Exception Processing and Interrupt

Basic Concepts of Interrupt Processing
- A variety of unexpected events in a computer system
- I/O events, error conditions, network events, etc.
- These events are handled by interrupt processing
- Speed disparity of various devices in a computer
- Allow multiple and parallel processing of tasks
- An analogous example: A reading process
  - Phone rings
  - Recognition of an event
  - Answer the phone or not? Priority
  - Book mark the page, store context
  - Answer the phone, handler
  - Continue the reading process after phone conversation

Interrupts
- Motivations
  - Inform a program of some external events timely
    - Polling vs Interrupt
  - Implement multi-tasking with priority support
  - Merriam-Webster: "to break the uniformity or continuity of"

Polling vs Interrupt
- Polling: You pick up the phone every three seconds to check whether you are getting a call.
- Interrupt: Do whatever you should do and pick up the phone when it rings.

Necessary Procedure of an Interrupt Process
- Interrupt requester ↔ CPU
- Recognition of an interrupt request:
  - Interrupt requester makes an interrupt request
  - CPU recognizes the interrupt request
- Prioritization: Determining whether granting the request or not:
  - Requester provides its priority
  - CPU compares it with the priority of its current process
- Context saving to be able to come back after interrupt:
  - Program counter marks where interrupt happened
  - Status register and possible other necessary context information

SoC interconnect diagram
- Any module capable of generating an interrupt will depend on NVIC operation
- NMI can be generated from:
  - External pin (must configure the MUX)
  - CoreSight Embedded Trace Buffer (ETB)
Interrupt Service Routine Vector Table

- Start address for the exception handler for each exception type is fixed and pre-defined
- Processor loads PC with this fixed pre-defined address
- Exception Vector Table starts at memory address 0
- Program Counter $pc = 0x00000004$ initially

Stacking & Unstacking

- **Stacking**: The processor automatically pushes these eight registers into the main stack before an interrupt handler starts
- **Unstacking**: The processor automatically pops these eight registers out of the main stack when an interrupt handler exits

Interrupt

- Program Counter $pc$ starts at memory address 0
- Exception Vector Table

Registers

- MSP: Main Stack Pointer
- PSP: Process Stack Pointer

Two stack pointers: Main SP (MSP) and Process SP (PSP)

- Determined by operating mode, and bit 6 of the CONTROL register
- Handler mode: $SP = MSP$
- User mode: $SP = PSP$

Interrupt Signal

User Program

Interrupt Handler

User Program

Interrupt Exit

Stacking

Unstacking

Thread Mode

Handler Mode

Time
### Processor Mode: Handler Mode vs Thread Mode

- Handler mode and Thread mode
- Handler mode always use MSP (Main Stack Pointer)
- Thread Mode uses either PSP (Process Stack Pointer) or MSP
  - Control[1] = 0, SP = MSP (default)
  - Control[0] = 1, SP = PSP
- When the processor is reset, the default is the thread mode.
- The processor enters the handler mode when an exception occurs.

### Sequence of register setups

The steps for enabling an interrupt on NVIC:

1. Enable the peripheral to be used
2. Set the proper bit on the NVIC
3. Clear any pending interrupt by writing to the NVIC
4. Configure the interrupt priority by writing to the NVIC
5. Write the ISR
6. Enable global interrupts

### Stacking & Unstacking

Control[1] = 0 ➔ User program uses MSP.

![Stacking & Unstacking Diagram](image)

Control[1] = 1 ➔ User program uses PSP.

![Stacking & Unstacking Diagram](image)

### Interrupt Number

Cortex-M supports up to 256 interrupts.

- First 16 are system exceptions
- CMSIS defines their interrupt numbers as negative
- Defined by ARM core
- The rest 240 are peripheral interrupts
- Peripheral interrupt number starts with 0.
- Defined by chip manufacturers.

### Interrupt Number in CMSIS vs in PSR

**Interrupt number for CMSIS functions**

- NVIC_EnableIRQ IRQn); // Enable interrupt
- NVIC_DeableIRQ IRQn); // Disable interrupt
- NVIC_ClearingPendingIRQ IRQn); // Clear pending status
- NVIC_SetPriority IRQn, priority); // Set priority level

<table>
<thead>
<tr>
<th>Interrupt Number in Program Status Register (PSR)</th>
<th>0 or Exception Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
</tr>
<tr>
<td>--------------------------------------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>Exception</td>
<td>0</td>
</tr>
<tr>
<td>NVIC[S]</td>
<td>31</td>
</tr>
<tr>
<td>ESR</td>
<td></td>
</tr>
<tr>
<td>ESR</td>
<td></td>
</tr>
<tr>
<td>ESR</td>
<td></td>
</tr>
<tr>
<td>ESR</td>
<td></td>
</tr>
<tr>
<td>ESR</td>
<td></td>
</tr>
<tr>
<td>ESR</td>
<td></td>
</tr>
<tr>
<td>ESR</td>
<td></td>
</tr>
<tr>
<td>ESR</td>
<td></td>
</tr>
<tr>
<td>ESR</td>
<td></td>
</tr>
<tr>
<td>ESR</td>
<td></td>
</tr>
<tr>
<td>ESR</td>
<td></td>
</tr>
<tr>
<td>ESR</td>
<td></td>
</tr>
<tr>
<td>ESR</td>
<td></td>
</tr>
<tr>
<td>ESR</td>
<td></td>
</tr>
<tr>
<td>ESR</td>
<td></td>
</tr>
<tr>
<td>ESR</td>
<td></td>
</tr>
<tr>
<td>ESR</td>
<td></td>
</tr>
</tbody>
</table>

**Interrupt Number in PSR = 16 + Interrupt Number for CMSIS**
Enable an Interrupt

- Enable a system exception
  - Some are always enabled (cannot be disabled)
  - No centralized registers for enabling/disabling
  - Each are control by its corresponding components, such as SysTick module
- Enable a peripheral interrupt
  - Centralized register arrays for enabling/disabling
  - ISER registers for enabling
  - ICER registers for disabling

Disabling Peripheral Interrupts

```
NVIC->ICER[IRQn >> 5] = 1 << (IRQn & 0x1F); // Disable Interrupt
```

Interrupt Priority

- Inverse Relationship:
  - Lower priority value means higher urgency.
  - Priority of Interrupt A = 5.
  - Priority of Interrupt B = 2.
  - B has a higher priority/urgency than A
- Fixed priority for Reset, HardFault, and NMI.

<table>
<thead>
<tr>
<th>Exception</th>
<th>IRQn</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>N/A</td>
<td>3 (highest)</td>
</tr>
<tr>
<td>Non-maskable Interrupt (NMI)</td>
<td>-14</td>
<td>-2 (2nd highest)</td>
</tr>
<tr>
<td>Hard Fault</td>
<td>-13</td>
<td>-1</td>
</tr>
</tbody>
</table>

Adjustable for all the other interrupts
Interrupt Priority

- Interrupt priority is configured by Interrupt Priority Register (IP).
- Each priority consists of two fields, including preempt priority number and sub-priority number.
  - The preempt priority number defines the priority for preemption.
  - The sub-priority number determines the order when multiple interrupts are pending with the same preempt priority number.

Priority Byte

<table>
<thead>
<tr>
<th>Preempt Priority</th>
<th>Sub-priority Number</th>
<th>Not Implemented</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Preemption and Sub-priority Configuration

- NVIC_SetPriorityGrouping(n)
  - Perform unlock, and update AIRCR register

<table>
<thead>
<tr>
<th>n</th>
<th># of bits in preemption priority</th>
<th># of bits in sub-priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2 (default)</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

Priority of Peripheral Interrupts

// Set the priority for EXTI 0 (Interrupt number 6)
NVIC->IP[6] = 0xF0;

Priority of System Interrupts

// Set the priority of a system interrupt IRQn
SCB->SMP[(IRQn) & 0xF] - 4 = (priority << 4) & 0xFF;

Masking Priority

- Disable all interrupts with less urgency
  - For critical code, only certain interrupts are allowed.
  - BASEPRI: Disable all interrupts of specific priority level or higher priority level

// Disable interrupts with priority value same or higher
_set_BASEPRI($ << 4 )

// Critical code
---

// Remove BASEPRI masking
_set_BASEPRI(0);

Interrupt Priority Levels

NVIC_SetPriority(7, 6);

core_cm4.h or core_cm3.h

typedef struct {
  ... // Interrupt Priority Register
  volatile uint8_t IP[36];
  ...} NVIC_Type;

IP = 0x60 = 96

It is equivalent to:

NVIC->IP[7] = (6 << 4) & 0xFF;

Exception-masking registers (PRIMASK, FAULTMASK and BASEPRI)

- **PRIMASK**: Used to disable all exceptions except for Non-maskable interrupt (NMI) and hard fault.
  - Write 1 to PRIMASK to disable all interrupts except NMI

- **FAULTMASK**: Like PRIMASK but change the current priority level to -1, so that even hard fault handler is blocked

- **BASEPRI**: Disable interrupts only with priority lower than a certain level
  - Example, disable all exceptions with priority level higher than 0x60

```assembly
MOV R0, #1
MSR PRIMASK, R0

MOV R0, #0
MSR PRIMASK, R0

MOV R0, #0x60
MSR BASEPRI, R0
```

External Interrupt (EXTI) Sources

- SYSCFG external interrupt configuration register
- NVIC
- Cortex-M4 Nested-Vectored Interrupt Controller (NVIC)

<table>
<thead>
<tr>
<th>EXTIx</th>
<th>Source selection</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 0, 1, 2, ..., 15</td>
<td></td>
</tr>
</tbody>
</table>

```assembly
PA.x
PB.x
PC.x
PD.x
PE.x
PF.x
PG.x
PH.x
```

Interrupt Vector Table

<table>
<thead>
<tr>
<th>Interrupt Number (8 bits)</th>
<th>Memory Address of ISR (32 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>EXTI0_IRQn = 6</td>
<td>EXTI0_IRQHandler</td>
</tr>
<tr>
<td>EXTI1_IRQn = 7</td>
<td>EXTI1_IRQHandler</td>
</tr>
<tr>
<td>EXTI2_IRQn = 8</td>
<td>EXTI2_IRQHandler</td>
</tr>
<tr>
<td>EXTI3_IRQn = 9</td>
<td>EXTI3_IRQHandler</td>
</tr>
<tr>
<td>EXTI4_IRQn = 10</td>
<td>EXTI4_IRQHandler</td>
</tr>
<tr>
<td>EXTI5_0_IRQn = 23</td>
<td>EXTI5_0_IRQHandler</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>EXTI15_10_IRQn = 48</td>
<td>EXTI15_10_IRQHandler</td>
</tr>
</tbody>
</table>

External Interrupt (EXTI) Controller

- GPIO, RTC, COM, PVD, PVM
- USART/I2C/OTG/LPTIM/LCD
- High Speed and Direct Data Transfer Between Memory and Peripherals

DMA: Direct Memory Access
Basic Concepts of DMA

- Limitations of Interrupt Processing
  - CPU involvements
  - Good for discrete events with small amount of data
  - Inefficient for large data transfers

- Needs for High Speed Data Transfer between
  - disk and RAM,
  - NIC and RAM,
  - more

- An analogous example: Program of Study
  - Student asks dean for advising and signature \(\rightarrow\) Request
  - Dean directs student to the advisor \(\rightarrow\) address, tasks, and go
  - Student talks with the advisor \(\rightarrow\) communication/data transfer
  - Advisor signed the program of study after completing advising and send student back o dean for final signature \(\rightarrow\) report completion

General Procedure of DMA

- DMA Request
  - Request from peripheral through hardware
  - Explicit software initiation
  - Channel to channel linking for continual transfer

- Source/Destination and amount of Data Transfer
  - CPU write registers in DMA controller to define
    - Source address, destination address, and byte count

- Direct and Continuous Data Transfer
  - Data transfer is done directly between memory and peripheral device
  - without CPU involvement

- Report Completion
  - When transfer is done, reporting completion through interrupt

Direct Memory Access (DMA)

- DMA releases CPU from moving data
  - between peripherals and memory, or
  - between one peripheral and another peripheral.

- DMA uses bus matrix to allow concurrent transfers

DMA Sets Core Free

- CPU delegates reads/writes to DMA controller
- Low overhead and fast

Programmed I/Os

- Processor executes a lot Loads/Stores to move data
  - High overhead and slow

DMA Controller

- Basic Procedures
  - DMA device requests bus
  - CPU grants bus request
  - CPU takes its signals to HiZ

- Key DMA Controller Registers
  - DMA memory address register (CMAR)
  - DMA peripheral address register (CPAR)
  - DMA number of data register (CNDTR)
  - DMA configuration register (CCR)

- DMA are often used together with interrupts
**DMA Mode:** Incremental Mode

**DIR:** Data transfer direction: 0 = Read from peripheral; 1 = Read from memory

- **Non-incremental mode**
- **Incremental mode**

**DMA Bus Matrix**

- **DMA peripheral address register**
- **DMA memory address register**

**1st DMA Transfer**

**2nd DMA Transfer**

**DMA stops since CNDTR is zero now.**

**DMA Mode:** Circular Mode

- **Circular Mode**
- **Handle circular buffers and continuous data flows**
- **The number of data to be transferred (CNDTR) is automatically reloaded and DMA requests continue to be served.**

**1st DMA Transfer**
DMA Mode:
Incremental Mode

DIR: Data transfer direction: 0 = Read from peripheral; 1 = Read from memory

Peripheral registers/memory

DMA Bus Matrix

memory

non-incremental mode

incremental mode

DIR = 0
CNDTR = 0
CIR = 1

2nd DMA Transfer

DMA resets and continues to run!

DMA Interrupts

- Programmable and Independent source and destination transfer data size: Byte, Halfword or Word
- Three event flags: DMA Half Transfer, DMA Transfer complete and DMA Transfer Error
- Software programmable priorities: Very high, High, Medium or Low

DMA Request Mapping

- High Priority Request
- Low Priority Request

DMA Summary

- Without DMA, CPU has to execute many load and store instructions, leading to slower performance.
- DMA, which makes an automatic data transfer when received a DMA request without involving CPU, accelerates the overall performance.