UND	+0x04
Software Interrupt (SWI)	SVC	+0x08
Prefetch Abort	ABT	+0x0c
Data Abort	ABT	+0x10
Not assigned	—	+0x14
IRQ	IRQ	+0x18
FIQ	FIQ	+0x1c

```
0x00000000: 0xe59ffa38  RESET: > ldr  pc, [pc, #reset]
0x00000004: 0xea000502  UNDEF:   b    undInstr
0x00000008: 0xe59ffa38  SWI   :  ldr  pc, [pc, #swi]
0x0000000c: 0xe59ffa38  PABT  :  ldr  pc, [pc, #prefetch]
0x00000010: 0xe59ffa38  DABT  :  ldr  pc, [pc, #data]
0x00000014: 0xe59ffa38   -    :  ldr  pc, [pc, #notassigned]
0x00000018: 0xe59ffa38  IRQ   :  ldr  pc, [pc, #irq]
0x0000001c: 0xe59ffa38  FIQ   :  ldr  pc, [pc, #fiq]
```

Exception Priorities

- **Exception can occur simultaneously, so the processor has to adopt a priority mechanism**
  - **For instance, the Reset exception is the highest priority and occurs when power is applied to the processor**

Exception priority levels.

Exceptions	Priority	*I* bit	*F* bit
Reset	1	1	1
Data Abort	2	1	—
Fast Interrupt Request	3	1	1
Interrupt Request	4	1	—
Prefetch Abort	5	1	—
Software Interrupt	6	1	—
Undefined Instruction	6	1	—

The Reset Handler

- The reset handler initializes the system, including setting up memory and caches
  - External interrupt sources should be initialized before enabling IRQ or FIQ interrupts to avoid the possibility of spurious interrupts occurring before the appropriate handler has been set up
  - The reset handler must also set up the stack pointers for all processor modes

- During the first few instructions of the handler, it is assumed that no exceptions or interrupts will occur
  - The code should be designed to avoid SWIs, undefined instructions, and memory accesses that may abort, that is, the handler is carefully implemented to avoid further triggering of an exception

The Data Abort Handler

- Data Abort exceptions occur when the memory controller or MMU indicates that an invalid memory address has been accessed (for example, if there is no physical memory for an address) or when the current code attempts to read or write to memory without the correct access permissions
  - An FIQ exception can be raised within a Data Abort handler since FIQ exceptions are not disabled
  - When the FIQ is completely serviced, control is returned back to the Data Abort handler

The FIQ and IRQ Handlers

- A Fast Interrupt Request (FIQ) exception occurs when an external peripheral sets the FIQ pin to nFIQ
  - An FIQ exception is the highest priority interrupt
  - The core disables both IRQ and FIQ exceptions on entry into the FIQ handler

- An Interrupt Request (IRQ) exception occurs when an external peripheral sets the IRQ pin to nIRQ
  - An IRQ exception is the second-highest interrupt
  - On entry to the IRQ handler, the IRQ exceptions are disabled and should remain disabled until the current interrupts source has been cleared

The Prefetch Abort Handler

- A Prefetch Abort exception occurs when an attempt to fetch an instruction results in a memory fault
  - This exception is raised when the instruction is in the execute stage of the pipeline and if none of higher exceptions have been raised
  - On entry to the handler, IRQ exceptions will be disabled, but the FIQ exceptions will remain unchanged
  - If FIQ is enabled and an FIQ exception occurs, it can be taken while servicing the Prefetch Abort

The SWI Handler

sw debugger - swi
hw debugger - jteg

- **A Software Interrupt (SWI) exception occurs when the SWI instruction is executed and none of the other higher-priority exceptions have been flagged**
  - **On entry to the handler, the cpsr will be set to supervisor mode**

- **If the system uses nested SWI calls, the link register r14 and spsr must be stored away before branching to the nested SWI to avoid possible corruption of the link register and the spsr**

The Undefined Instruction Handler

- An Undefined Instruction exception occurs when an instruction not in the ARM or Thumb instruction set reaches the execute stage of the pipeline and none of the other exceptions have been flagged
  - The ARM processor "asks" the coprocessors if they can handle this as a coprocessor instruction
  - Since coprocessors follow the pipeline, instruction identification can take place in the execute stage of the core
  - If none of the coprocessors claims the instruction, an Undefined Instruction exception is raised
- Both the SWI instruction and Undefined Instruction have the same level of priority, since they cannot occur at the same time

Link Register Offsets

- When an exception occurs, the link register is set to a specific address based on the current pc
  - When an IRQ exception is raised, the link register lr points to the last executed instruction plus 8
  - Care has to be taken to make sure the exception handler does not corrupt lr because lr is used to return from an exception handler
  - The IRQ exception is taken only after the current instruction is executed, so the return address has to point to the next instruction, or lr - 4

Link Register Offsets

backward compatability 때문에 link register offset을 그대로 유지

Table 9.4 Useful link-register-based addresses.

Exception	Address	Use
Reset	—	*lr* is not defined on a Reset
Data Abort	*lr* – 8	points to the instruction that caused the Data Abort exception
FIQ	*lr* – 4	return address from the FIQ handler
IRQ	*lr* – 4	return address from the IRQ handler
Prefetch Abort	*lr* – 4	points to the instruction that caused the Prefetch Abort exception
SWI	*lr*	points to the next instruction after the SWI instruction
Undefined Instruction	*lr*	points to the next instruction after the undefined instruction

a 1004
b 1008

prefetch - pc 증가후, fetch,decode 사이에 파악가능
execute - execute 마지막 부분에서 interrupt 검사

arm7
fetch - decode - execute

b a
pc = 1008 undef,swi 파악됨 따라서 lr - 1008 ==pc

Link Register Offsets

❒ Typical methods of returning from an IRQ and FIQ handler

```
handler
      <handler code>
      …
      SUBS  pc, r14, #4  ; pc=r14-4
```

When there is an S and the pc is the destination, the cpsr is automatically restored from the spsr

```
handler
      SUB     r14, r14, #4; r14-=4
      …
      <handler code>
      …
      MOVS pc, r14        ; return
```

Using the interrupt stack

```
handler
      SUB     r14, r14, #4; r14-=4
      STMFD  r13!, {r0-r3, r14}   ; store context
      …
      <handler code>
      …
       LDMFD  r13!, {r0-r3, pc}^  ; return
```

The ^ symbol forces the cpsr to be restored from the spsr

Assigning Interrupts

- **When it comes to assigning interrupts, system designer have adopted a standard design practice**
  - **Software Interrupts are normally reserved to call privileged operating system routines**
  - **Interrupt Requests are normally assigned for general-purpose interrupts**
    - For example, a periodic timer interrupt to force a context switch tends to be an IRQ exception
    - The IRQ exception has a lower priority and higher interrupt latency
  - **Fast Interrupt Requests are normally reserved for a single interrupt source that requires a fast response time**
    - For example, direct memory access specifically used to move blocks of memory
    - The FIQ exception is used for a specific application, leaving the IRQ exception for more general operating system activities

Interrupt Latency

- Interrupt-driven embedded systems have to fight a battle with interrupt latency
  - Interrupt latency is the interval of time from an external interrupt request signal being raised to the first fetch of an instruction of a specific interrupt service routine (ISR)

- Interrupt latency depends on a combination of hardware and software
  - System architects must balance the system design to handle multiple simultaneous interrupt sources and minimize interrupt latency
  - If the interrupts are not handled in a timely manner, then the system will exhibit slow response times

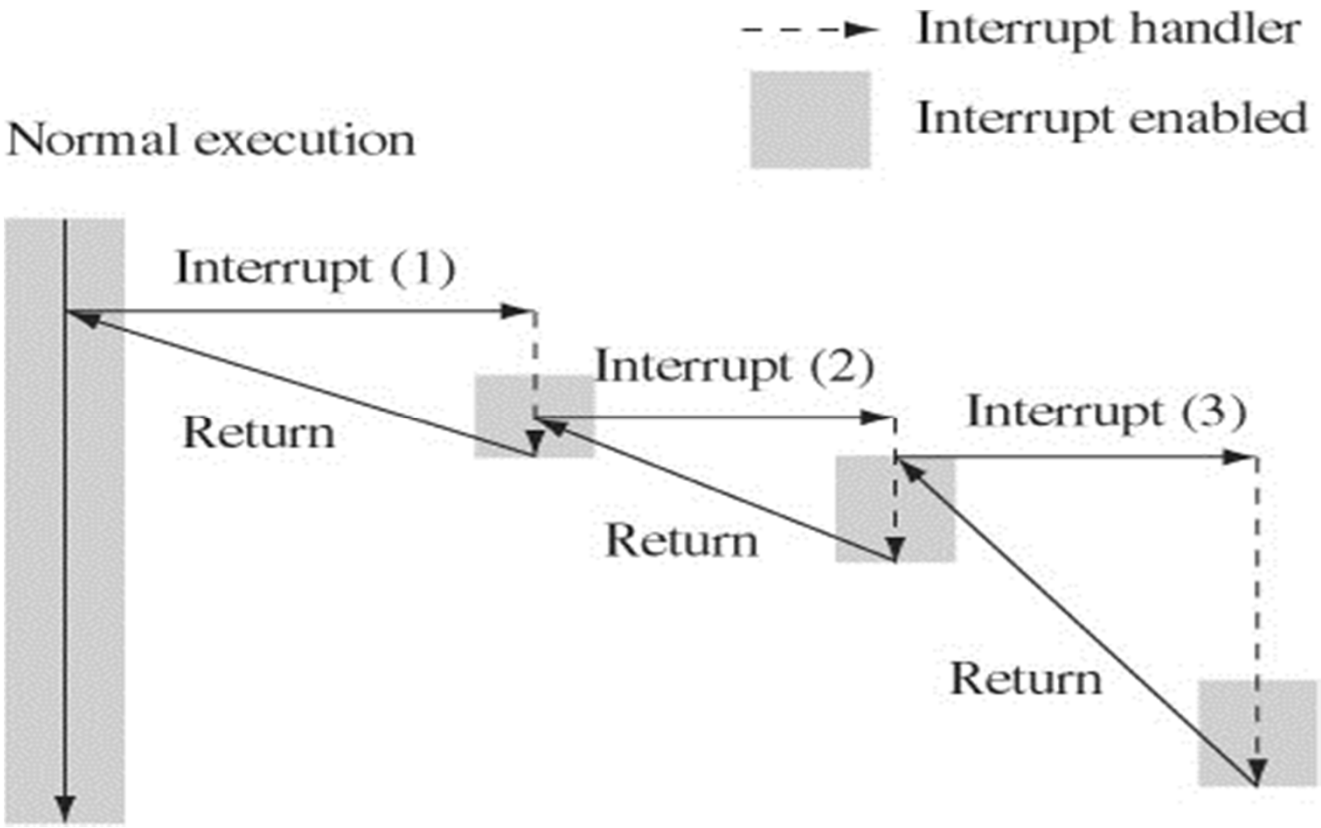
Interrupt Latency

- Software handlers have two main methods to minimize interrupt latency
  - Nested interrupt handler
  - Prioritization

- Nested interrupt handler allows further interrupts to occur even when currently servicing an existing interrupt
  - This is achieved by reenabling the interrupts as soon as the interrupt source has been serviced (so it won't generate more interrupts) but before the interrupt handling is complete
  - Once a nested interrupt has been serviced, then control is relinquished to the original interrupt service routine

Interrupt Latency

- A three-level nested interrupt

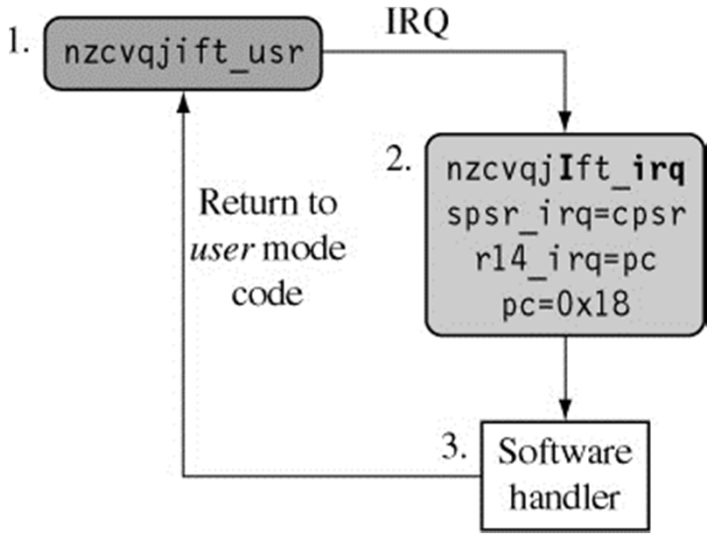


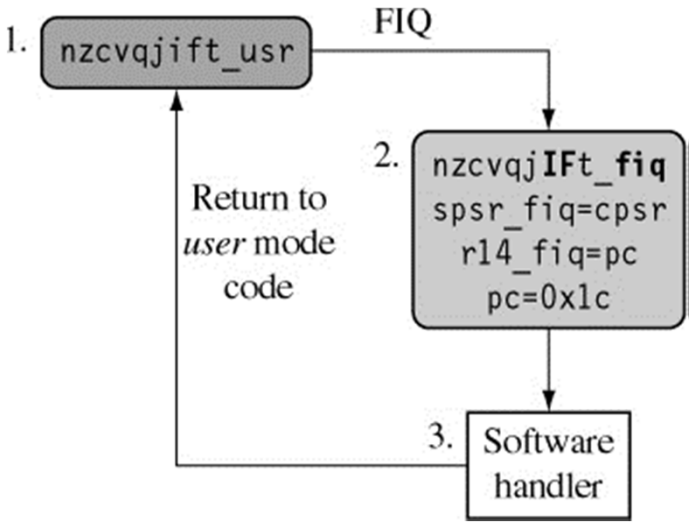
Interrupt Latency

- **Prioritization is to program the interrupt controller to ignore interrupts of the same or lower priority than the interrupt you are handling**
  - **Only a higher-priority task can interrupt your handler, and you then reenable interrupts**
  - **Higher-priority interrupts have a lower average interrupt latency than the lower-priority interrupts**

IRQ and FIQ Exceptions

❒ IRQ and FIQ examples





- **Changing to FIQ mode means there is no requirement to save registers r8 to r12 since these registers are banked in FIQ mode**
- **These registers can be used to hold temporary data, such as buffer pointers or counters**

IRQ and FIQ Exceptions

❒ Enabling and Disabling FIQ and IRQ Exceptions

- The ARM processor core has a simple procedure to manually enable and disable interrupts that involves modifying the cpsr when the processor is in a privileged mode

Enabling an interrupt.

cpsr value	IRQ	FIQ
Pre	*nzcvqjIFt_SVC*	*nzcvqjIFt_SVC*
Code	`enable_irq`	`enable_fiq`
	`MRS r1, cpsr`	`MRS r1, cpsr`
	`BIC r1, r1, #0x80`	`BIC r1, r1, #0x40`
	`MSR cpsr_c, r1`	`MSR cpsr_c, r1`
Post	*nzcvqjiFt_SVC*	*nzcvqjIft_SVC*

Disabling an interrupt.

cpsr	IRQ	FIQ
Pre	*nzcvqjift_SVC*	*nzcvqjift_SVC*
Code	`disable_irq`	`disable_fiq`
	`MRS r1, cpsr`	`MRS r1, cpsr`
	`ORR r1, r1, #0x80`	`ORR r1, r1, #0x40`
	`MSR cpsr_c, r1`	`MSR cpsr_c, r1`
Post	*nzcvqjIft_SVC*	*nzcvqjiFt_SVC*

- The first instruction MRS copies the contents of the cpsr into register r1
- The second instruction clears the IRQ or FIQ mask bit
- The third instruction then copies the updated contents in register r1 back into the cpsr

Basic Interrupt Stack Design and Implementation

- **Exceptions handlers make extensive use of stacks, with each mode having a dedicated register containing the stack pointer**

  왜 user랑 stack 동일하게 사용하지 않는가? user stack 상태를 알지 못하므로 stack overflow 발생 가능

- **The design of the exception stacks depends upon the following factors**
  - **Operating system requirements: each operating system has its own requirements for stack design**
  - **Target hardware: the target hardware provides a physical limit to the size and positioning of the stack in memory**

Basic Interrupt Stack Design and Implementation

- **Two design decisions need to be made for stacks**
  - **The location determines where in the memory map the stack begins**
    - Most ARM-based systems are designed with a stack that descends downwards with the top of the stack at a high memory address
  - **Stack size depends upon the type of handler, nested or nonnested**
    - A nested interrupt handler requires more memory space since the stack will grow with the number of nested interrupts

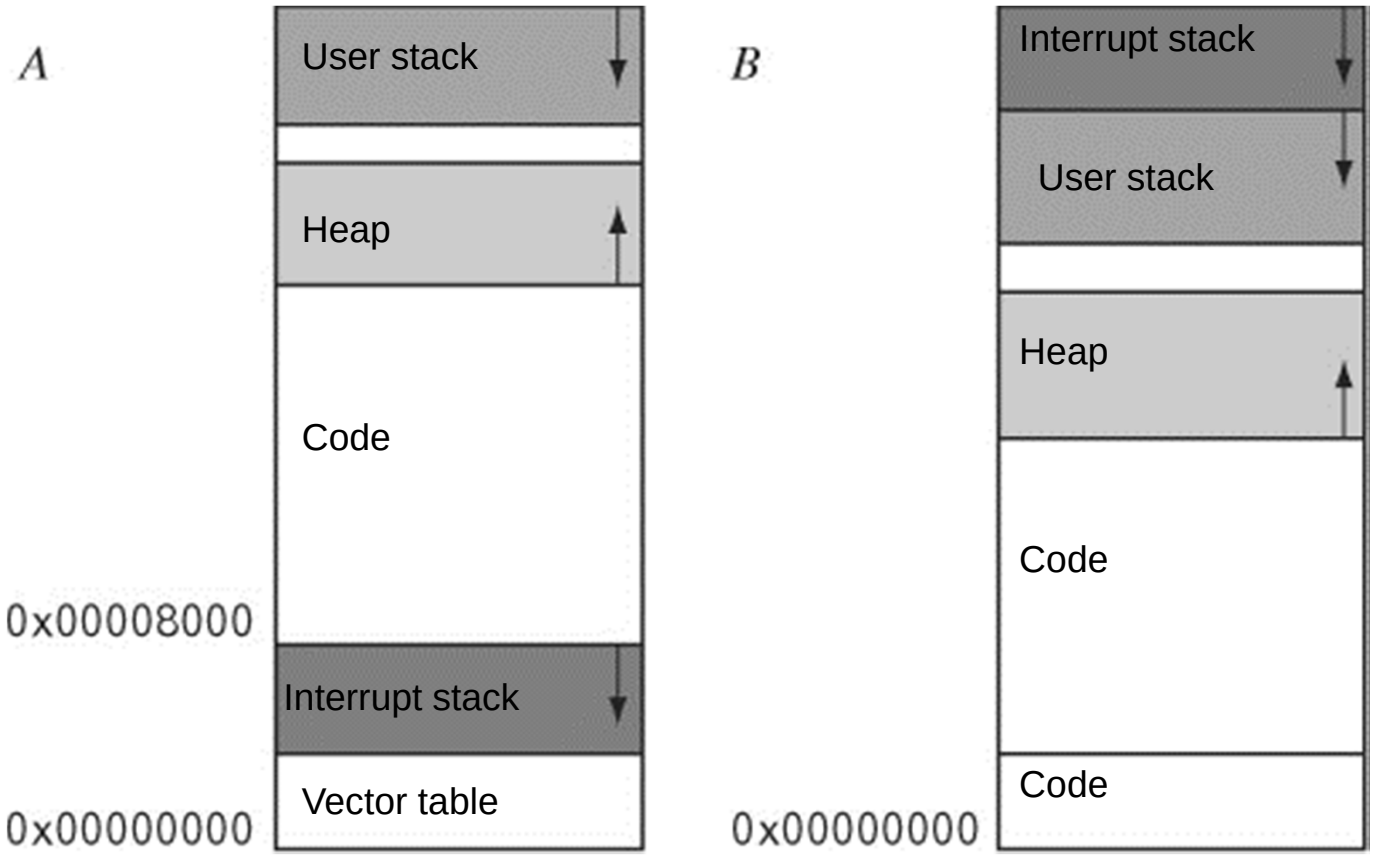
Basic Interrupt Stack Design and Implementation

- A good stack design tries to avoid stack overflow—where the stack extends beyond the allocated memory—because it causes instability in embedeed systems
  - There are software techniques that identify overflow and that allow corrective measures to take place to repair the stack before irreparable memory corruption occurs
  - The two main methods are (1) to use memory protection and (2) to call a stack check function at the start of each routine

Basic Interrupt Stack Design and Implementation

- The IRQ mode stack has to be set up before interrupts are enabled—normally in the initialization code for the system
  - It is important that the maximum size of the stack is known in a simple embedded system, since the stack size is reserved in the initial stages of boot-up by the firmware
- The following figures show two typical layouts in a linear address space
  - A: Interrupt stack stored underneath the code segment
  - B: Interrupt stack at the top of the memory above the user stack
  - The main advantage of B over A is that B does not corrupt the vector table when a stack overflow occurs, and so the system has a chance to correct itself when an overflow has been identified

Basic Interrupt Stack Design and Implementation



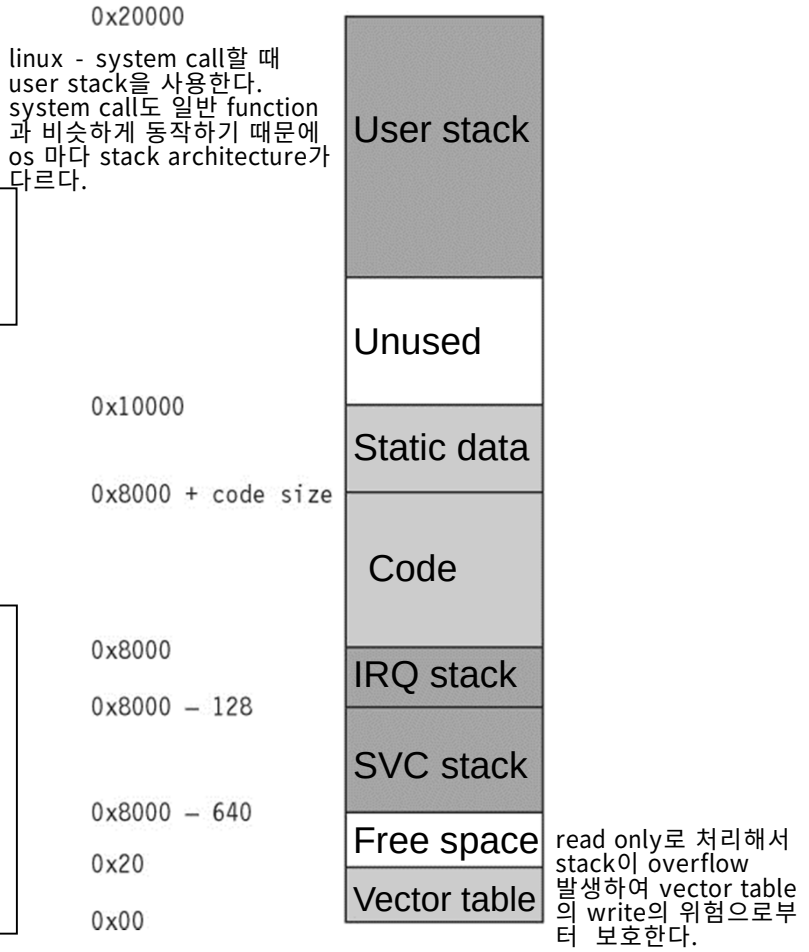
Basic Interrupt Stack Design and Implementation

- For each processor mode a stack has to be set up
  - This is carried out every time the processor is reset

```
USR_Stack   EQU 0x20000
IRQ_Stack   EQU 0x8000
SVC_Stack   EQU IRQ_Stack-128
```

  - We can declare set of defines that map each processor mode with a particular mode bit pattern

```
Usr32md     EQU 0x10        ; User mode
FIQ32md     EQU 0x11        ; FIQ mode
IRQ32md     EQU 0x12        ; IRQ mode
SVC32md     EQU 0x13        ; Supervisor mode
Abt32md     EQU 0x17        ; Abort mode
Und32md     EQU 0x1b        ; Undefined instruction mode
Sys32md     EQU 0x1f        ; System mode
NoInt       EQU 0xc0        ; Disable interrupts
```



Basic Interrupt Stack Design and Implementation

- **Initialization code starts by setting up the stack registers for each processor mode**
  - **The stack register r13 is one of the registers that is always banked when a mode change occurs**
  - **The code first initializes the IRQ stack**
  - **For safety reasons, it is always best to make sure that interrupts are disabled by using a bitwise OR between NoInt and the new mode**

Basic Interrupt Stack Design and Implementation

- Here is an example of how to set up three different stacks when the processor core comes out of reset
  - Using separate stacks for each mode rather than using a single stack has one main advantage: errant tasks can be debugged and isolated from the rest of the system

- Supervisor mode stack
  - The processor core starts in supervisor mode so the SVC stack setup involves loading register r13_svc with the address pointed to by SVC_NewStack

```
    LDR          r13, SVC_NewStack          ;r13_svc
    …
SVC_NewStack
    DCD    SVC_Stack
```

Basic Interrupt Stack Design and Implementation

❒ IRQ mode stack

- To set up the IRQ stack, the processor mode has to change to IRQ mode
- This is achieved by storing a cpsr bit pattern into register r2
- Register r2 is then copied into the cpsr, placing the processor into IRQ mode
- This action immediately makes register r13_irq viewable, and it can then be assigned the IRQ_Stack value

```
    MOV         r2, #NoInt|IRQ32md
    MSR         cpsr_c,r2
    LDR         r13, IRQ_NewStack          ;r13_irq
    …
IRQ_NewStack
    DCD   IRQ_Stack
```

Basic Interrupt Stack Design and Implementation

❒ User mode stack

- It is common for the user mode stack to be the last to be set up because when the processor is in user mode there is no direct method to modify the cpsr
- An alternative is to force the processor into system mode to set up the user mode stack since both modes share the same registers

```
    MOV         r2, #NoInt|Sys32md
    MSR         cpsr_c,r2
    LDR         r13, User_NewStack          ;r13_usr
    …
USR_NewStack
    DCD   USR_Stack
```

Nonnested Interrupt Handler

- The simplest interrupt handler is a handler that is nonnested: interrupts are disabled until control is returned back to the interrupted task or process
  - Because this can only service a single interrupt, handlers of this form are not suitable for complex embedded systems that service multiple interrupts with differing priority levels

- Processing stages
  1. Disable interrupts
  2. Save context
  3. Interrupt handler
  4. Interrupt service routine
  5. Restore context
  6. Enable interrupts

Nonnested Interrupt Handler

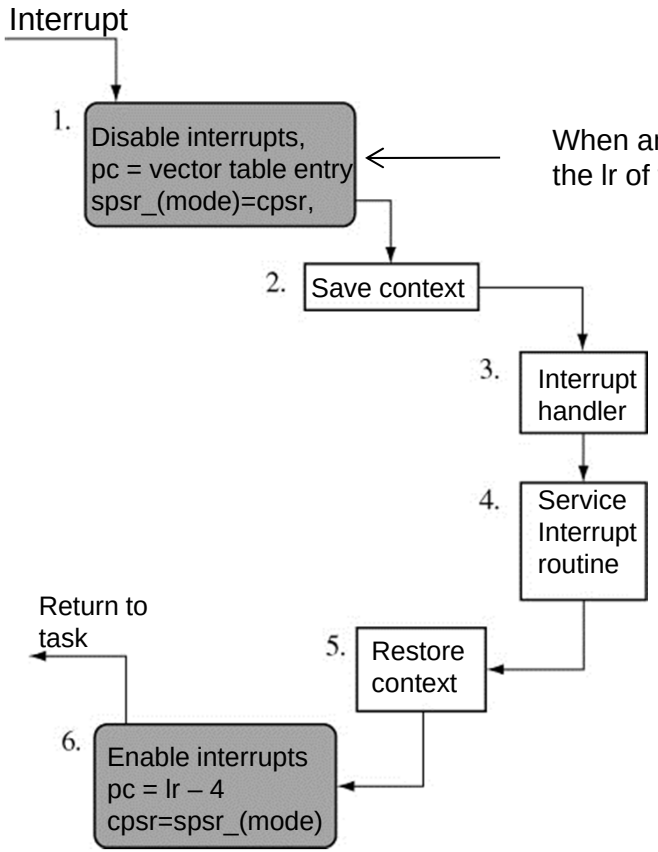
1. **Disable interrupts**
   - When the IRQ exception is raised, the ARM processor will disable further IRQ exceptions from occurring
   - The processor mode is set to the appropriate interrupt request mode, and the previous cpsr is copied into the newly available spsr_irq
   - The processor will then set the pc to point to the correct entry in the vector table and execute the instruction
   - This instruction will alter the pc to point to the specific interrupt handler
2. **Save context**
   - On entry the handler code saves a subset of the current processor mode nonbanked registers

Nonnested Interrupt Handler

3. **Interrupt handler** context 저장 & device 파악
   - The handler then identifies the external interrupt source and executes the appropriate interrupt service routine (ISR)
4. **Interrupt service routine** device마다의 특정 행동
   - The ISR services the external interrupt source and resets the interrupt
5. **Restore context**
   - The ISR returns back to the interrupt handler, which restores the context
6. **Enable interrupts**
   - The spsr_irq is restored back into the cpsr
   - The pc is then set to the next instruction after the interrupt was raised

Nonnested Interrupt Handler

❒ Simple nonnested interrupt handler



When an interrupt occurs, the processor also saves the pc to the lr of the IRQ mode and sets the cpsr to the IRQ mode

Nonnested Interrupt Handler

```
Interrupt_handler
    SUB  r14, r14, #4                  ; adjust lr (return address)
    STMFD  r13!, {r0-r3, r12, r14}     ; save context
    <interrupt service routine>        ; service interrupt
    LDMFD r13!, {r0-r3, r12, pc}^      ; restore context, return, and
                                         restore cpsr (enable interrupt)
```

- The first instruction sets the link register r14_irq to return back to the correct location in the interrupted task or process
- The STMFD instruction saves the context by placing a subset of the registers onto the stack
  - To reduce interrupt latency we save a minimum number of registers
- The LDMFD instruction restores the context and return from the interrupt handler
  - The ^ means that the cpsr will be restored from the spsr, which is only valid if the pc is loaded at the same time

Nested Interrupt Handler

- **Interrupts are nested if multiple interrupts may be handled concurrently**
  - **Makes system more responsive but harder to develop and validate**
  - **Often much harder!**

- **Only makes sense in combination with prioritized interrupt scheduling**
  - **Nesting w/o prioritization increases latency without increasing responsiveness!**

Prioritizing Interrupts

- Really easy on some hardware
  - E.g. x86 masks all interrupts >= current interrupt
- On other hardware not so easy
  - E.g. on ARM and AVR need to manually mask out lower priority interrupts before reenabling interrupts
- Important: When an interrupt controller has interrupt priorities, this usually refers to how it selects among multiple pending interrupts
  - != prioritized interrupt scheduling