

MODULE-4

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

RBT: L1, L2, L3

Introduction:

Exceptions and interrupts are unexpected events which will disrupt the normal flow of execution of instruction. An exception is an unexpected event from within the processor. Interrupt is an unexpected event from outside the process. Whenever an exception or interrupt occurs, the hardware starts executing the code that performs an action in response to the exception.

The following types of action can cause an exception:

- *Reset* is called by the processor when power is applied. This instruction branches to the initialization code.
- *Undefined instruction* is used when the processor cannot decode an instruction.
- *Software interrupt* is called when we execute a SWI instruction. The SWI instruction is frequently used as the mechanism to invoke an operating system routine.
- *Prefetch abort* occurs when the processor attempts to fetch an instruction from an address without the correct access permissions. The actual abort occurs in the decode stage.
- *Data abort* is similar to a prefetch abort but is raised when an instruction attempts to access data memory without the correct access permissions.
- *Interrupt request* is used by external hardware to interrupt the normal execution flow of the processor. It can only be raised if IRQs are not masked in the cpsr.

4.1 Exception handling

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

Example for exceptions are: ARM core reset, instruction fetch or memory access failure, an undefined instruction fetch ,execution of software interrupt instruction ,when an external interrupt has been raised.

Exception handling is the method of processing these exceptions. Most exceptions have an associated software exception handler. Software exception handlers are software routines that execute 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.

The Reset exception is a special case of exception and it is used to initialize an embedded system.

4.1.1 ARM Processor Exceptions and Modes

Whenever an exception occurs, the core enters a specific mode. The ARM processor modes can be entered manually by changing the CPSR.

When an exception occurs, the ARM processor always switches to ARM state. Figure 4.1 shows exceptions and associated modes.

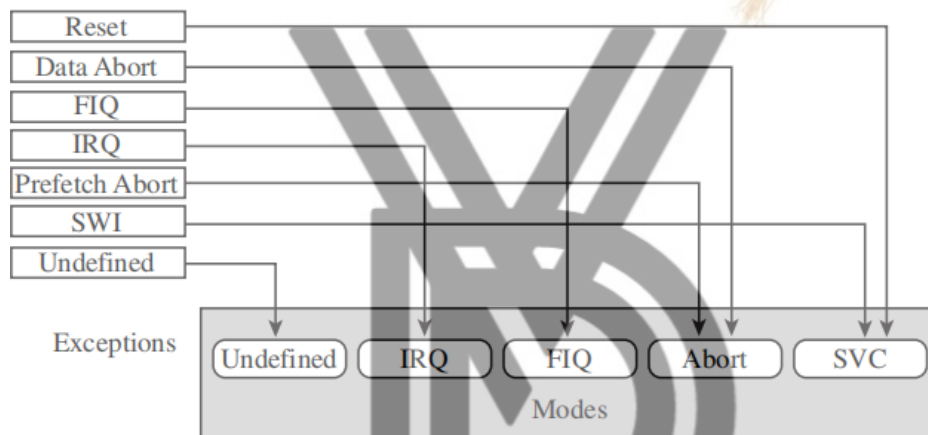


Figure 4.1 Exceptions and associated modes.

The **user** and **system** mode are the only two modes that are not entered by an exception.

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

4.1.2 Vector Table

The vector table is a table of addresses that the ARM core branches to when an exception is raised. These addresses contain branch instructions. The memory map address 0x00000000 is reserved for the vector table, a set of 32-bit words. On some processors the vector table can be optionally located at a higher address in memory (starting at the offset 0xffff0000).

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

Table 4.2 Vector table and processor modes

The branch instruction can be any 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. This form gives slight delay in branching as it need the extra memory access. But using this form we can branch to any address in memory.

LDR pc, [pc, #-0xff0]—This load register instruction loads a specific interrupt service routine address from address 0xfffff030 to the pc. This specific instruction is used when a vector interrupt controller is present (VIC PL190).

MOV pc, #immediate—This move instruction copies an immediate value into the pc. The address must be an 8-bit immediate rotated right by an even number of bits.

```

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]

```

Figure 4.2 Example vector table.

4.1.3 Exception Priorities

Exceptions can occur simultaneously, so the processor has to adopt a priority mechanism. Each exception is dealt with according to the priority level set out in Table 9.3.

Table 9.3 shows the various exceptions that occur on the ARM processor and their associated priority level.

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	—

Table 4.3 Exception priority levels.

- ◆ The Reset exception is the highest priority and it occurs when power is applied to the processor. The reset handler initializes the system, and setting up memory and caches. The reset handler must also set up the stack pointers for all processor modes. When a reset occurs, it takes precedence over all other exceptions.
- ◆ The lowest priority level is shared by two exceptions, the Software Interrupt and Undefined Instruction exceptions.
- ◆ As shown in the table Certain exceptions also disable interrupts by setting the I or F bits in the cpsr
- ◆ Code should be designed such that there is no exceptions or interrupts will occur during the first few instructions of the handler.
- ◆ Data Abort exceptions occur when the memory controller or MMU indicates that an invalid memory address has been accessed or when the current code attempts to read or write to memory without the correct access permissions. When Data Abort occurs, it takes precedence over all other exceptions except Reset exception.
- ◆ 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. Thus, no external source can interrupt the processor unless the IRQ and/or FIQ exceptions are reenabled by software.
- ◆ 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. On entry to the IRQ handler, the IRQ exceptions are disabled and should remain disabled until the current interrupt source has been cleared.
- ◆ 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 the higher exceptions have been raised. After enter 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.

- ◆ 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 *lr* and *spsr* must be stored away before branching to the nested SWI to avoid possible corruption of the link register and the *spsr*.
- ◆ 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.

4.1.4 Link Register Offsets

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

For example when an IRQ exception is raised, the link register *lr* points to the last executed instruction plus 8 because of three stage pipeline. 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 $lr + 4$.

Table 4.4 provides a list of useful addresses for the different exceptions.

Exception	Address	Use
Reset	—	<i>lr</i> 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	<i>lr</i>	points to the next instruction after the SWI instruction
Undefined Instruction	<i>lr</i>	points to the next instruction after the undefined instruction

Table 4.4 Useful link-register-based addresses.

Example(1)

This show the 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

```

Since 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.

Example (2)

This example shows another method that subtracts the offset from the link register r14 at the beginning of the handler.

```

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

```

After servicing is complete, return to normal execution occurs by moving the link register r14 into the pc and restoring cpsr from the spsr.

Example (3)

The example uses the interrupt stack to store the link register. This method first subtracts an offset from the link register and then stores it onto 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

```

To return to normal execution, the LDM instruction is used to load the pc. The ^ symbol in the instruction forces the cpsr to be restored from the spsr.

4.2 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.

4.2.1 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 unit is used to connect multiple external interrupts to one of the two ARM interrupt requests either IRQ or FIQ.

The system designers will use a standard design practice to assigning interrupts.

- ◆ 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.
- ◆ IRQ Requests are normally assigned for general-purpose interrupts. 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.
- ◆ In an embedded operating system design, the FIQ exception is used for a specific application and the IRQ exception are used for more general operating system activities.

4.2.2 Interrupt Latency

It is the time interval, 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 designer 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.

Software handlers have two main methods to minimize interrupt latency.

- 1) Nested interrupt handler,
- 2) Prioritization.

Nested interrupt handler

Nested interrupt handler allows other interrupts to occur even when it is 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. Fig 4.3 shows the three level nested interrupt,

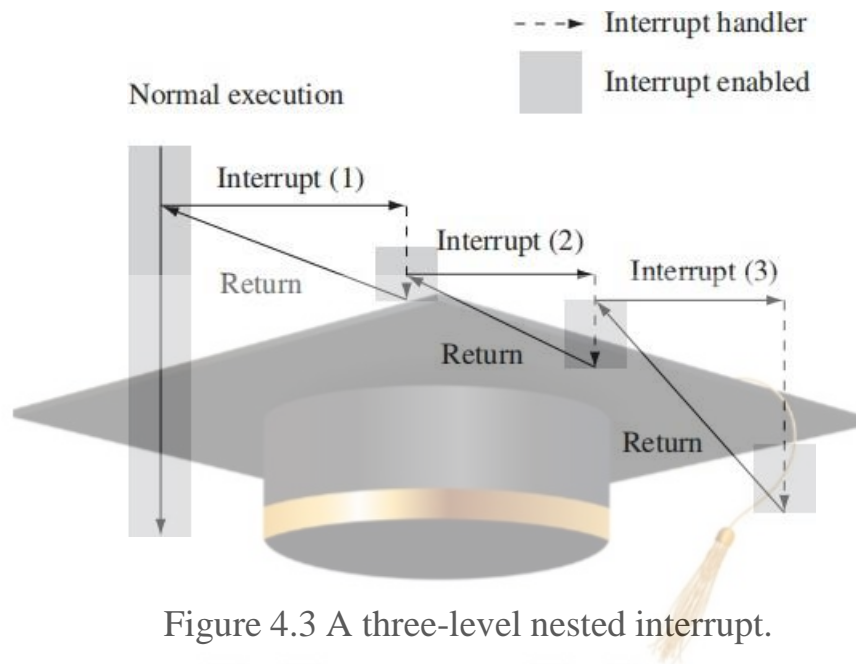


Figure 4.3 A three-level nested interrupt.

Prioritization

We can program the interrupt controller to ignore interrupts of the same or lower priority than the interrupt we are handling presently, so only a higher-priority task can interrupt our handler. We then re-enable the interrupts. The processor spends time in the lower-priority interrupts until a higher-priority interrupt occurs. Therefore higher-priority interrupts have a lower average interrupt latency than the lower-priority interrupts. It reduces latency by speeding up the completion time on the critical time-sensitive interrupts.

4.2.3 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 IRQ or FIQ exception causes the processor hardware to go through a standard procedure listed below,

- 1) The processor changes to a specific interrupt request mode, which being raised.
- 2) The previous mode's **cpsr** is saved into the **spsr** of the new interrupt request mode.
- 3) The **pc** is saved in the **lr** of the new interrupt request mode.
- 4) 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.

5) The processor branches to a specific entry in the vector table.

Example 4.5: what happens when an IRQ exception is raised when the processor is in user mode?

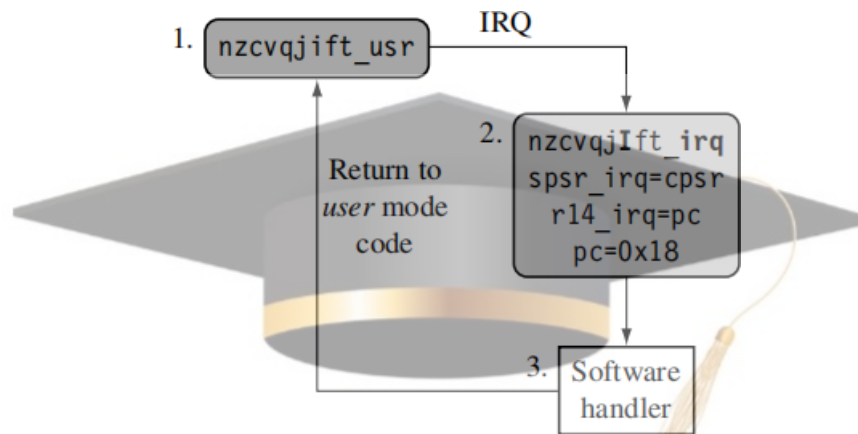


Figure 4.4 Interrupt Request (IRQ).

- i. The processor starts in state 1. In this mode both the IRQ and FIQ exception bits in the cpsr are enabled.
- ii. When an IRQ occurs the processor moves into state 2.->
 - a) This transition automatically sets the IRQ bit to one, disabling any further IRQ exceptions,
 - b) The FIQ exception remains enabled because FIQ has a higher priority and does not get disabled when a low-priority IRQ exception is raised,
 - c) The cpsr processor mode changes to IRQ mode,
 - d) The user mode cpsr is automatically copied into spsr_irq,
 - e) Register r14_irq is assigned the value of the pc when the interrupt was raised,
 - f) The pc is then set to the IRQ entry +0x18 in the vector table.
- iii. In state 3 the software handler takes over and calls the appropriate interrupt service routine to service the source of the interrupt. After completion, the processor mode reverts back to the original user mode code in state 1.

Example 4.6 what happens when an FIQ exception is raised when the processor is in user mode?

Figure 4.5 shows an example of an FIQ exception.

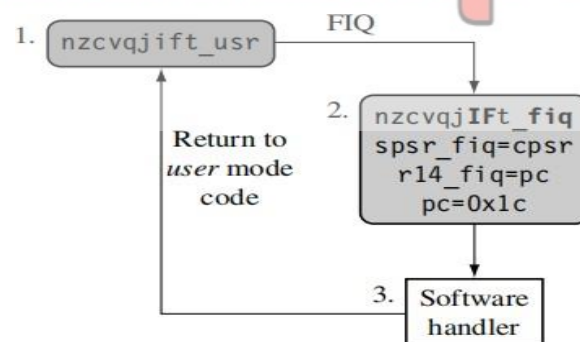


Figure 4.5 Fast Interrupt Request (FIQ).

- i) The processor starts in state 1. In this mode both the IRQ and FIQ exception bits in the cpsr are enabled.
- ii) When an FIQ occurs the processor moves into state 2.->
 - a) This transition automatically sets the IRQ bit and FIQ to one, disabling both IRQ and FIQ exceptions,
 - b) The cpsr processor mode changes to FIQ mode,
 - c) The user mode cpsr is automatically copied into spsr_fiq,
 - d) Register r14_fiq is assigned the value of the pc when the interrupt was raised,
 - e) The pc is then set to the FIQ entry +0x1c in the vector table.
- iii) In state 3 the software handler takes over and calls the appropriate interrupt service routine to service the source of the interrupt. After completion, the processor mode reverts back to the original user mode code in state 1.
- iv) When processor changes from user mode to FIQ mode, 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. This makes FIQ ideal for servicing a single-source, high-priority, low-latency interrupt.

4.2.3.1 Enabling and Disabling FIQ and IRQ Exceptions

The ARM processor core has a simple procedure to manually enable and disable interrupts by modifying the **cpsr** when the processor is in a privileged mode.

The procedure uses three ARM instructions.

- 1) The instruction MRS copies the contents of the **cpsr** into register **r1**.
- 2) The instruction BIC clears the IRQ or FIQ mask bit.
- 3) The instruction MSR then copies the updated contents in register **r1** back into the **cpsr**, to enable the interrupt request.

Table 4.5 shows how IRQ and FIQ interrupts are enabled.

The postfix **_c** identifies that the bit field being updated is the control field bit [7:0] of the cpsr.

<i>cpsr</i> value	IRQ	FIQ
Pre	<i>nzcqvjiFt_SVC</i>	<i>nzcqvjiFt_SVC</i>
Code	<i>enable_irq</i>	<i>enable_fiq</i>
	MRS r1, cpsr	MRS r1, cpsr
	BIC r1, r1, #0x80	BIC r1, r1, #0x40
	MSR cpsr_c, r1	MSR cpsr_c, r1
Post	<i>nzcqvjiFt_SVC</i>	<i>nzcqvjiFt_SVC</i>

Table 4.5 Enabling an interrupt.

Table 4.6 shows procedure to disable or mask an interrupt request.

cpsr	IRQ	FIQ
Pre	<i>nzcvqjift_SVC</i>	<i>nzcvqjift_SVC</i>
Code	<i>disable_irq</i>	<i>disable_fiq</i>
	MRS r1, cpsr	MRS r1, cpsr
	ORR r1, r1, #0x80	ORR r1, r1, #0x40
	MSR cpsr_c, r1	MSR cpsr_c, r1
Post	<i>nzcvqjift_SVC</i>	<i>nzcvqjiFt_SVC</i>

Table 4.6 Disabling an interrupt.

To enable and disable both the IRQ and FIQ exceptions, the immediate value on the data processing BIC or ORR instruction has to be changed to 0xc0.

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.

4.2.4 Basic Interrupt Stack Design and Implementation

Exceptions handlers uses the stacks to save the register contents. Each mode has 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: which 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.

stack overflow—when the stack extends beyond the allocated memory. It causes instability in embedded 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) use memory protection
- (2) call a stack check function at the start of each routine.

The IRQ mode stack has to be set up during the initialization code for the system. The stack size is reserved in the initial stages of boot-up. Figure 4.6 shows two memory layouts in a linear address space.

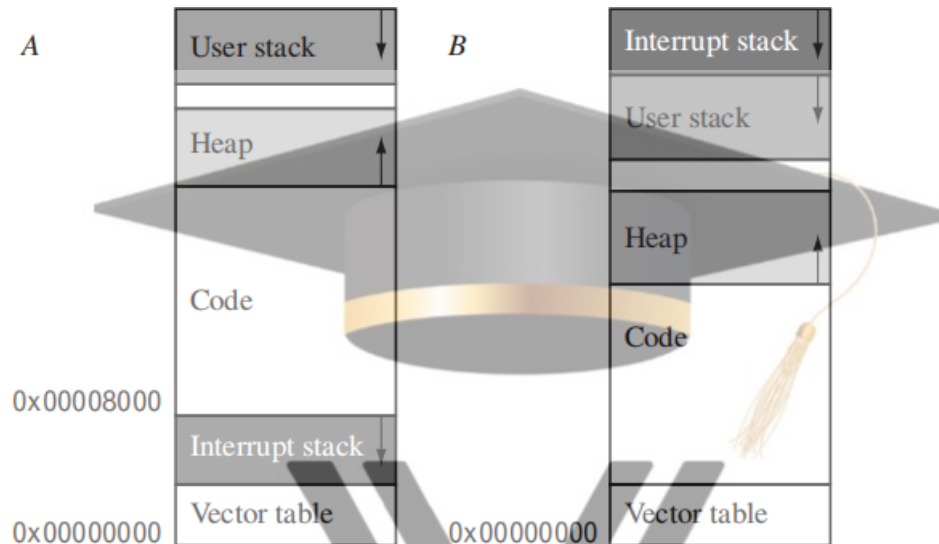


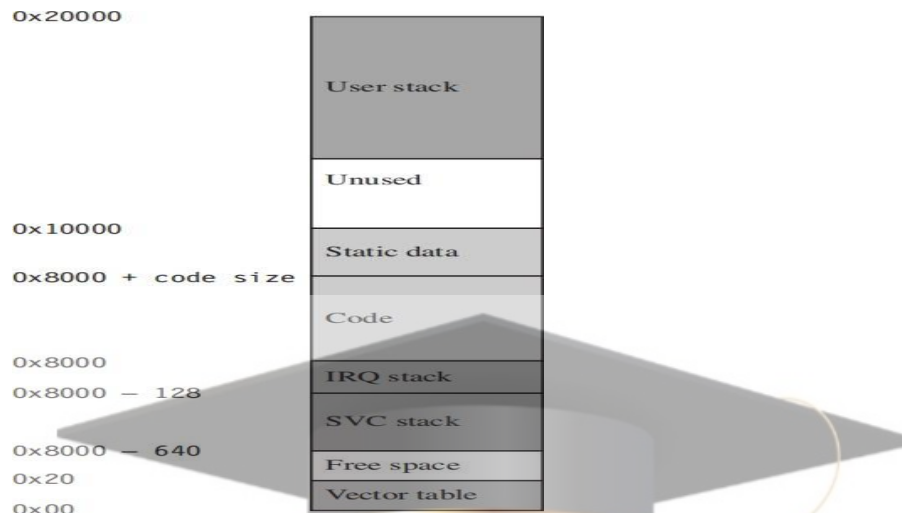
Figure 4.6 Memory layouts.

The first layout, A, shows a traditional stack layout with the interrupt stack stored underneath the code segment.

The second layout, B, shows the interrupt stack at the top of the memory above the user stack.

The main advantage of layout B over A is that Layout 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.

For each processor mode a stack has to be set up. This is carried out every time the processor is reset. Figure 9.7 shows an implementation of stack using layout A.



There is an advantage using separate stacks for each mode rather than using a single stack. Errant tasks can be debugged and isolated from the rest of the system.

Each mode stack must be set up. Here is an example to set up three different stacks when the processor core comes out of reset.

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.

A set of defines are declared that map the memory region names with an absolute address.

Example, the User stack is given the label **USR_Stack** and is set to address 0x20000. The Supervisor stack is set to an address that is 128 bytes below the IRQ stack.

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

A set of defines that map each processor mode with a particular mode bit pattern. These labels can then be used to set the **cpsr** to a new mode.

```
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
```

Example to set up 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
```

Example to set up IRQ mode stack:

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.

```
NoInt      EQU 0xc0          ; Disable interrupts
```

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.

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

Example to set up user mode stack:

The user mode stack will be set up last 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
...
USR_NewStack
DCD    USR_Stack
```