MODULE-4: Exception, Interrupt Handling and Firmware

Exception Handling

An **exception** is any condition that disrupts the normal sequential execution of instructions in a processor. This can occur due to various reasons, such as resets, instruction fetch failures, undefined instructions, software interrupts, or external interrupts. **Exception handling** refers to the mechanisms used to process these exceptions.

Key Topics Covered

1. ARM Processor Mode and Exceptions
2. Vector Table
3. Exception Priorities
4. Link Register Offsets

ARM Processor Exceptions and Modes:

ARM processors handle various exceptions that can disrupt normal execution. Each exception leads to the processor entering a specific mode. Understanding these exceptions and modes is crucial for effective system design and programming in ARM architecture.

ARM Processor Exceptions

ARM exceptions cause the processor to switch modes, automatically saving the current state. Each exception type has an associated mode that dictates the handling mechanism.

Exception Handling Mechanism

When an exception occurs:

- The **Current Program Status Register (CPSR)** is saved to the **Saved Program Status Register (SPSR)** of the exception mode.
- The **Program Counter (PC)** is saved to the **Link Register (LR)** of the exception mode.
- The processor enters **ARM state** and sets the PC to the address of the corresponding exception handler.

Manual Mode Entry

- User mode and System mode can be entered manually by modifying the CPSR. All other modes are entered via exceptions.
- Each mode serves a specific purpose related to system management and exception handling.

Dr. Devaraju B M, Dept. of CSE, RNSIT

- Entering FIQ and IRQ modes allows the processor to respond quickly to external events, while SVC is used for privileged operations within operating systems.

ARM Processor Exceptions and Associated Modes:

Exception Type	Mode	Main Purpose
Fast Interrupt Request (FIQ)	FIQ	Fast interrupt request handling
Interrupt Request (IRQ)	IRQ	General interrupt request handling
Software Interrupt (SWI) and Reset	SVC	Protected mode for operating systems
Prefetch Abort	ABORT	Handle memory protection and virtual memory issues
Data Abort	ABORT	Handle memory access violations
Undefined Instruction	UND	Software emulation of hardware coprocessors

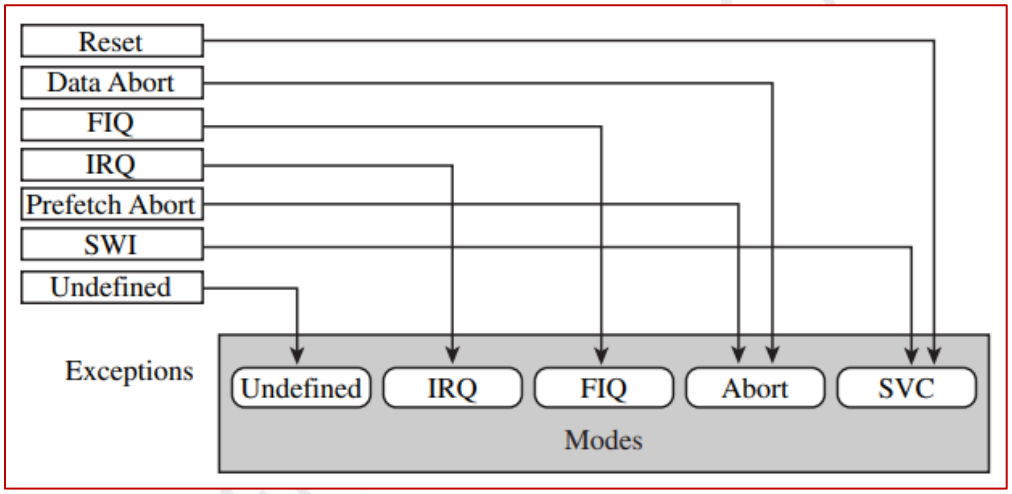


Figure: illustrates the simplified relationship between exceptions and their associated modes, highlighting the flow of control during exception handling.

Exception Handling Flow

1. **Exception Occurs:** The ARM processor detects an exception.
2. **Mode Switch:** The processor automatically switches to the corresponding exception mode.
3. **State Saving:**
   - CPSR is saved to SPSR of the exception mode.
   - PC is saved to LR of the exception mode.
4. **Handler Invocation:** The PC is set to the address of the exception handler for processing.

Vector Table:

The **vector table** is a crucial component of the ARM architecture, containing addresses that the ARM core branches to when an exception occurs. Understanding the vector table is essential for managing exceptions effectively in ARM-based systems.

Vector Table Structure

Each entry in the vector table corresponds to a specific exception type and is typically formatted as one of the following instructions:

Branch Instructions

1. **B <address>**
   - **Function:** Branch relative from the program counter (PC).
   - **Usage:** Directly jumps to a handler address.
2. **LDR pc, [pc, #offset]**
   - **Function:** Loads the handler address from memory to the PC.
   - **Usage:** Allows branching to any address in memory but incurs a slight delay due to an extra memory access. The address is an absolute 32-bit value stored near the vector table.
3. **LDR pc, [pc, #-0xff0]**
   - **Function:** Loads a specific interrupt service routine (ISR) address from a fixed memory location (e.g., `0xffffF030`).
   - **Usage:** Used when a Vector Interrupt Controller (VIC) is present (e.g., VIC PL190).
4. **MOV pc, #immediate**
   - **Function:** Copies an immediate value into the PC.
   - **Usage:** Can span the full address space, but the address must be an 8-bit immediate rotated right by an even number of bits.

Exception Type	Mode	Vector Table Offset	Instruction
Reset	SVC	+0x00	LDR pc, [pc, #reset]
Undefined Instruction	UND	+0x04	B undInstr
Software Interrupt (SWI)	SVC	+0x08	LDR pc, [pc, #swi]
Prefetch Abort	ABT	+0x0C	LDR pc, [pc, #prefetch]
Data Abort	ABT	+0x10	LDR pc, [pc, #data]
Not Assigned	—	+0x14	LDR pc, [pc, #notassigned]
IRQ	IRQ	+0x18	LDR pc, [pc, #irq]
FIQ	FIQ	+0x1C	LDR pc, [pc, #fiq]

Dr. Devaraju B M, Dept. of CSE, RNSIT

Example Vector Table:

```
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]
```

- The FIQ handler can be placed directly at the FIQ vector location, but it often uses the LDR instruction for consistency with other entries.
- Each branch instruction in the vector table allows the processor to jump to the designated handler for the specific exception.

Exception Priorities:

In ARM processors, exceptions can occur simultaneously, necessitating a priority mechanism to handle them effectively. Each exception has an assigned priority level that determines how it is managed.

Exception Priority Levels

Exception Type	Priority	I Bit	F Bit
Reset	1	1	1
Data Abort	2	1	—
Fast Interrupt Request (FIQ)	3	1	1
Interrupt Request (IRQ)	4	1	—
Prefetch Abort	5	1	—
Software Interrupt (SWI)	6	1	—
Undefined Instruction	6	1	—

1.Reset Exception

- **Priority:** Highest (1)
- **Handling:** Always taken when signaled. Initializes the system, sets up memory and caches, and establishes stack pointers for all modes.
- **Considerations:** External interrupts should be initialized before enabling IRQ or FIQ to prevent spurious interrupts.

2. Data Abort Exception

- **Priority:** 2
- **Handling:** Triggered by invalid memory access or permission issues. Can allow FIQ exceptions to occur during handling.
- **Considerations:** The handler must avoid further exceptions.

Dr. Devaraju B M, Dept. of CSE, RNSIT

3. Fast Interrupt Request (FIQ) Exception

- **Priority:** 3
- **Handling:** Triggered by external peripherals. The highest priority interrupt; disables IRQ on entry.
- **Considerations:** Must efficiently service the exception without being interrupted.

4. Interrupt Request (IRQ) Exception

- **Priority:** 4
- **Handling:** Triggered by external peripherals if no higher-priority exceptions are raised. IRQ disabled on entry.
- **Considerations:** Remains disabled until the interrupt source is cleared.

5. Prefetch Abort Exception

- **Priority:** 5
- **Handling:** Occurs on instruction fetch memory faults. IRQ disabled on entry, FIQ state remains unchanged.
- **Considerations:** If FIQ is enabled, it can be serviced during this exception.

6. Software Interrupt (SWI) Exception

- **Priority:** 6
- **Handling:** Occurs when SWI instruction is executed. Enters supervisor mode on entry.
- **Considerations:** Requires saving link register and SPSR if using nested SWI calls.

7. Undefined Instruction Exception

- **Priority:** 6
- **Handling:** Triggered by executing an unrecognized instruction. The processor queries coprocessors before raising the exception.
- **Considerations:** Shares priority with SWI, as they cannot occur simultaneously.
- Certain exceptions disable interrupts by setting the I or F bits in the CPSR to prevent further interruptions.
- The design of handlers for exceptions like FIQ, SWI, and IRQ must be efficient to ensure quick servicing and minimize latency.

Understanding exception priorities is critical for designing robust ARM systems. Proper handling based on priority levels ensures system stability and responsiveness in the face of unexpected events.

Link Register Offsets:

The link register (LR) is crucial for handling exceptions in ARM processors. It stores the return address when an exception occurs, pointing to specific addresses based on the current program counter (PC). Care must be taken to avoid corrupting the LR, as it is used to return from exception handlers.

Dr. Devaraju B M, Dept. of CSE, RNSIT

Link Register Behavior by Exception

Exception	LR Address Use
Reset	lr is not defined during a Reset
Data Abort	lr - 8 points to the instruction causing the Data Abort
FIQ	lr - 4 returns from the FIQ handler
IRQ	lr - 4 returns from the IRQ handler
Prefetch Abort	lr - 4 points to the instruction causing the Prefetch Abort
SWI	lr points to the next instruction after SWI
Undefined Instruction	lr points to the next instruction after the undefined instruction

Returning from Exception Handlers

Example 1: Using `SUBS`

A common method to return from an IRQ or FIQ handler is to use the SUBS instruction:

```
handler:
    <handler code>
    ...
            SUBS pc, r14, #4  ; pc = r14 – 4
```

The S at the end of SUB automatically restores the CPSR from SPSR when the PC is the destination register.

Example 2: Subtracting Offset from LR

Another approach is to subtract the offset from LR at the beginning of the handler:

Dr. Devaraju, Dept. of CSE, RNSIT

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

This moves the modified LR into the PC to resume execution and restores CPSR from SPSR.

Example 3: Using the Interrupt Stack

This method uses the interrupt stack to store the link register:

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

Dr. Devaraju B M Dept. of CSE, RNSIT

The LDM instruction loads the PC from the stack. The ^ symbol ensures CPSR is restored from SPSR.

Understanding link register offsets and how to manage them during exception handling is essential for maintaining system stability and ensuring correct program flow in ARM architecture. Proper handling allows the processor to return to normal execution after servicing exceptions without loss of context or data integrity.

Interrupts:

Interrupts are signals that temporarily suspend the normal flow of a program, allowing the processor to address important events. There are two main types of interrupts in ARM processors:

1. **External Interrupts:** IRQ (Interrupt Request) and FIQ (Fast Interrupt Request).
2. **Software Interrupts:** Triggered by the SWI (Software Interrupt) instruction.

Topics Covered

1. Assigning Interrupts
2. Interrupt Latency
3. IRQ and FIQ Exceptions
4. Basic Interrupt Stack Design and Implementation

1. Assigning Interrupts:

System designers determine which hardware peripherals generate specific interrupt requests. This can be done through hardware, software, or a combination of both, depending on the embedded system.

Interrupt Controller

An interrupt controller connects multiple external interrupts to the ARM's IRQ or FIQ. Advanced controllers can allow specific external sources to trigger either type of interrupt.

Standard Design Practices

- **Software Interrupts (SWI):** Reserved for privileged OS routines, such as switching from user mode to privileged mode.
- **Interrupt Requests (IRQ):** Generally assigned for standard interrupts (e.g., timer interrupts), with lower priority and higher latency compared to FIQ.
- **Fast Interrupt Requests (FIQ):** Reserved for time-sensitive tasks, such as direct memory access, which require immediate attention.

Dr. Devaraju B M, Dept. of CSE, RNSIT

2. Interrupt Latency

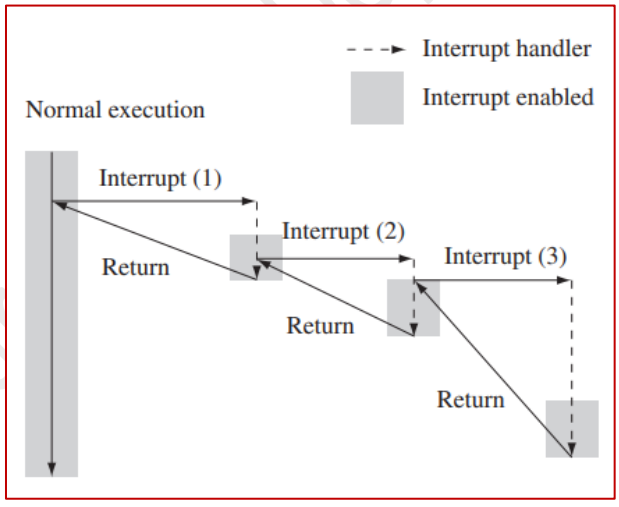
Interrupt latency is the time from when an interrupt request is raised to when the first instruction of the corresponding Interrupt Service Routine (ISR) is executed. Minimizing this latency is crucial for responsive embedded systems.

Factors Affecting Interrupt Latency

- **Hardware:** Design of the interrupt controller and processor architecture.
- **Software:** Efficiency of the interrupt handling routines and system design.

Strategies to Minimize Interrupt Latency

1. **Nested Interrupt Handlers:**
   - Allows higher-priority interrupts to be serviced while handling a current interrupt.
   - Interrupts are reenabled as soon as the source is serviced, facilitating quick responses to new interrupts (figure shows a three-level nested interrupt).
2. **Prioritization:**
   - Configure the interrupt controller to ignore lower-priority interrupts during the handling of a current interrupt.
   - This prioritization allows higher-priority interrupts to interrupt the current handler, reducing overall latency.



IRQ and FIQ Exceptions

IRQ (Interrupt Request) and FIQ (Fast Interrupt Request) exceptions are critical mechanisms in the ARM architecture that allow the processor to respond to external events. These exceptions are triggered only when specific interrupt mask bits in the CPSR (Current Program Status Register) are cleared.

Dr. Devaraju B M, Dept of CSE, RNSIT

Key Points

- **Execution Stage:** The ARM processor completes the current instruction before handling the interrupt, which is vital for deterministic interrupt handler design.
- **Interrupt Handling Procedure:**
    1. Transition to the appropriate interrupt request mode.
    2. Save the previous mode's CPSR to the SPSR (Saved Program Status Register) of the new mode.
    3. Save the program counter (PC) in the link register (LR) of the new mode.
    4. Disable interrupts (IRQ or both IRQ and FIQ).
    5. Branch to the appropriate entry in the vector table.

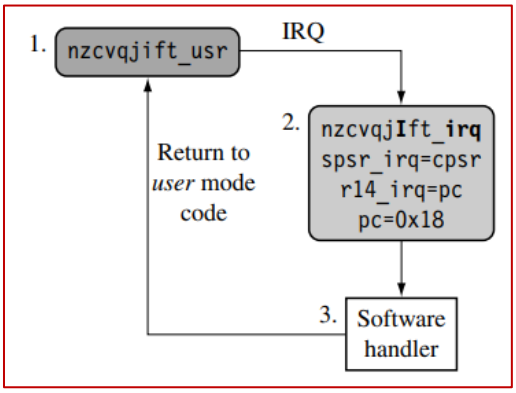
Handling IRQ Exceptions

Example 1: IRQ Exception Raised

1. **Initial State:** The processor is in user mode with both IRQ and FIQ enabled in the CPSR.
2. **Transition:** Upon an IRQ occurrence:
    - The processor enters IRQ mode.
    - The IRQ bit in the CPSR is set to 1, disabling further IRQ exceptions.
    - The FIQ bit remains enabled since FIQ has a higher priority.
    - The user mode CPSR is copied to SPSR_IRQ.
3. **Link Register:** The link register `r14_irq` is set to the PC value at the time of the interrupt.
4. **Vector Table:** The PC is updated to point to the IRQ entry in the vector table (offset +0x18).

Example Flow

- **State 1:** User mode (initial state).
- **State 2:** Transition to IRQ mode (handling the interrupt).
- **State 3:** Execute the IRQ handler and service the interrupt source.
- **Return:** After servicing, the processor reverts to the original user mode.



Dr. Devaraju B M, Dept of CSE, RNSIT

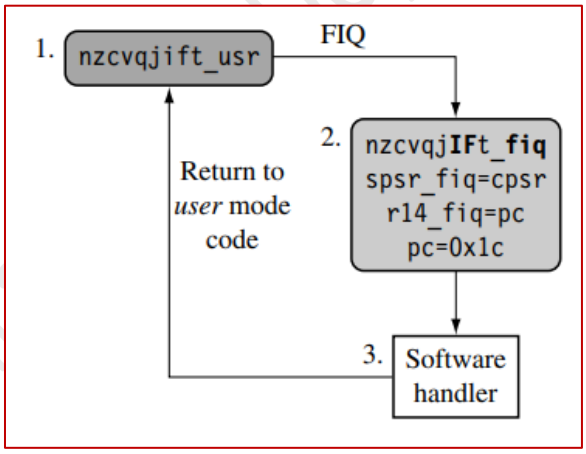
Handling FIQ Exceptions

Example 2: FIQ Exception Raised

1. **Initial State:** Similar to IRQ, the processor starts in user mode with both IRQ and FIQ enabled.
2. **Transition:** Upon an FIQ occurrence:
   - The processor enters FIQ mode.
   - The FIQ bit in the CPSR is set to 1, disabling further FIQ exceptions.
   - The IRQ bit remains enabled, allowing lower-priority interrupts to be masked.
   - The user mode CPSR is copied to SPSR_FIQ.
3. **Link Register:** The link register `r14_fiq` is set to the PC value at the time of the interrupt.
4. **Vector Table:** The PC is updated to point to the FIQ entry in the vector table (offset +0x1C).

Example Flow

- **State 1:** User mode (initial state).
- **State 2:** Transition to FIQ mode (handling the interrupt).
- **State 3:** Execute the FIQ handler and service the interrupt source.
- **Return:** After servicing, the processor returns to the original user mode.



IRQ and FIQ exceptions provide a structured way for the ARM processor to manage interrupts efficiently. The process involves changing processor modes, saving the state, and branching to the vector table, ensuring that the system can handle interrupts while maintaining the integrity of program execution.

Enabling and Disabling FIQ and IRQ Exceptions

The ARM processor provides a straightforward way to enable and disable interrupts by modifying the CPSR (Current Program Status Register) when in a privileged mode. Below are the procedures for enabling and disabling IRQ and FIQ interrupts.

Enabling Interrupts

Procedure

To enable IRQ and FIQ interrupts, follow these steps using the MRS, BIC, and MSR instructions:

1. **Copy CPSR to Register:**
   - Use the `MRS` instruction to copy the current CPSR to a general-purpose register (e.g., `r1`).
2. **Clear the IRQ or FIQ Mask Bit:**
   - Use the `BIC` instruction to clear the appropriate bit for IRQ (`0x80`) or FIQ (`0x40`).
3. **Update the CPSR:**
   - Use the `MSR` instruction to write the modified value back to CPSR.

Example Code for Enabling Interrupts:

Enabling an interrupt.

cpsr value	IRQ	FIQ
Pre	*nzcvqj**I**Ft_SVC*	*nzcvqj**I**F**t**_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	*nzcvqj**i**Ft_SVC*	*nzcvqj**I**f**t**_SVC*

Disabling Interrupts

Procedure

To disable IRQ and FIQ interrupts, the steps are similar but involve setting the appropriate bits.

1. **Copy CPSR to Register:**
   - Use the `MRS` instruction to copy the CPSR to a register (e.g., `r1`).
2. **Set the IRQ or FIQ Mask Bit:**
   - Use the `ORR` instruction to set the appropriate bit for IRQ (`0x80`) or FIQ (`0x40`).
3. **Update the CPSR:**
   - Use the `MSR` instruction to write the modified value back to CPSR.

Dr. Devaraju B M Dept of CSE, RNSIT

Example Code for Disabling Interrupts:

Disabling an interrupt.

cpsr	IRQ	FIQ
Pre	*nzcvqj**i**ft_SVC*	*nzcvqj**i**ft_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	*nzcvqj**I**ft_SVC*	*nzcvqji**F**t_SVC*

Basic Interrupt Stack Design and Implementation

Exception handlers heavily utilize stacks, with each processor mode having a dedicated register for its stack pointer. The design of these exception stacks is influenced by various factors:

Key Factors Influencing Stack Design

1. **Operating System Requirements:**
   - Different operating systems have specific needs for stack design, impacting the overall architecture and implementation.
2. **Target Hardware:**
   - The physical limitations of the target hardware dictate the size and location of the stack within memory.

Design Decisions for Stacks

1. Stack Location

- **Memory Map Positioning:**
  - Most ARM systems use a downward-growing stack, starting at a high memory address and descending toward lower addresses.

2. Stack Size

- **Nested vs. Non-nested Interrupts:**
  - Nested interrupt handlers require more stack space as the stack grows with each additional interrupt. This necessitates careful sizing to accommodate worst-case scenarios.

Avoiding Stack Overflow

To prevent stack overflow, which can destabilize embedded systems, the following software techniques can be employed:

Dr. Devaraju B M, Dept of CSE, RNSIT

1. **Memory Protection:**
    - Implementing memory protection mechanisms can help detect and prevent stack overflows before they cause significant issues.
2. **Stack Check Function:**
    - Calling a stack check function at the beginning of each routine can help monitor stack usage and trigger corrective actions if overflow is detected.

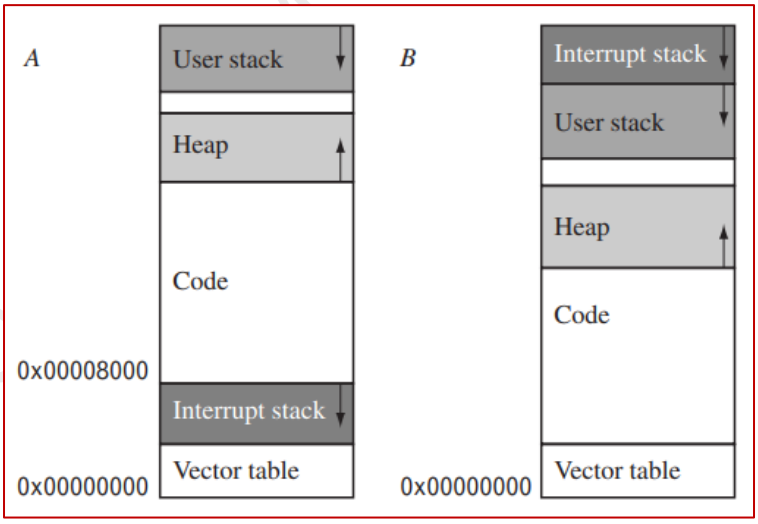
Setting Up the IRQ Mode Stack

- The IRQ mode stack must be initialized before enabling interrupts, typically during system initialization. Understanding the maximum stack size is crucial in embedded systems since it must be reserved during the boot process by the firmware.

Typical Memory Layouts

Figure 9.6: Memory Layouts

1. **Layout A: Traditional Stack Layout**
    - The interrupt stack is stored beneath the code segment. This can lead to issues if the stack overflows into adjacent memory areas.
2. **Layout B: Improved Stack Layout**
    - The interrupt stack is positioned above the user stack. This configuration protects the vector table from being corrupted during a stack overflow, providing the system an opportunity to recover from overflow conditions.



Advantages of Layout B

- **Protection of Vector Table:** Prevents corruption during stack overflow, allowing the system to implement corrective measures.

Dr. Devaraju B M, Dept. of CSE, RNSIT

Stack Setup for Processor Modes

For each ARM processor mode, a dedicated stack must be established during system reset. Below is an implementation using a memory layout (Layout A), which sets up the necessary stacks and processor mode definitions.

Memory Layout Definitions

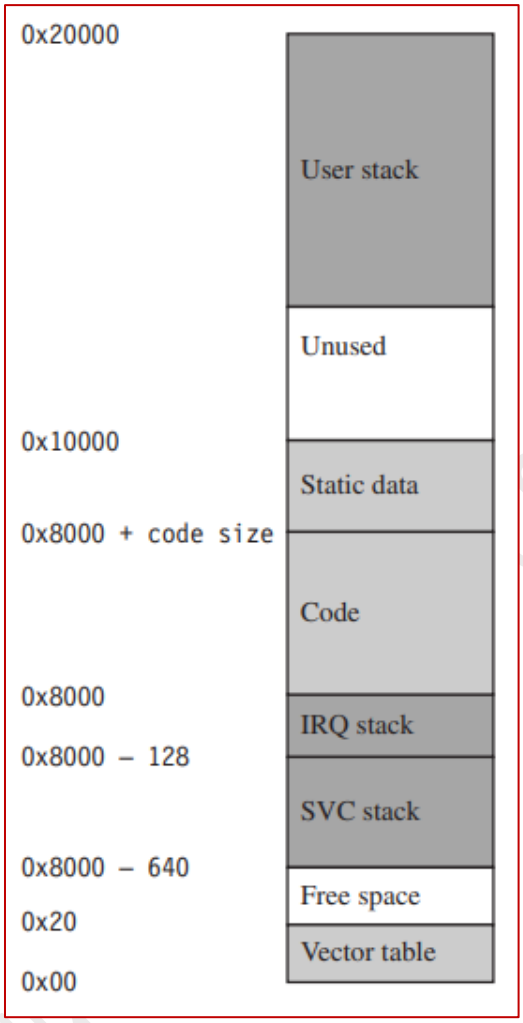
Stack Addresses

Each stack is assigned a specific address in memory. The following defines map memory region names to absolute addresses:

```
USR_Stack EQU 0x20000      ; User stack address
IRQ_Stack EQU 0x8000       ; IRQ stack address
SVC_Stack EQU IRQ_Stack - 128 ; Supervisor stack, 128 bytes below IRQ stack
```

Dr. Devaraju B M, Dept. of CSE, RNSIT

Example implementation using layout A.



Dr. Devaraju B M, Dept of CSE, RNSIT

Processor Mode Definitions

To facilitate switching between processor modes, each mode is associated with a specific 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
```

Disabling Interrupts

For safety, it is important to disable both IRQ and FIQ exceptions in the CPSR. This is achieved with the following define:

```
NoInt EQU 0xc0 ; Disable interrupts (sets both masks to 1)
```

Initialization of Stack Registers for Processor Modes

When the ARM processor core comes out of reset, it is crucial to set up stack registers for each mode. Each mode has a dedicated stack pointer (r13), which is banked, ensuring each mode operates with its own stack.

Stack Initialization Process

Supervisor Mode Stack:

The processor starts in Supervisor mode (SVC), so we initialize the SVC stack first. The stack pointer (r13_svc) is set to point to the SVC stack location.

Dr. Devaraju B M, Dept of CSE, RNSIT

```
LDR r13, SVC_NewStack  ; Load SVC stack address into r13

...

SVC_NewStack

DCD SVC_Stack          ; Define SVC stack location
```

IRQ Mode Stack:

Next, the IRQ stack needs to be initialized. To switch to IRQ mode, we store a specific bit pattern in register r2 and update the CPSR.

```
MOV r2, #NoInt | IRQ32md  ; Set CPSR to disable interrupts and
switch to IRQ mode

MSR cpsr_c, r2            ; Update CPSR to change mode

LDR r13, IRQ_NewStack     ; Load IRQ stack address into r13

...

IRQ_NewStack

DCD IRQ_Stack             ; Define IRQ stack location
```

Dr. Devaraju B M, Dept of CSE, RNSIT

User Mode Stack:

Finally, the User mode stack is set up. Since the processor cannot modify the CPSR in User mode directly, we switch to System mode to initialize the User stack.

```
MOV r2, #Sys32md          ; Set CPSR to System mode

MSR cpsr_c, r2            ; Update CPSR to change mode

LDR r13, USR_NewStack     ; Load User stack address into r13

...

USR_NewStack

DCD USR_Stack             ; Define User stack location
```

Advantages of Separate Stacks

Using separate stacks for each processor mode has a significant advantage: it allows for better debugging and isolation of errant tasks. This design ensures that issues in one mode do not affect the stability of others, enhancing overall system reliability.

By following this initialization process, we ensure that the processor is correctly configured with dedicated stacks for Supervisor, IRQ, and User modes, setting a solid foundation for reliable operation in an embedded system.

Dr. Devaraju B M, Dept. of CSE, RNSIT

Firmware and Bootloader

1. **Firmware:**
    - **Role and Function:** Firmware is deeply embedded, low-level software providing an interface between hardware and application/operating system level software. It resides in ROM and executes when power is applied to the system. Firmware can remain active after system initialization to support basic operations.
    - **Usage:** The choice of firmware for an ARM-based system depends on the application, ranging from loading a sophisticated OS to relinquishing control to a microkernel. Requirements vary greatly across implementations. For instance, minimal firmware support may be sufficient for booting a small OS in a simple system.
    - **Purpose:** One primary purpose of firmware is to provide a stable mechanism for loading and booting an OS.
2. **Bootloader:**
    - **Role and Function:** A bootloader is a small application responsible for installing the OS or application onto a hardware target. It exists only until the OS or application is executing and is often part of the firmware.
    - **Execution Flow:** The bootloader's function is complete once the OS or application takes control.

Firmware Execution Flow:

Stage 1: Set Up Target Platform

- **Program the Hardware System Registers:** Initializes the hardware environment to prepare it for booting the operating system. This includes setting control registers, configuring the memory map, and ensuring all hardware components are in a known state.
- **Platform Identification:** Identifies the core and platform by reading specific registers (e.g., register 0 in coprocessor 15) and checking for known peripherals or preprogrammed chips.
- **Diagnostics:** Runs initial diagnostics to detect basic hardware malfunctions, specific to the hardware being used.
- **Debug Interface:** Provides a module or monitor to assist in debugging code on the hardware target.
- **Command Line Interpreter (CLI):** Allows interaction with the firmware, enabling changes to configurations and other system settings via commands.

Stage 2: Abstract the Hardware

- **Hardware Abstraction Layer (HAL):** Provides a consistent set of programming interfaces that hide the underlying hardware differences. This makes it easier to port software to new hardware platforms without changing the software's high-level logic.
- **Device Driver:** Interfaces between the HAL and specific hardware peripherals, providing a standard API to read and write to these devices.

Dr. Devaraju B M, Dept. of CSE, RNSIT

Stage 3: Load a Bootable Image

- **Basic Filing System:** Manages storage media (e.g., flash ROM, hard drives) and provides a way to store and retrieve executable images.
- **Relinquish Control:** Alters the program counter (PC) to point to the new image and updates the vector table to redirect exception and interrupt vectors to the operating system handlers.
- **Diagnostics Software:** Ensures the correct operation of the hardware, often specific to the hardware used.

Debug Capability:

- **Breakpoints:** Allows program interruption to examine the state of the processor core.
- **Memory Operations:** Enables listing and modifying memory using peek and poke operations.
- **Register Contents:** Displays the current processor register contents.
- **Disassembly:** Converts memory into ARM and Thumb instruction mnemonics.

Interactive Functions:

- Commands can be sent through the CLI or a dedicated host debugger.
- RAM images can be debugged through a software debug mechanism if internal hardware debug circuitry is not accessible.

Stage 4: Relinquish Control

- **Updating Vector Table:** Redirects exception and interrupt vectors to operating system handlers.
- **Modifying PC:** Points the program counter to the OS entry point address.
- **Data Structure for OS:** For complex OSes like Linux, a standard data structure is passed to the kernel, detailing the runtime environment (e.g., available RAM, MMU type).

Loading Process:

- **Image Formats:** Supports various image formats such as plain binary and ELF (Executable and Linking Format).
- **Image Handling:** May involve decryption or decompression of the image before execution.
- **Flash ROM Filing System (FFS):** Common in ARM-based systems, allowing multiple executable images to be stored and managed.

Dr. Devaraju B M, Dept of CSE, RNSIT

ARM Firmware Suite (AFS)

ARM Firmware Suite (AFS) is a firmware package developed by ARM specifically for ARM-based embedded systems. It supports a variety of boards and processors, including Intel XScale and StrongARM processors. AFS comprises two major components: the Hardware Abstraction Layer (μHAL) and the debug monitor called Angel.

1. Hardware Abstraction Layer (μHAL):

- **Purpose:** Provides a low-level device driver framework that enables it to operate over different communication devices (USB, Ethernet, serial).
- **Standard API:** μHAL offers a standard API which simplifies the porting process to different hardware platforms by implementing hardware-specific parts according to μHAL API functions.
- **Key Features:**
  - **System Initialization:** Sets up the target platform and processor core. This can range from simple to complex depending on the platform.
  - **Polled Serial Driver:** Basic communication method with a host.
  - **LED Support:** Controls LEDs for user feedback and operational status display.
  - **Timer Support:** Sets up periodic interrupts, essential for preemptive context-switching operating systems.
  - **Interrupt Controllers:** Supports different types of interrupt controllers.
- **Boot Monitor:** Contains a Command Line Interface (CLI) for interaction.

2. Debug Monitor (Angel):

- **Purpose:** Facilitates communication between a host debugger and a target platform.
- **Functionality:**
  - **Memory Inspection and Modification:** Allows for the inspection and modification of memory.
  - **Image Download and Execution:** Supports downloading and executing images.
  - **Breakpoints and Register Display:** Enables setting breakpoints and displaying processor register contents.
  - **Communication:** Utilizes SWI (Software Interrupt) and IRQ/FIQ (Interrupt Request/Fast Interrupt Request) vectors for communication.

- **APIs Provided by Angel:**
  - **File System Access:** Programs can open, read, and write to a host filing system through SWI instructions.
  - **Interrupt Handling:** Uses IRQ/FIQ interrupts for communication with the host debugger.

Dr. Devaraju B M, Dept. of CSE, RNSIT

Advantages:

- **Standardized Framework:** The µHAL standard API makes the porting process straightforward.
- **Flexibility:** Once firmware is ported, the operating system can be moved to the new platform, leveraging the µHAL API calls for hardware access.
- **Comprehensive Debugging:** Angel provides extensive debugging capabilities through the host debugger.

The ARM Firmware Suite (AFS), with its µHAL and Angel components, offers a robust solution for ARM-based embedded systems. It simplifies the porting process, supports a wide range of hardware, and provides extensive debugging tools to facilitate development and maintenance.

Red Hat RedBoot:

RedBoot is a versatile and powerful firmware tool developed by Red Hat, available under an open-source license without any royalties or upfront fees. It is designed to execute on various CPUs such as ARM, MIPS, SH, and others. RedBoot combines debug capabilities with bootloader functionality, making it a valuable tool for embedded system development.

Key Features:

1. **Communication:**
   - **Serial Communication:** Utilizes the X-Modem protocol to interface with the GNU Debugger (GDB).
   - **Ethernet Communication:** Uses TCP to communicate with GDB, supporting a range of network protocols such as BOOTP, Telnet, and TFTP.
2. **Flash ROM Memory Management:**
   - Provides a set of file system routines for downloading, updating, and erasing images in flash ROM.
   - Supports both compressed and uncompressed images, offering flexibility in managing firmware images.
3. **Full Operating System Support:**
   - Capable of loading and booting various operating systems, including Embedded Linux and Red Hat eCos.
   - Allows the definition of parameters that can be passed directly to the kernel upon booting, particularly useful for Embedded Linux.

Dr. Devaraju B M, Dept of CSE, RNSIT

Components:

- **HAL (Hardware Abstraction Layer):** The core of RedBoot is based on a HAL, which abstracts the hardware specifics, allowing RedBoot to run on different CPU architectures seamlessly.
- **GNU Debugger (GDB):** Integrated with GDB, RedBoot provides extensive debugging capabilities, enabling developers to set breakpoints, inspect memory, and control execution flow.

Use Cases:

- **Debugging:** With its integration with GDB, RedBoot allows developers to debug embedded systems effectively.
- **Bootloading:** RedBoot serves as a reliable bootloader for initializing hardware and loading the operating system or application images into memory.
- **Firmware Management:** Its flash ROM management capabilities make it easy to handle firmware updates and maintenance.

Network Standards Supported:

- BOOTP: Bootstrap Protocol for network booting.
- Telnet: For remote communication and control.
- TFTP: Trivial File Transfer Protocol for transferring firmware images over the network.

Practical Applications:

- **Embedded Linux Systems:** RedBoot is widely used to boot Embedded Linux systems, providing a mechanism to pass boot parameters directly to the Linux kernel.
- **Embedded Operating Systems:** Beyond Linux, RedBoot supports other embedded operating systems like Red Hat eCos, making it a versatile choice for various embedded projects.

RedBoot's combination of HAL-based architecture, robust communication protocols, comprehensive debugging support, and versatile flash ROM management makes it an essential tool for embedded system developers. Its open-source nature and support for multiple CPU architectures add to its appeal, enabling efficient development, debugging, and deployment of firmware and operating systems in embedded environments.

Sandstone

Sandstone is a minimal firmware system designed to perform three essential tasks:

Dr. Devaraju B M, Dept. of CSE, RNSIT

1. Set up the target platform environment.
2. Load a bootable image into memory.
3. Relinquish control to an operating system.

Despite its simplicity, Sandstone is a real working example that illustrates how a minimal system can be used to initialize hardware and boot software on an ARM-based platform.

Platform Specifics:

- **Target Platform:** ARM Evaluator-7T, which includes an ARM7TDMI processor.
- **Implementation:** Written entirely in ARM assembler.
- **Purpose:** To provide a clear example of how to set up a simple platform and load/boot a software payload, which can be an application or an operating system image.
- **Configuration:** Static design, meaning it cannot be reconfigured after the build process.

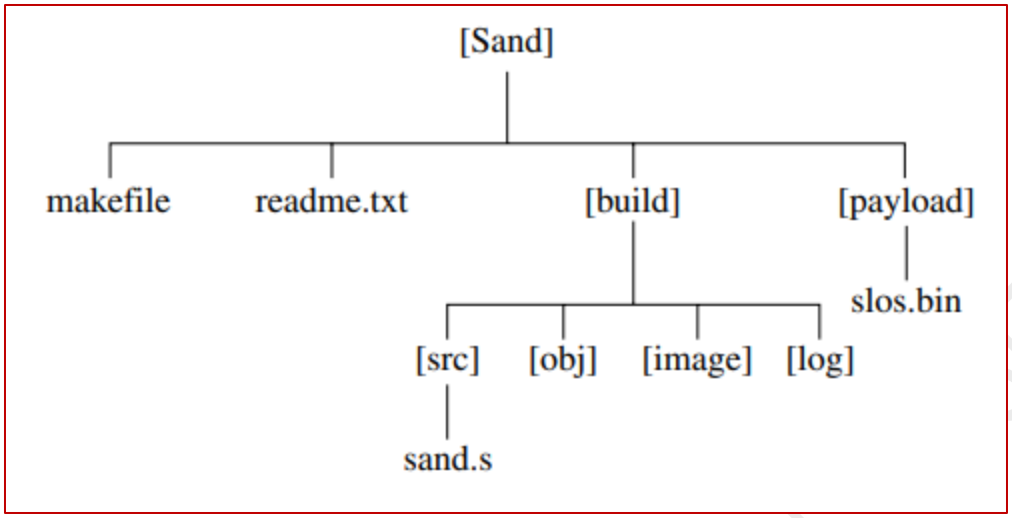
Basic Characteristics:

- **Code:** ARM instructions only.
- **Tool Chain:** ARM Developer Suite 1.2.
- **Image Size:** 700 bytes.
- **Source Size:** 17 KB.
- **Memory Remapped:** Yes.

Directory Layout: The directory structure of Sandstone follows a standard style, which will be used in further projects. Here is a summary of the layout:

- **sand/**: Root directory.
    - **build/**: Contains build-related files.
        - **src/**: Source code directory, includes `sand.s`.
        - **obj/**: Contains object files produced by the assembler.
        - **image/**: Final Sandstone image, which includes both Sandstone code and the payload.
    - **payload/**: Contains the payload image to be loaded and booted by Sandstone.
    - **readme.txt**: Description of the build procedure and instructions on creating the binary image for the ARM Evaluator-7T.

Dr. Devaraju B M, Dept. of CSE, RNSIT



Build Process: To understand the build procedure, refer to the `readme.txt` file in the root directory (`sand/`). This file provides step-by-step instructions on building the example binary image.

Code Structure:

- **Initialization:** The code focuses on initializing the target platform, which involves setting up hardware registers and ensuring the platform is ready to boot an operating system.
- **Boot Process:** After initialization, Sandstone loads a bootable image into memory and then modifies the program counter (PC) to point to the new image, effectively handing control over to the operating system or application.

Usage: Sandstone is a practical example for students to learn about firmware development, platform initialization, and the boot process. It shows how minimal code can be used to perform essential tasks on an embedded system.

Sandstone Code Structure

Sandstone consists of a single assembly file, structured into several key steps corresponding to the execution flow. Each step prepares the platform for running firmware.

Sandstone execution flow.

Step	Description
1	Take the Reset exception
2	Start initializing the hardware
3	Remap memory
4	Initialize communication hardware
5	Bootloader—copy payload and relinquish control

Devaraju B M, Dept. of CSE, RNSIT

The primary goal of Sandstone is to set up the platform environment, providing feedback that the firmware is running and has control.

Step 1: Take the Reset Exception

- **Execution begins** with a Reset exception.
- **Default vector table** contains only the reset vector; other vectors lead to infinite loops (dummy handlers).
- **Initial Control Flow:**

```
AREA start,CODE,READONLY

ENTRY

sandstone_start

B sandstone_init1 ; reset vector

B ex_und ; undefined vector

B ex_swi ; swi vector

B ex_pabt ; prefetch abort vector

B ex_dabt ; data abort vector

NOP ; not used...

B int_irq ; irq vector

B int_fiq ; fiq vector
```

Results:

- Dummy handlers set up.
- Control passed to hardware initialization.

Step 2: Start Initializing the Hardware

- **System Registers:** Must be set up before accessing hardware.
- **Example:** For the ARM Evaluator-7T, the seven-segment display is initialized for feedback.
- **Base Address Configuration:** System registers base address set to 0x03ff0000.

Dr. Devaraju B M, Dept. of CSE, RNSIT

sandstone_init1

LDR r3, =SYSCFG ; where SYSCFG=0x03ff0000

LDR r4, =0x03ffffa0

STR r4, [r3]

Results:

- System registers set to base address `0x03ff0000`.
- Segment display configured for progress indication.

Step 3: Remap Memory

- **Memory Initialization:** Set up SRAM and remap flash ROM.
- **Initial Memory State:**

Initial memory state.

Memory type	Start address	End address	Size
Flash ROM	0x00000000	0x00080000	512K
SRAM bank 0	Unavailable	unavailable	256K
SRAM bank 1	Unavailable	unavailable	256K

LDR r14, =sandstone_init2

LDR r4, =0x01800000 ; new flash ROM location

ADD r14, r14, r4

ADRL r0, memorymaptable_str

LDMIA r0, {r1-r12}

LDR r0, =EXTDBWTH ; =(SYSCFG + 0x3010)

STMIA r0, {r1-r12}

MOV pc, r14 ; jump to remapped memory

Results:

- Memory remapped according to the new layout.
- Control now points to the next step in the newly remapped flash ROM.

Dr. Devaraju B M, Dept of CSE, RNSIT

Step 4: Initialize Communication Hardware

- **Communication Setup:** Configure serial port and send a banner.
- **Serial Port Configuration:** 9600 baud, no parity, one stop bit, no flow control.
- **Results:**
  - Serial port initialized.
  - Banner output:

Sandstone Firmware (0.01)

- platform e7t

- status ..... alive

- memory ..... remapped

+ booting payload ...

Step 5: Bootloader—Copy Payload and Relinquish Control

- **Copying the Payload:**

```
sandstone_load_and_boot
MOV r13,#0 ; destination addr
LDR r12,payload_start_address ; start addr
LDR r14,payload_end_address ; end addr
copy
LDMIA r12!,{r0-r11}
STMIA r13!,{r0-r11}
CMP r12,r14
BLT _copy
MOV pc,#0
```

Final Steps:

- Payload copied into SRAM at address `0x00000000`.
- Control relinquished to the payload:

Dr. Devaraju B M, Dept. of CSE, RNSIT

Sandstone Firmware (0.01)

- platform e7t

- status ..... alive

- memory ..... remapped

+ booting payload ...

Simple Little OS (0.09)

- initialized ..... ok

- running on e7t

- **Payload copied:** into SRAM, starting at `0x00000000`.
- **Control of the pc:** transferred to the payload, indicating system boot completion.

Dr. Devaraju B M, Dept. of CSE, RNSIT

Module – 4: Questions

Exception Handling

1. Define an exception in the context of ARM processors. What are the common causes of exceptions?
2. Explain the mechanism of exception handling in ARM processors. What happens when an exception occurs?
3. Describe the function of the Vector Table in ARM architecture. What is its significance in exception handling?
4. List and explain the different types of exceptions in ARM processors and the modes they trigger.
5. How does the ARM processor prioritize exceptions? What are the priority levels for each type of exception?

Interrupt Handling

6. Differentiate between IRQ and FIQ exceptions in ARM processors. What are their typical uses?
7. Outline the process that occurs when an IRQ exception is raised in an ARM processor.
8. Describe how the ARM processor handles FIQ exceptions. How is it different from handling IRQ exceptions?
9. Explain the terms "interrupt latency" and "nested interrupt handlers." How can interrupt latency be minimized in ARM systems?
10. What are the procedures for enabling and disabling IRQ and FIQ exceptions in ARM processors? Provide example code for each.

Link Register and Exception Return

11. Explain the role of the Link Register (LR) during exception handling in ARM processors. How is it used to return from exception handlers?
12. Discuss the different methods for returning from an IRQ or FIQ handler. Provide example code for at least two methods.

Stack Design

13. What factors influence the design of interrupt stacks in ARM systems?
14. Compare and contrast traditional and improved stack layouts in ARM systems. Which layout provides better protection against stack overflow and why?

Dr. Devaraju B M, Dept. of CSE, RNSIT

15. Describe the process of setting up the IRQ mode stack during system initialization. Why is it important to understand the maximum stack size in embedded systems?

Practical Applications

16. In a scenario where multiple exceptions occur simultaneously, how does the ARM processor determine which exception to handle first?
17. Given a piece of code that triggers a software interrupt (SWI), explain how the processor transitions to handling the SWI exception and what steps are involved.
18. Design a simple interrupt service routine (ISR) for handling an external IRQ in an ARM-based embedded system. Ensure to include code for saving and restoring context.
19. Explain how memory protection mechanisms can help prevent stack overflow in ARM-based systems. Provide an example of how this can be implemented.
20. Discuss the importance of the CPSR (Current Program Status Register) in managing exceptions and interrupts. How does modifying the CPSR affect the processor's behavior?

Dr. Devaraju B M, Dept. of CSE, RNSIT