RV Institute of Technology and Management®

MODULE 4
Exception and Interrupt Handling

Exception and Interrupt Handling: Exception handling, ARM processor exceptions and modes, vector table, exception priorities, link register offsets, interrupts, assigning interrupts, interrupt latency, IRQ and FIQ exceptions, basic interrupt stack design and implementation.

Firmware: Firmware and bootloader, ARM firmware suite, Red Hat redboot, Example: sandstone, sandstone directory layout, sandstone code structure.

Textbook 1: Chapter 9.1 and 9.2, Chapter 10

Exception and Interrupt Handling

- *Exception handling.* Exception handling covers the specific details of how the ARM processor handles exceptions.
- *Interrupts.* ARM defines an interrupt as a special type of exception. This section discusses the use of interrupt requests, as well as introducing some of the common terms, features, and mechanisms surrounding interrupt handling.
- *Interrupt handling schemes.* The final section provides a set of interrupt handling methods. Included with each method is an example implementation.

Exception handling

An exception is any condition that needs to halt the normal sequential execution of instructions

Examples : When the ARM core is reset

when an instruction fetch or memory access fails, when an undefined instruction is encountered,

when a software interrupt instruction is executed

when an external interrupt has been raised

Exception handling is the method of processing these exceptions.

- Most exceptions have an associated software *exception handler*—a software routine that executes when an exception occurs
- . The handler first determines the cause of the exception and then services the exception. Servicing takes place either within the handler or by branching to a specific service routine

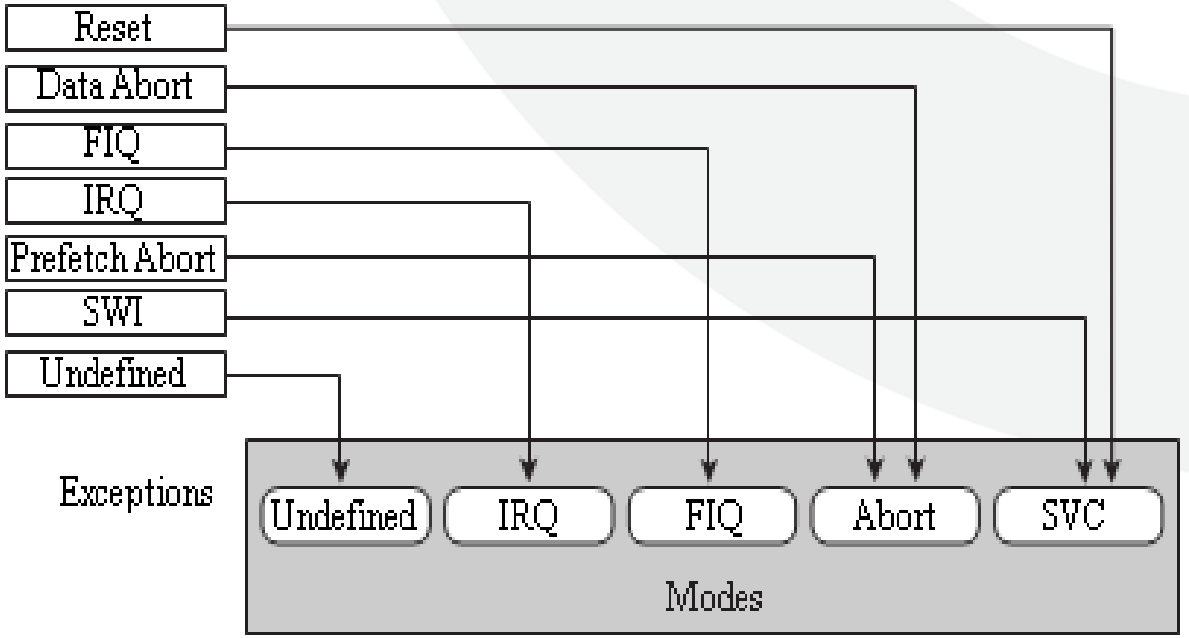
ARM processor exceptions and modes

- ARM processor exceptions and 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

Exceptions and associated 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 *pc* to the address of the exception handler



Vector table

Table 9.2 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

Vector table

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, #-0xff0]`—This *load register instruction* loads a specific interrupt ser- vice routine address from address `0xffffff030` to the *pc*. This specific instruction 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*. It lets you span the full address space but at limited alignment. The address must be an 8-bit immediate rotated right by an even number of bits.

Exception priorities

- Exceptions can occur simultaneously, so the processor has to adopt a priority mechanism

Table 9.3	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	—

Exception priorities

- The Reset exception is the highest priority exception and is always taken whenever it is signaled. 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.
- 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.
- 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.
- An Interrupt Request (IRQ) exception occurs when an external peripheral sets the IRQ pin to *nIRQ*. An IRQ exception is the second-highest priority interrupt. The IRQ handler will be entered if neither an FIQ exception nor Data Abort exception occurs

Exception priorities

- A Prefetch Abort exception occurs when an attempt to fetch an instruction results in a memory fault.
- A Software Interrupt (SWI) exception occurs when the SWI instruction is executed and none of the other higher-priority exceptions have been flagged.
- 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.

Link register offsets.

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

Link register offsets.

- When an exception occurs, the link register is set to a specific address based on the current *pc*. For instance, 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.

EXAMPLE

- 9.2

This example shows that a typical method of returning from an IRQ and FIQ handler is to use a SUBS instruction:

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

- Because there is an S at the end of the SUB instruction and the *pc* is the destination register, the *cpsr* is automatically restored from the *spsr* register.

Interrupts

There are two types of interrupts available on the ARM processor.

The first type of interrupt causes an exception raised by an external peripheral—namely, IRQ and FIQ.

The second type is a specific instruction that causes an exception—the SWI instruction. Both types suspend the normal flow of a program.

- In this section we will focus mainly on IRQ and FIQ interrupts. We will cover these topics:

  - Assigning interrupts
  - Interrupt latency
  - IRQ and FIQ exceptions
  - Basic interrupt stack design and implementation

Interrupts-

ASSIGNING INTERRUPTS

- A system designer can decide which hardware peripheral can produce which interrupt request. This decision can be implemented in hardware or software (or both) and depends upon the embedded system being used.

- An *interrupt controller* connects multiple external interrupts to one of the two ARM interrupt requests. Sophisticated controllers can be programmed to allow an external interrupt source to cause either an IRQ or FIQ exception.

standard design practice:

- Software Interrupts are normally reserved to call privileged operating system routines. For example, an SWI instruction can be used to change a program running in *user* mode to a privileged mode.

- 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 than the FIQ exception.

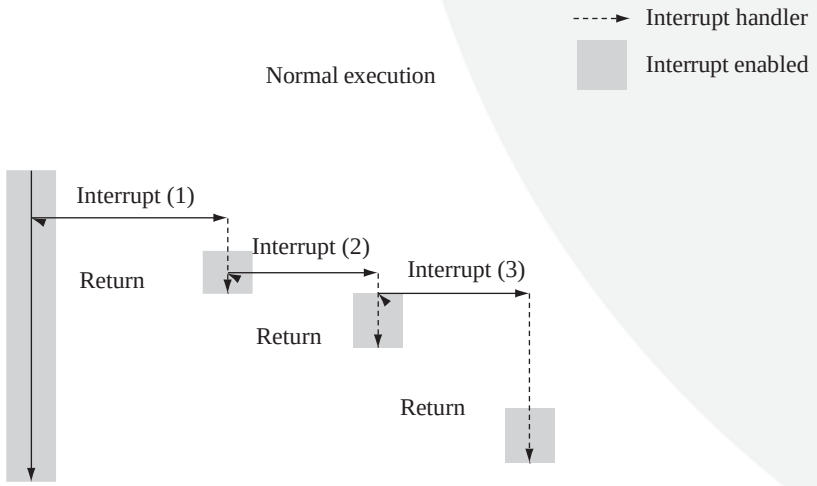
- Fast Interrupt Requests are normally reserved for a single interrupt source that requires a fast response time—for example, direct memory access specifically used to move blocks of memory. Thus, in an embedded operating system design, the FIQ exception is used for a specific application, leaving the IRQ exception for more general operating system activities.

INTERRUPT LATENCY

- *interrupt latency*—the inter- val 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.
- If the interrupts are not handled in a timely manner, then the system will exhibit slow response times.
- Software handlers have two main methods to minimize interrupt latency. The first method is to use a *nested interrupt handler*, which 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 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

- The second method involves *prioritization*. You program the interrupt controller to ignore interrupts of the same or lower priority than the interrupt you are handling, so only a higher-priority task can interrupt your handler. You then reenable interrupts.
- The processor spends time in the lower-priority interrupts until a higher-priority inter- rupt occurs. Therefore higher-priority interrupts have a lower average interrupt latency than the lower-priority interrupts, which reduces latency by speeding up the completion time on the critical time-sensitive interrupts.

IRQ and FIQ exceptions

- IRQ and FIQ exceptions only occur when a specific interrupt mask is cleared in the *cpsr*. The ARM processor will continue executing the current instruction in the execution stage of the pipeline before handling the interrupt—an important factor in designing a deterministic interrupt handler since some instructions require more cycles to complete the execution stage.

  procedure

  - The processor changes to a specific interrupt request mode, which reflects the interrupt being raised.
- The previous mode's *cpsr* is saved into the *spsr* of the new interrupt request mode.
- The *pc* is saved in the *lr* of the new interrupt request mode.
- *Interrupt/s are disabled*—either the IRQ or both IRQ and FIQ exceptions are disabled in the *cpsr*. This immediately stops another interrupt request of the same type being raised.
- The processor branches to a specific entry in the vector table.

Enabling and Disabling FIQ and IRQ Exceptions

- 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*, enabling the interrupt request

Enabling an interrupt.

cpsr value	IRQ	FIQ
Pre	*nzcvqj**I**Ft_SVC*	*nzcvqjI**F**t_SVC*

Code	`enable_irq`
	`MRS r1, cpsr`
	`BIC r1, r1, #0x80`
	`MSR cpsr_c, r1`
		`enable_fiq`
		`MRS r1, cpsr`
		`BIC r1, r1, #0x40`
		`MSR cpsr_c, r1`

Post	*nzcvqj**i**Ft_SVC*	
	*nzcvqjI**f**t_SVC*	

- It is important to understand that the interrupt request is either enabled or disabled only once the MSR instruction has completed the execution stage of the pipeline.
- Interrupts
- can still be raised or masked prior to the MSR completing this stage.
-

Basic interrupt stack design and implementation

- Exceptions handlers make extensive use of stacks, with each mode having a dedicated register containing the stack pointer. The design of the exception stacks depends upon these 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.
- Two design decisions need to be made for the 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.

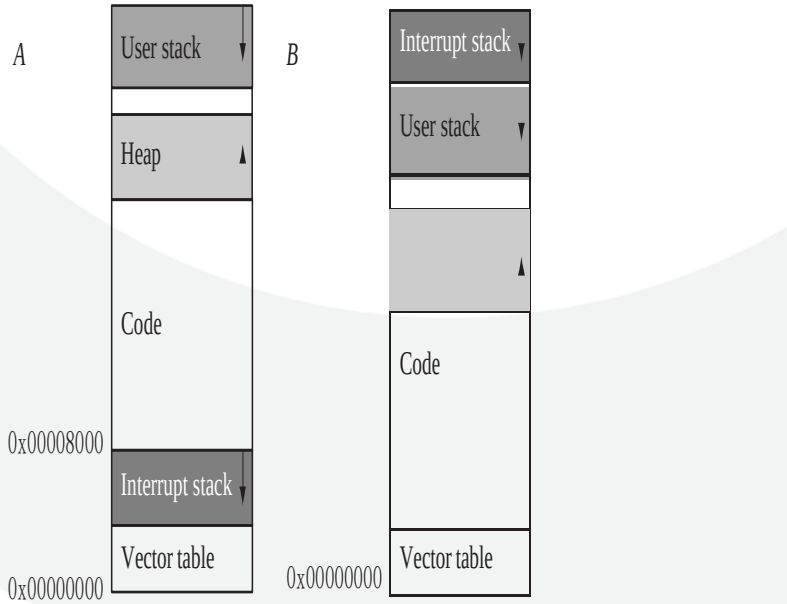
Basic interrupt stack design and implementation

- ***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`. For this example the value is `SVC_Stack`.

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

- ***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, #Sys32md
MSR     cpsr_c, r2
LDR     r13, USR_NewStack          ; r13_usr
...
```

Firmware: Firmware and bootloader

- **The *firmware* is the deeply embedded, low-level software that provides an interface between the hardware and the application/operating system level software. It resides in the ROM and executes when power is applied to the embedded hardware system**
- **Key Responsibilities:**

1. **Hardware Initialization**: Firmware initializes the microcontroller's hardware components, including clocks, GPIOs, timers, UARTs, SPI/I2C interfaces, ADCs, and other peripherals.
2. **Application Logic**: Firmware executes the main application logic, which includes tasks such as data processing, control algorithms, user interface interactions, and communication protocols.
3. **Interrupt Handling**: Firmware manages interrupts triggered by events such as timer overflows, GPIO changes, serial communication, and external signals. It defines interrupt service routines (ISRs) to handle these events.
4. **Peripheral Interaction**: Firmware interacts with peripherals to read sensor data, control actuators, communicate with external devices, and manage hardware resources efficiently.
5. **Error Handling and Recovery**: Firmware implements error detection, recovery mechanisms, and fault handling to maintain system reliability and robustness.

- The *bootloader* is a small application that installs the operating system or application onto a hardware target

  - Debug capabiliy is provided in the form of a module or monitor that provides software assistance for debugging code running on a hardware target. This assistance includes the following:

    - Setting up breakpoints in RAM. A breakpoint allows a program to be interrupted and the state of the processor core to be examined.
    - Listing and modifying memory (using peek and poke operations).
    - Showing current processor register contents.
    - Disassembling memory into ARM and Thumb instruction mnemonics.

- **Key Responsibilities**:

1. **System Initialization**: The bootloader initializes basic system settings, such as the stack pointer and memory configuration, before loading the firmware.
2. **Firmware Loading**: The bootloader reads the main firmware (stored in non-volatile memory) and copies it into the microcontroller's RAM.
3. **Hardware Initialization**: The bootloader sets up essential hardware configurations required for the firmware to run properly, such as clock settings and peripheral initialization.
4. **Boot Process Management**: The bootloader manages the boot process, including determining which firmware image to load (e.g., selecting from multiple firmware versions, performing checks for integrity or security), and transferring control to the firmware.
5. **Recovery and Maintenance**: Bootloaders often include features for system recovery and maintenance, such as firmware update capabilities (e.g., over-the-air updates), system diagnostics, and recovery from critical errors.

ARM firmware suite

- ARM has developed a firmware package called the ARM Firmware Suite (AFS). AFS is designed purely for ARM-based embedded systems. It provides support for a number of boards and processors including the Intel XScale and StrongARM processors. The pack- age includes two major pieces of technology, a Hardware Abstraction Layer called *µ*HAL (pronounced micro-HAL) and a debug monitor called Angel.
    - *µ*HAL supports these main features:
        - *System initialization*—setting up the target platform and processor core. Depending upon the complexity of the target platform, this can either be a simple or complicated task.
        - *Polled serial driver*—used to provide a basic method of communication with a host.
        - *LED support*—allows control over the LEDs for simple user feedback. This provides an application the ability to display operational status.
        - *Timer support*—allows a periodic interrupt to be set up. This is essential for preemptive context switching operating systems that require this mechanism.
        - *Interrupt controllers*—support for different interrupt controllers.

ARM firmware suite

- RedBoot is a firmware tool developed by Red Hat. It is provided under an open source license with no royalties or up front fees. RedBoot is designed to execute on different CPUs (for instance, ARM, MIPS, SH, and so on). It provides both debug capability through GNU Debugger (GDB), as well as a bootloader. The RedBoot software core is based on a HAL.
  - RedBoot supports these main features:
    - *Communication*—configuration is over serial or Ethernet. For serial, X-Modem proto- col is used to communicate with the GNU Debugger (GDB). For Ethernet, TCP is used to communicate with GDB. RedBoot supports a range of network standards, such as *bootp*, *telnet*, and *tftp*.
- *Flash ROM memory management*—provides a set of filing system routines that can download, update, and erase images in flash ROM. In addition, the images can either be compressed or uncompressed.
- *Full operating system support*—supports the loading and booting of Embedded Linux, Red Hat eCos, and many other popular operating systems. For Embedded Linux, RedBoot supports the ability to define parameters that are passed directly to the kernel upon booting.

Sandstone

- carries out only the following tasks: set up target platform environment, load a bootable image into memory, and relinquish control to an operating system
- SANDSTONE DIRECTORY LAYOUT