

Low Latency Interrupt Handling in Safety Critical Systems

Sivam S. Jadhav¹, Mohan V. Kulkarni², Ayesha P. Khan³

Associate Professor, Students

Department of Electronics Engineering, Shantiniketan

PSG College of Technology, Coimbatore, India

sivamsjadhavdh@gmail.com¹, mohankulkarni12v@yahoo.com², khanayeshappp@rediffmail.com³

Abstract

Safety critical embedded systems such as automotive controllers, avionics computers, medical devices and industrial automation units demand extremely fast and deterministic response to external events. Interrupt handling plays a very important role in ensuring that such systems react within strict timing constraints. Any delay in servicing interrupts may lead to system instability, failure or even risk to human life. Therefore, low latency interrupt handling is considered as a core requirement in the design of real-time safety systems. This paper reviews the concepts, challenges and techniques for achieving low interrupt latency in safety critical environments. Architectural aspects, operating system support, hardware prioritization, nested interrupts, cache effects, and compiler optimizations are discussed. Practical design strategies and comparisons are presented with tables and diagrams. The study also highlights the importance of deterministic response, preemption, interrupt vectoring, and real time scheduling.

Keywords: Interrupt latency, safety critical systems, real time systems, deterministic response, embedded systems, ISR optimization, interrupt prioritization.

1. INTRODUCTION

Safety critical systems are those systems where failure may cause severe damage, injury or loss of life. Examples include

anti-lock braking system (ABS), aircraft flight control, patient monitoring devices, nuclear plant controllers and railway

signaling systems. These systems operate under strict real time constraints.

In such systems, **interrupts** are primary mechanism through which external devices communicate urgent events to the processor. The speed at which the processor responds to these interrupts is called **interrupt latency**.

Interrupt latency must be:

- Predictable
- Minimal
- Deterministic

If interrupt service routine (ISR) is delayed, critical data may be lost or control action may be too late.

2. BASICS OF INTERRUPT HANDLING

Interrupts are fundamental mechanism used in embedded and real-time processors to respond to urgent events without continuously polling devices. In safety critical systems, interrupts allow the processor to react immediately to sensor signals, fault conditions, communication requests, or timing events.

An interrupt can be generated by:

- **Hardware sources** – sensors, timers, communication peripherals, ADC completion, fault lines
- **Software sources** – system calls, exceptions, watchdog alerts

When an interrupt occurs, the processor temporarily halts the current program flow and transfers control to a predefined routine called the **Interrupt Service Routine (ISR)**. The time taken for this transfer is crucial for real-time performance.

2.1 Detailed Interrupt Handling Steps

The interrupt handling process inside the CPU is not instantaneous. It involves a well-defined sequence of micro-operations that together contribute to interrupt latency.

Step 1: Interrupt Request Generated

A peripheral device or internal module asserts an **interrupt request line (IRQ)** to the processor's interrupt controller.

Examples:

- Timer overflow
- Data received on UART
- Sensor threshold crossed
- Fault detected in system
- The interrupt controller checks:
 - Whether interrupts are enabled
 - Priority level of the interrupt
 - Masking status

If conditions are satisfied, interrupt is forwarded to CPU.

Step 2: CPU Finishes Current

Instruction

The processor does **not stop immediately**.

It completes the execution of the current instruction to maintain program consistency.

This delay depends on:

- Instruction length
- Processor architecture (CISC instructions may take longer)
- Pipeline stage of execution

This is the **first contributor** to interrupt latency.

Step 3: Context is Saved

Before jumping to ISR, the CPU must save the current state so it can resume later.

The context typically includes:

- Program Counter (PC)
- Status Register / Flags
- General purpose registers
- Stack pointer (in some architectures)

In some processors (e.g., ARM Cortex-M), this is done automatically using hardware stacking. In others, software saving is needed, increasing delay.

Step 4: ISR Address is Fetched from Interrupt Vector Table

Each interrupt has a predefined memory location stored in a structure called the **Interrupt Vector Table (IVT)**.

The CPU:

- Identifies interrupt number
- Looks up ISR address from IVT
- Loads this address into PC

If the vector table is placed in slow memory, extra delay occurs.

Step 5: ISR Execution Begins

Now the ISR starts executing. ISR should be short and efficient.

Typical ISR tasks:

- Read data from hardware register
- Clear interrupt flag
- Set a status flag for main program
- Exit quickly

Long computations inside ISR increases overall latency for other interrupts.

Step 6: Context Restored

After ISR completes, the processor restores the previously saved context:

- Registers restored
- Status flags restored
- PC restored

This ensures program resumes exactly from where it was interrupted.

Step 7: Normal Execution Resumes

The CPU continues execution of the main program as if interruption never happened. This transparent switching is essential for reliable embedded operation.

**Interrupt Latency = Time from
interrupt request to start of ISR
execution**

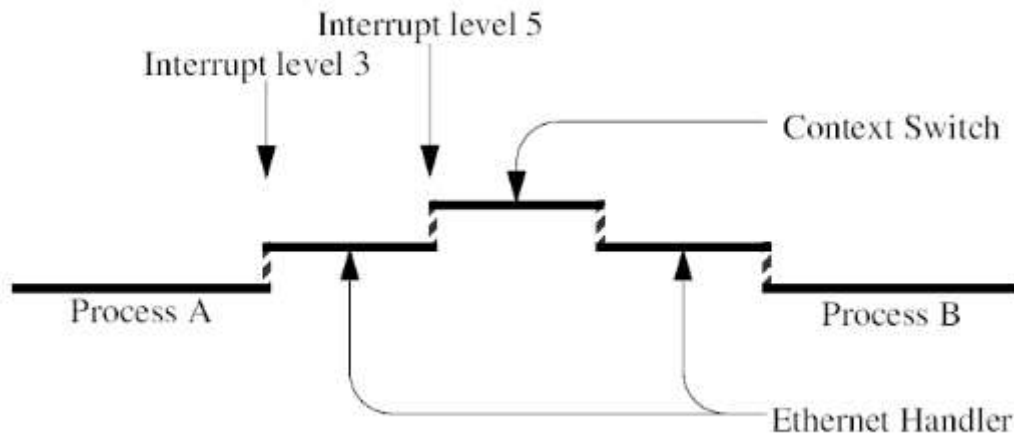


Figure 1: Interrupt Handling Timeline

3. IMPORTANCE OF INTERRUPT HANDLING IN SAFETY CRITICAL SYSTEMS

In safety critical systems, the correctness of operation is not only about *what* the system does, but also *when* it does it. A correct action performed too late can be as dangerous as a wrong action. Interrupts are the primary mechanism by which such systems become aware of urgent events in real time. Therefore, the speed and predictability of interrupt handling directly affects system safety.

Safety critical applications continuously interact with the physical world through sensors, actuators, and communication

interfaces. Events such as over-temperature, sudden pressure rise, obstacle detection, heartbeat irregularity, or braking command must be handled immediately. These events are usually signaled to the processor through interrupts.

If the processor fails to respond within the required time limit, the system may enter an unsafe state.

3.1 Time-Critical Nature of Safety Events

Many safety events occur unexpectedly and require reaction within strict deadlines measured in microseconds or milliseconds. Examples:

Table 1:

System/Application	Interrupt Source	Consequence of Delay
Airbag Control Unit	Crash sensor trigger	Airbag deploys late → severe injury
Anti-lock Braking System	Wheel speed sensor	Wheel lock → vehicle skidding
Cardiac Pacemaker	Heart rhythm sensor	Improper pacing → patient risk
Industrial Boiler Control	Pressure/temperature sensor	Explosion or equipment damage
Aircraft Flight Control	Gyroscope/IMU sensor	Loss of stability or wrong correction

These systems cannot rely on polling methods because polling wastes time and may miss urgent signals. Interrupts guarantee immediate notification.

3.2 Deterministic Response

Requirement

Safety standards such as **ISO 26262 (automotive)**, **DO-178C (avionics)**, and **IEC 61508 (industrial safety)** demand deterministic system behavior.

Deterministic means:

- Interrupt must be serviced within known maximum time
- Worst case latency must be calculable
- System response must be predictable under all conditions

Non-deterministic interrupt behavior makes system certification difficult.

3.3 Prevention of Hazardous Conditions

Interrupts often indicate fault or abnormal conditions:

- Overcurrent detection
- Overvoltage in power electronics
- Memory parity error
- Watchdog timeout
- Sensor malfunction
- Immediate ISR execution can:
- Shut down system safely
- Trigger alarms
- Switch to backup system
- Log error for diagnosis

Delay in handling such interrupts may allow fault to propagate.

3.4 Real-Time Control Loop

Dependency

Many safety systems operate using **closed-loop control**.

Example: In ABS braking,

1. Wheel sensor generates interrupt
2. ISR reads wheel speed
3. Controller adjusts brake pressure

If interrupt is delayed, controller works with old data, causing wrong actuation. Thus, control accuracy directly depends on interrupt response time.

3.5 Reduction of Data Loss

High-speed peripherals like ADCs, communication ports, and sensors generate data rapidly. If interrupts are not serviced quickly:

- Data registers may overflow
- Samples may be lost
- Communication frames may be corrupted

In medical monitoring or avionics telemetry, loss of such data is unacceptable.

4. FACTORS AFFECTING INTERRUPT LATENCY (ELABORATED)

Interrupt latency is not decided by a single element in the system. It is influenced by a combination of processor design, memory behavior, interrupt configuration, operating system policies, and ISR structure.

In safety critical systems, understanding these factors is necessary to predict and minimize the worst case interrupt response time.

4.1 Processor Architecture

The internal design of the processor has a direct impact on how fast it can respond to an interrupt.

CISC vs RISC Behavior

- **CISC (Complex Instruction Set Computer)** processors execute complex instructions that may take multiple clock cycles. If an interrupt occurs during such an instruction, the CPU must wait until the instruction completes. Also, context saving may involve many registers.
- **RISC (Reduced Instruction Set Computer)** processors use simple and fixed-length instructions that complete in fewer cycles. This reduces the waiting time before ISR entry.

Hardware Context Saving

Some modern processors (e.g., ARM Cortex-M) provide:

- Automatic stacking of registers
- Dedicated interrupt registers
- Shadow register sets

These features reduce software overhead and speed up interrupt entry.

Pipeline and Superscalar Effects

Deep pipelines and out-of-order execution may require pipeline flushing when

interrupt occurs, adding delay.

Key Point: Processors designed for embedded real-time use generally provide faster and more predictable interrupt response than general purpose CPUs.

4.2 Interrupt Prioritization

In practical systems, multiple interrupts may occur at the same time. If priorities are not configured correctly, critical interrupts may be forced to wait.

Priority Levels

Interrupt controllers allow assigning priority levels:

- High priority → serviced first
- Low priority → delayed

If a low-priority interrupt is being serviced and a high-priority interrupt occurs, the system should allow **preemption**.

Improper Priority Mapping

If priorities are assigned without analysis:

- Non-critical device may block critical sensor interrupt
- Worst case latency increases
- System behavior becomes unpredictable

Example: Display refresh interrupt blocking emergency shutdown interrupt.

4.3 Cache and Memory Access

Memory behavior significantly influences interrupt latency.

Interrupt Vector Table Location

If vector table is stored in:

- Flash memory → slower access
- SRAM → faster access

Cache Miss Penalty

When interrupt occurs:

- ISR code may not be present in cache
- Data used by ISR may not be cached
- Cache miss causes additional memory fetch cycles

Instruction and Data Cache Effects

Cold cache condition leads to longer ISR start time compared to warm cache.

In safety systems, designers sometimes lock ISR code into cache or place it in fast memory.

4.4 Operating System Overhead

In systems using RTOS, the operating system introduces additional layers between hardware and ISR.

Kernel Intervention

RTOS may:

- Save additional context
- Manage interrupt nesting
- Call scheduler after ISR completion

These operations add extra cycles.

Critical Sections and Interrupt Masking

RTOS uses critical sections where interrupts are temporarily disabled to protect shared resources. If interrupt occurs during this period, it must wait.

Long critical sections increase worst case interrupt latency.

Scheduling Decisions

Some RTOS perform task switching after ISR, adding delay before returning to main task.

4.5 Nested Interrupts

Nested interrupts allow higher priority interrupts to preempt lower priority ISRs. While this reduces latency for critical interrupts, it introduces complexity.

Stack Usage

Each nested interrupt requires additional context saving on stack.

Excessive nesting may:

- Increase memory usage
- Cause stack overflow
- Increase restoration time

Priority Inversion Possibility

Improper nesting may create scenarios where low priority ISR indirectly delays higher priority tasks.

Complex Timing Analysis

Calculating worst case latency becomes more difficult with nested interrupts.

5. HARDWARE TECHNIQUES FOR LOW LATENCY

5.1 Interrupt Vector Table in Fast Memory

Placing vector table in SRAM reduces fetch delay.

5.2 Dedicated Interrupt Controllers

Advanced interrupt controllers like NVIC (ARM) provide:

- Priority levels
- Fast vectoring
- Tail chaining

5.3 Direct Memory Access (DMA)

DMA reduces CPU load and interrupt frequency.

5.4 Fast Context Switching Registers

Some processors provide shadow registers for ISR.

6. SOFTWARE AND RTOS SUPPORT

6.1 Real Time Operating Systems

RTOS like FreeRTOS, VxWorks, QNX are designed for minimal latency.

Features:

- Preemptive scheduling
- Priority based ISR handling
- Minimal kernel intervention

6.2 ISR Design Rules

- Keep ISR short
- Avoid loops
- Avoid function calls
- Use flags for deferred processing

Table 2: Good vs Poor ISR Design

Good ISR Practice	Poor ISR Practice
Short execution time	Long calculations inside ISR
Use of volatile variables	Memory intensive operations
Defer tasks to main loop	Complex logic in ISR
Minimal stack usage	Deep function calls

7. Nested Interrupt Handling

Nested interrupts allow high priority interrupt to preempt low priority ISR.

Benefits:

- Critical task served first
 - Reduced worst case latency
- But requires careful design to avoid stack overflow and priority inversion.

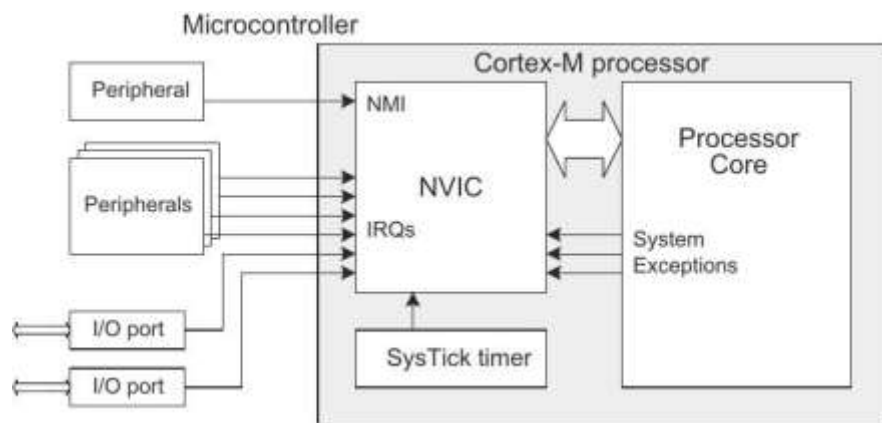


Figure 2: Nested Interrupt Concept

8. Deterministic Response and Worst Case Latency

Safety systems require not only fast response but predictable response.

Worst Case Interrupt Latency (WCIL)

must be calculated considering:

- Longest instruction execution
- Maximum disabled interrupt time
- Context save time

9. Compiler Level Optimizations

Compilers also play role.

- Inline small functions
- Optimize stack usage
- Avoid unnecessary memory access
- Use register variables

Improper compiler optimization can increase latency.

10. Role of Priority Inversion

If low priority task holds resource and high priority interrupt waits, latency increases.

Solutions include:

- Priority inheritance
- Resource locking protocols

11. Case Study: Automotive ABS

Controller

ABS system uses wheel speed sensors generating frequent interrupts.

Design considerations:

- High priority sensor interrupts
- Minimal ISR (only data capture)
- Processing in background task
- Dedicated interrupt lines

This ensures braking response is immediate.

12. Measurement of Interrupt Latency

Latency is measured using:

- Oscilloscope and GPIO toggle
- Logic analyzer
- Performance counters

Table 3: Comparison of Different Architectures

Architecture	Typical Latency	Features
ARM Cortex-M	Very Low	NVIC, tail chaining
AVR	Moderate	Simple vectoring
x86	Higher	Complex context saving
RISC-V	Low	Custom interrupt controllers

14. Challenges in Modern Systems

- Multicore interference
- Cache coherency delays
- Virtualization overhead
- Security checks (MPU/MMU)

These increase latency if not designed carefully.

15. Best Practices for Designers

1. Choose processor with fast interrupt support
2. Use RTOS with deterministic kernel
3. Optimize ISR code
4. Proper interrupt priority mapping
5. Measure and verify latency practically
6. Avoid disabling interrupts for long time

16. Future Trends

- Hardware assisted interrupt routing
- AI based scheduling in RTOS
- Chiplet based interrupt domains
- Deterministic multicore communication

17. Conclusion

Low latency interrupt handling is very important requirement in safety critical systems. Without deterministic and fast response, system may fail to perform its intended safety function.

Both hardware architecture and software design contribute to interrupt performance.

Proper ISR design, priority handling, RTOS support, compiler optimization and measurement techniques ensure minimal interrupt latency. Designers must consider worst case latency rather than average time.

With modern processors and intelligent design methods, it is possible to achieve very low interrupt delay suitable for automotive, medical, avionics and industrial applications. Future systems will focus more on hardware acceleration and predictable multicore interrupt handling.

References

1. J. Liu, *Real-Time Systems*, Pearson Education, 2000.
- A. Burns and A. Wellings, *Real-Time Systems and Programming Languages*, Addison Wesley, 2009.
2. ARM Ltd., *Cortex-M Interrupt Handling Guide*, Technical Manual, 2021.
3. K. Ramamritham, "Real-Time Interrupt Handling Mechanisms," *IEEE Computer*, vol. 31, no. 6, 2018.
4. R. Barry, *Using the FreeRTOS Real Time Kernel*, 2019.
5. P. Marwedel, *Embedded System Design*, Springer, 2017.
6. M. Joseph, *Real-Time Systems: Specification and Verification*, Prentice Hall, 2015.
7. S. Heath, *Embedded Systems Design*, Newnes, 2018.
8. NXP Semiconductors, *NVIC and Interrupt Latency Application Note*, 2020.
9. D. Simon, "Interrupt Latency Analysis in Safety Critical Applications," *IEEE Embedded Systems Letters*, 2022.