Interrupt in Zephyr OS

An interrupt service routine (ISR) is a function that executes asynchronously in response to a hardware or software interrupt. An ISR normally preempts the execution of the current thread, allowing the response to occur with very low overhead. Thread execution resumes only once all ISR work has been completed.

# Concepts of ISR (Interrupt Service Routine)

In Zephyr OS,an ISR is a special function that runs when something urgent happens, like a hardware device needing attention (e.g., a button press, a sensor value, etc.).

#### **Key Concepts:**

* **Interrupt Request (IRQ):** A hardware or software interrupt signal triggering the ISR.
* **Priority Levels:** Each IRQ has an associated priority.
* **Argument Value:** A parameter passed to the ISR to distinguish interrupt sources.especially useful if the same function is handling multiple interrupt sources.
* **IDT/Vector Table:** Associates an IRQ source with an ISR function.
* **Default ISR:** Handles unexpected interrupts by raising a fatal error.
* **Nesting:** Supports preemption of a running ISR by higher-priority interrupts.
* **Interrupt Context:** ISRs run in a kernel-provided context with a dedicated stack.

**Note:** Use *k\_is\_in\_isr()* API to detect if execution is in ISR or thread context.

Some kernel features or APIs can only be used by threads, not inside ISRs.  
 If you try to do something in an ISR that’s only allowed in a thread, the system could crash or misbehave.

So, before calling certain functions, you can check where your code is running using *k\_is\_in\_isr()*

## Multi-level Interrupt Handling

In Zephyr OS, a multi-level interrupt system refers to an interrupt architecture where more than one level (or layer) of interrupt controllers is used to handle and prioritize multiple interrupt sources.

#### Why Use Multi-level Interrupts?

Most modern embedded systems have more interrupt sources than the CPU can directly handle. To manage this:

* Interrupts are first handled by secondary (or even tertiary) interrupt controllers.
* These controllers group and prioritize interrupts, and then forward them to the primary interrupt controller, which finally signals the CPU.

Platforms with nested interrupt controllers must enable:

* *CONFIG\_MULTI\_LEVEL\_INTERRUPTS* – Enables support for multi-level interrupt handling.
* *CONFIG\_2ND\_LEVEL\_INTERRUPTS* – Enables handling of a second level of nested interrupts.
* CONFIG\_3RD\_LEVEL\_INTERRUPTS – Enables a third nested interrupt level.

#### Interrupt number-

Its unique 32-bit number for every interrupt in a multi-level interrupt tree. This number encodes the path from the top-level interrupt controller all the way down to the actual device.

**Example**:

**** 9 2 0

\_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ (LEVEL 1)

5 | A |

\_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ (LEVEL 2)

| C B

\_ \_ \_ \_ \_ \_ \_ (LEVEL 3)

D

Each level is encoded with an offset:

* A → 0x00000004

LEVEL 1 → line 4 (no nesting)

Device A is connected directly to LEVEL 1 on line 4

* B → 0x00000302

LEVEL 1 → line 9

LEVEL 2 → line 2

Device B is on line 2 of a LEVEL 2 controller that's on line 9 of LEVEL 1

* C → 0x00000409

LEVEL 1 → line 2

LEVEL 2 → line 9

Device C is on line 3 of a LEVEL 2 controller that's on line 2 of LEVEL 1

* D → 0x00030609

LEVEL 1 → line 9

LEVEL 2 → line 6

LEVEL 3 → line 3

Device D is on line 6 of a LEVEL 3 controller, inside a LEVEL 2 controller, inside LEVEL 1

### 

### **Preventing Interruptions**

Preventing interruptions means temporarily stopping hardware interrupts (IRQs) from being handled by the processor. This is done when a thread is executing a critical section — a part of the code that must not be interrupted to ensure data consistency, timing accuracy, or safe hardware access.

**Zephyr provides two main ways to prevent interruptions:**

#### **IRQ Locking (***irq\_lock()* **/** *irq\_unlock()***)**

* Disables all interrupts while a thread is running.
* The lock is thread-specific, meaning it only blocks interrupts during that thread’s execution.
* Interrupts are automatically re-enabled when the thread sleeps or switches out.

unsigned int key = irq\_lock();

// critical section

irq\_unlock(key);

When the thread is resumed and becomes current again, its previous lock state is restored, continuing the critical section. This mechanism ensures thread isolation without globally disabling interrupts across the system.

**Example Flowchart:**

****[Thread A Active] → [irq\_lock()] → [Critical Section Begins]

↓ ↓

[Thread Preempted] [Interrupts Disabled For A Only]

↓

[Thread B Runs] → [IRQ Functionality Unaffected for B]

↓

[Thread A Resumes] → [Critical Section Resumes]

↓

[irq\_unlock()] → [Interrupts Re-enabled for A]

#### **Disabling Specific IRQs** *(irq\_disable(irq\_number) / irq\_enable(irq\_number))*

* These functions are used to temporarily turn off and on a specific interrupt line (IRQ) in the system.
* *irq\_disable(irq\_number)*- It prevents the Interrupt Service Routine (ISR) tied to irq\_number from running, even if the hardware sends the signal.
* Use *irq\_enable(irq\_number)* to re-enable it
* Affects all threads system-wide.

irq\_disable(17);

// IRQ 17 is now disabled

irq\_enable(17);

// IRQ 17 is enabled again



### **Zero Latency Interrupts**

Zero Latency Interrupts are hardware interrupts that are configured to bypass the kernel’s interrupt masking (IRQ locking) mechanisms, allowing them to be serviced immediately without any delay, regardless of what the kernel is doing.

For time-critical applications:

Enable: CONFIG\_ZERO\_LATENCY\_IRQS

CONFIG\_ZERO\_LATENCY\_IRQS=y



* Use *IRQ\_ZERO\_LATENCY* flag with *IRQ\_CONNECT* or *IRQ\_DIRECT\_CONNECT*

IRQ\_CONNECT(IRQ\_LINE, PRIORITY, isr\_function, ISR\_ARG, IRQ\_ZERO\_LATENCY);

IRQ\_DIRECT\_CONNECT(IRQ\_LINE, PRIORITY, isr\_function, FLAGS);

**Caution:** Zero-latency ISRs must not interact with kernel APIs such as k\_sem\_give(), k\_fifo\_put(), or k\_work\_submit(). These functions rely on kernel infrastructure that is not safe to invoke from a zero-latency context. Using them may lead to undefined behavior, as zero-latency ISRs bypass kernel locking mechanisms and context awareness for maximum responsiveness.

**Architecture Specific:** Currently implemented for ARM Cortex-M.

### **Offloading ISR Work**

An ISR should execute quickly to ensure predictable system operation. If time consuming processing is required the ISR should offload some or all processing to a thread, thereby restoring the kernel’s ability to respond to other interrupts.

#### Why Offload?

* ISRs block other interrupts --> long ISRs = bad performance.
* Threads can run more complex code safely.
* Keeps the system predictable and responsive.

**Methods of Offloading:**

**1. Using Kernel Objects (FIFOs, Lifos, Semaphores)**

* Use FIFO/LIFO queues with *k\_fifo\_put() or k\_lifo\_put()*

void my\_isr(const void \*arg) {

static struct data\_item\_t item = { .value = 42 };

k\_fifo\_put(&my\_fifo, &item); // Send data to thread via FIFO

}

void my\_isr(const void \*arg) {

static struct data\_item\_t item = { .value = 99 };

k\_lifo\_put(&my\_lifo, &item); // Send data to thread via LIFO

}

* Use semaphores with *k\_sem\_give()*

void my\_isr(const void \*arg) {

k\_sem\_give(&my\_semaphore); // Wake up a thread to handle the work

}



**Diagram: ISR Offloading using Semaphore**

****[Interrupt Signal]

↓

[ISR Entry Point]

↓

[k\_sem\_give() called]

↓

[ISR Exits]

↓

[Thread waiting on Semaphore]

↓

[Thread Executes Deferred Work]

This ensures quick ISR execution and maintains overall system responsiveness.

#### **2. Using Workqueue Threads**

* A **workqueue** allows the ISR to schedule work to be done by a dedicated worker thread. This is a more flexible mechanism and supports multiple worker threads for concurrent execution.
* Work items are processed asynchronously and may be handled by the system's workqueue thread.

struct k\_work my\_work;

void work\_handler(struct k\_work \*item) {

// This function will run in a thread context

printk("Work item processed\n");

}

void isr\_handler(const struct device \*dev, struct gpio\_callback \*cb, uint32\_t pins) {

k\_work\_submit(&my\_work); // Offload work to the workqueue

}

void main(void) {

k\_work\_init(&my\_work, work\_handler);

}



### **Sharing Interrupt Lines**

On some hardware platforms (especially embedded SoCs), multiple peripherals share the same IRQ line. A single interrupt number may be used by multiple hardware blocks (e.g., DMA, audio, timers)

#### **Example Scenario:**

Suppose we have:

* **DMA Controller** using IRQ 17 for transfer completion.
* **DAI Controller** using IRQ 17 for FIFO watermark alert.

Enable interrupt sharing with:

IRQ\_CONNECT(17, 0, dma\_isr, NULL, 0);

IRQ\_CONNECT(17, 0, dai\_isr, NULL, 0);

In shared interrupt mode, both *dma\_isr* and *dai\_isr* will be invoked when IRQ 17 is signaled. Each ISR checks hardware status to determine the cause.

**ISR Execution Timeline Example:**

****Time(ms) Event

-------- ----------------------------

0 IRQ 17 triggered by DMA

1 dma\_isr() executes

2 IRQ 17 triggered again by DAI

3 dai\_isr() executes

**Diagram:**

**** +-------------+

| IRQ Line | <-- IRQ 17

+------+------+------------------+

| |

+---v---+ +----v----+

| DMA | | DAI |

| Ctrlr | | Ctrlr |

+-------+ +---------+

Shared ISR Handler

|

+---------+----------+

| ISR |

| Checks source |

+--------------------+

**Reminder:** Ensure *CONFIG\_SHARED\_INTERRUPTS* is enabled.

This system scales poorly with many clients, so use shared interrupts with care.

* Enable *CONFIG\_SHARED\_INTERRUPTS*
* Use *IRQ\_CONNECT* or *irq\_connect\_dynamic()*

#### APIs used -

***irq\_connect\_dynamic() -***

Used when you want to register a new ISR on a (possibly shared) interrupt line at runtime.

int irq\_connect\_dynamic(unsigned int irq\_p,

unsigned int priority,

void (\*routine)(const void \*parameter),

const void \*parameter,

uint32\_t flags);



***irq\_disconnect\_dynamic()*-**

Used to dynamically remove an ISR from a shared IRQ line.

void irq\_disconnect\_dynamic(unsigned int irq\_p,

void (\*routine)(const void \*parameter),

const void \*parameter);



**irq\_enable() / irq\_disable()-**

Used to manually control whether a specific IRQ line is active.

void irq\_enable(unsigned int irq\_p);

void irq\_disable(unsigned int irq\_p);



#### **Kconfig Macros for Shared Interrupts**

1. CONFIG\_SHARED\_INTERRUPTS

* Enables shared interrupt support in Zephyr.

2. CONFIG\_DYNAMIC\_INTERRUPTS

* Enables dynamic ISR connection/disconnection at runtime.

3. CONFIG\_SHARED\_IRQ\_MAX\_NUM\_CLIENTS

* Sets the max number of ISR clients per shared IRQ line.

4. CONFIG\_IRQ\_OFFLOAD(Optional)

* Enables offloading interrupt handling to a thread context

### **Summary**

Zephyr OS provides a powerful and flexible interrupt management system suitable for complex real-time applications. From multi-level nested interrupts to zero-latency priorities, developers have fine-grained control over ISR behavior.