Zephyr RTOS Threading Model using the shared link:
🔗 [Zephyr Project - Kernel Threads]

🧠 What Are Threads in Zephyr?

In **Zephyr**, a **thread** is the basic unit of execution. Unlike traditional bare-metal programming where code runs linearly, Zephyr uses **multithreading**, enabling **concurrent execution of tasks**.

Every running task, whether it's user-defined, system-defined, or part of the OS internals, is a **thread** managed by the **kernel scheduler**.

🧩 Core Concepts Covered

1. 🧵 Thread Lifecycle

Threads in Zephyr go through **various states** in their lifecycle:

State	Meaning
READY	Waiting to be scheduled on CPU
RUNNING	Currently executing
PENDING	Blocked, waiting on an event (semaphore, queue, etc.)
SUSPENDED	Not eligible to run, must be resumed
DEAD	Thread has exited or been aborted
DUMMY	Placeholder for special contexts (e.g., ISRs)

Transitions between states occur due to API calls (e.g., k_sleep()), interrupts, or scheduler decisions.

2. 🎛️ Creating Threads

There are two main ways to create threads:

➤ **Static Creation** (at build time)

```
K_THREAD_DEFINE(my_tid, STACK_SIZE,        my_entry_fn, arg1, arg2, arg3,
                PRIORITY, OPTIONS, START_DELAY);
```

- Defined at compile time
- Runs automatically after optional delay
- Very common in embedded systems

➤ **Dynamic Creation (at runtime)**

k_tid_t my_tid = k_thread_create(&my_thread_data,

my_stack_area,

STACK_SIZE,

my_entry_fn,

arg1, arg2, arg3,

PRIORITY, OPTIONS,

START_DELAY);

- Allows more flexibility
- Useful for apps that need variable or short-lived threads

3. 🎛️ Thread Attributes

Each thread has a set of configurable parameters:

Attribute	Description
Stack size	Defined via K_THREAD_STACK_DEFINE()
Entry point	Function that executes when thread runs
Priority	Affects scheduling order
Start delay	Optional delay before first run
Options	Cooperative vs. preemptive, floating-point use, etc.
User mode flag	Used in memory protection setups (MMU/MPU)

Example for defining stack statically:

K_THREAD_STACK_DEFINE(my_stack_area, 1024);

4. 🧠 Preemptive vs. Cooperative Threads

Zephyr threads can behave differently depending on their priority:

- **Preemptive** (default): Thread can be interrupted by a higher-priority thread
- **Cooperative**: Thread must yield explicitly using k_yield() or k_sleep()

Configured via thread priority:

- Priorities < CONFIG_NUM_COOP_PRIORITIES → Preemptive
- Priorities ≥ CONFIG_NUM_COOP_PRIORITIES → Cooperative

5. ⚙️ Controlling Threads

Zephyr provides APIs to manage thread behavior:

API	Purpose
k_thread_suspend()	Manually suspend a thread
k_thread_resume()	Resume a suspended thread
k_thread_abort()	Kill a thread
k_yield()	Yield to other threads
k_sleep()	Delay execution
k_thread_start()	Start a thread that was created with delay

6. 👮 Thread Safety & Synchronization

When using threads, shared resources must be protected using **synchronization primitives** like:

- **Semaphores** (k_sem)
- **Mutexes** (k_mutex)
- **Message Queues**, **FIFOs**, and **LIFOs**
- **Events** (k_event)

Example:

k_mutex_lock(&my_mutex, K_FOREVER);

// critical section

k_mutex_unlock(&my_mutex);

7. 📄 Thread Stacks

Each thread must have a **dedicated stack**:

- Defined using macros like **K_THREAD_STACK_DEFINE**()
- Size must be carefully selected (typically 512–2048 bytes)
- **Stack overflows** can cause undefined behavior — use CONFIG_THREAD_STACK_INFO to debug

Example:

K_THREAD_STACK_DEFINE(my_stack, 1024);

8. 🛡️ User Mode Support (Optional)

If your system has memory protection (MMU/MPU), threads can run in:

- **Kernel mode** (full access)
- **User mode** (restricted access)

Use **k_thread_user_mode_enter**() to transition into user mode after startup.

9. 🔀 Thread Migration and SMP

If CONFIG_SMP is enabled:

- Threads can run on multiple CPUs
- Threads can be pinned to specific cores using **CPU affinity**

k_thread_cpu_mask_clear(tid);

k_thread_cpu_mask_enable(tid, 1); // run only on CPU1

📋 Summary Table

Feature	Description
Thread types	Preemptive, Cooperative
Creation	Static (K_THREAD_DEFINE) or Dynamic (k_thread_create)
Management	Suspend, Resume, Abort, Start
Synchronization	Semaphores, Mutexes, FIFOs, Events
Stack allocation	K_THREAD_STACK_DEFINE or heap
SMP support	Thread CPU pinning
Debugging	Thread analyzer, names, runtime status

✅ Best Practices

- Always use **synchronization primitives** for shared data.
- Keep **thread stacks small** and test with stack usage analyzer.
- Use **priorities wisely** to ensure real-time responsiveness.
- Avoid **aborting threads** unless absolutely necessary.
- Use **thread names** for better debug visibility.

💡 Real-World Example

Imagine this scenario:

- A **sensor thread** reads data every 500ms.
- A **Bluetooth thread** sends data.
- A **logging thread** writes to UART.

You would define each thread with different priorities and use semaphores to protect shared sensor data. Logging would use the system work queue or a lower-priority thread.

🧩 Thread States Explained Briefly

State	Description
CREATED	Thread is defined but hasn't started executing yet
READY	Thread is ready to run and waiting for CPU
RUNNING	Currently executing thread
SLEEPING	Temporarily suspended for a time duration
PENDING	Waiting for a resource (mutex, semaphore, etc.)
SUSPENDED	Suspended manually or due to error — must be resumed
DEAD	Thread has finished or was aborted
DUMMY (not shown)	Special non-running state used for ISRs or placeholders

⚙️ Common Transitions

- **READY → RUNNING**: When scheduler picks the thread
- **RUNNING → SLEEPING**: Voluntarily sleeps (k_sleep())
- **RUNNING → PENDING**: Waiting on a resource (k_sem_take(), k_msgq_get())
- **RUNNING → READY**: Yields or gets preempted
- **RUNNING → SUSPENDED**: Suspended via k_thread_suspend()
- **ANY → DEAD**: Ends naturally or aborted
- **SUSPENDED → READY**: Resumed via k_thread_resume()

🧠 Overview: What Is Scheduling in Zephyr?

Scheduling in Zephyr refers to how the **kernel decides which thread should run** at any given moment. Since Zephyr is a **real-time operating system (RTOS)**, its scheduler is designed for **deterministic, priority-driven, and low-latency decision making**.

Every thread in the system (user, system, or driver) competes for CPU time. The scheduler ensures that the most critical task runs at the right time, maintaining **real-time responsiveness**.

🧩 Core Concepts of Zephyr Scheduling

1. 🔢 Thread Priority Levels

Zephyr uses **priority-based scheduling**, where **lower numerical values represent higher priority**.

- **Priorities range from 0 (highest) to CONFIG_NUM_COOP_PRIORITIES + CONFIG_NUM_PREEMPT_PRIORITIES - 1 (lowest).**
- There are two categories:
  - **Preemptible** threads (default): can be interrupted by higher-priority threads
  - **Cooperative** threads: must yield control voluntarily

📌 **Preemptive** priorities come first in the range, followed by **cooperative** ones.

K_THREAD_DEFINE(my_tid, STACK_SIZE, my_func, NULL, NULL, NULL, 2, 0, 0);

// If CONFIG_NUM_COOP_PRIORITIES = 16, priority 2 is preemptible

2. 🤹 Preemptive vs Cooperative Threads

Type	Description
Preemptive	Automatically preempted if a higher-priority thread is ready
Cooperative	Never preempted; must call k_yield(), k_sleep() or block explicitly

Cooperative threads are useful in time-critical sections where you don't want interrupts.

✅ Configure the number of cooperative threads using:

CONFIG_NUM_COOP_PRIORITIES

3. 🕛 Time Slicing (Round-Robin within Priority)

If **multiple threads** share the **same priority**,

Zephyr supports **round-robin scheduling** via **time slicing**.

- Enabled via:

CONFIG_TIMESLICING=y

CONFIG_TIMESLICE_SIZE=10 // in milliseconds

CONFIG_TIMESLICE_PRIORITY=0 // applies to threads at priority >= 0

- Each thread gets a fixed **time slice** to execute, after which the scheduler **rotates** to the next ready thread at that priority level.

✅ Helps in preventing thread starvation within the same priority class.

4. 🧵 Thread States & Transitions

Threads can be in one of several states:

State	Description
READY	Eligible to run; waiting for CPU
RUNNING	Currently executing
PENDING	Waiting on a semaphore, event, or delay
SUSPENDED	Suspended explicitly or due to resource unavailability
DEAD	Thread has finished execution or was aborted

Common transitions:

- READY → RUNNING: When the thread is scheduled
- RUNNING → PENDING: When calling k_sem_take() or k_sleep()
- RUNNING → READY: If the thread yields (k_yield()), is preempted, or time slice expires

5. 🔀 Context Switching

Context switch is the kernel's act of saving the current thread's state and restoring another thread's state.

Trigger conditions:

- A higher-priority thread becomes READY
- Time slice expires for current thread (if enabled)
- Current thread calls **k_yield**() or blocks on a resource

Zephyr uses efficient **lockless mechanisms** and **interrupt-driven preemption** to achieve fast context switches, critical for real-time constraints.

6. 🧠 Symmetric Multi-Processing (SMP) Support

If your system has multiple CPU cores, Zephyr supports **SMP scheduling**:

- Enabled via:

CONFIG_SMP=y

- Features:
  - Uses a **global run queue**
  - Scheduler can choose the best CPU to run a thread
  - Threads can be **pinned to specific cores** using CPU affinity:

k_thread_cpu_mask_clear(thread);

k_thread_cpu_mask_enable(thread, 1); // Allow on CPU 1 only

8. 📊 Thread Scheduling Order (Internals)

- All **READY threads** are stored in **priority-based queues**
- The **scheduler scans** from **highest to lowest priority**
- If **multiple threads have the same priority**, it uses **round-robin** (if time slicing is enabled)
- If **SMP**, it finds the **best CPU core** to dispatch the thread

9. 🧪 Debugging Scheduling Behavior

Zephyr provides tools to **inspect scheduling and threads**:

Tool/API	Use
k_thread_foreach()	List all threads and their state
k_thread_state_str()	Print state of a thread
thread_analyzer_run()	Analyze stack usage
CONFIG_THREAD_NAME	Helps identify threads in logs

Example:

```
void list_all_threads(void) {
    k_thread_foreach(print_info, NULL);
}
```

⚙️ Important Configuration Options

Kconfig Option	Purpose
CONFIG_NUM_COOP_PRIORITIES	Number of cooperative priorities
CONFIG_NUM_PREEMPT_PRIORITIES	Number of preemptive priorities
CONFIG_TIMESLICING	Enables round-robin within same-priority threads
CONFIG_TIMESLICE_SIZE	Slice duration in milliseconds
CONFIG_SMP	Enables SMP support
CONFIG_SCHED_CPU_MASK	Enables CPU affinity masking

✅ Summary Table

Feature	Description
Priority Scheduling	Lower number = higher priority
Preemptive Scheduling	Most threads, can be interrupted
Cooperative Scheduling	Needs explicit yielding
Time Slicing	Round-robin within priority level
SMP Support	Multi-core aware scheduling
CPU Affinity	Pin thread to CPU
Scheduler Locking	Temporarily prevent preemption
Thread States	READY, RUNNING, PENDING, SUSPENDED, DEAD
Context Switching	Auto or manual depending on state/event

🧠 Real-World Example

Imagine this setup:

- **Sensor Thread** (Priority 2): Reads sensor data periodically
- **Logger Thread** (Priority 5): Logs data to UART
- **UI Thread** (Priority 10): Updates a screen
- **Idle Thread** (Lowest priority): Power-saving mode

With **time slicing enabled**, the **UI and Logger** can share CPU time when **Sensor Thread sleeps**. Scheduler ensures **Sensor Thread** always preempts others to meet its real-time requirements.

🟦 Need More?

In **Zephyr RTOS**, the scheduler is a **core part of the kernel** that manages thread execution. While Zephyr primarily uses a **single main scheduling strategy**, it supports various **configurable features and scheduling behaviors** to suit different real-time and multi-core requirements.

🧠 Types of Scheduling in Zephyr

Zephyr uses a **priority-based, preemptive scheduler** by default, but it supports the following **types or modes** of scheduling:

1. 🏁 Priority-Based Preemptive Scheduling (Default)

- **Type**: Deterministic, real-time, priority-driven
- **Core mechanism** used in Zephyr
- Always runs the **highest priority READY thread**
- **Preemption**: If a higher-priority thread becomes READY, it preempts the current one immediately

✅ Used in:

- Real-time control
- Sensor fusion
- Interrupt-driven applications

🔧 Controlled via:

CONFIG_NUM_PREEMPT_PRIORITIES

2. 🧍 Cooperative Scheduling

- Cooperative threads **must yield** control voluntarily (they don't get preempted)

- Scheduler won't interrupt cooperative threads unless they call:
  - k_yield()
  - k_sleep()
  - Block on a resource (k_sem_take() etc.)
- Good for **non-interruptible critical sections**

🔧 Defined by setting a thread's priority to ≥ CONFIG_NUM_COOP_PRIORITIES

3. 🔁 Round-Robin (Time-Sliced) Scheduling

- Allows **fairness** among threads with **equal priority**
- Enabled with:

CONFIG_TIMESLICING=y

CONFIG_TIMESLICE_SIZE=<ms>

- Threads of the same priority will **share CPU time** in slices
- Scheduler rotates threads after each slice expiration

✅ Useful when multiple background tasks with same priority must run cooperatively

4. 🔀 Symmetric Multi-Processing (SMP) Scheduling

- Applicable when CONFIG_SMP=y (multi-core CPUs)
- Zephyr uses a **global ready queue** shared across CPUs
- Threads can:
  - Run on any CPU (by default)
  - Be **affined** to specific CPUs using CONFIG_SCHED_CPU_MASK

⚠️ SMP-aware scheduling requires **architecture support** (e.g., x86, ARM Cortex-A, or Cortex-R/M with multiple cores)

5. 🛡️ Scheduler Locking

- While not a separate scheduler, **locking and unlocking** the scheduler can **alter scheduling behavior temporarily**
- Use k_sched_lock() and k_sched_unlock() to **prevent context switches**
- Ensures a thread runs **atomically** without being preempted

✅ Useful in critical regions where **no interruption is allowed**

6. 🧠 Custom Scheduling Features

While Zephyr doesn't expose "pluggable" schedulers like Linux CFS or FreeRTOS optional policies, it supports:

Feature	Function
Priority queues	Fast lookup of highest-priority threads
Ready queues	Efficient per-priority scheduling queues
Tickless kernel	Reduces unnecessary timer interrupts
Thread CPU masks	Control thread placement in SMP
Lazy scheduling	Some threads (like system workqueue) are triggered only when needed

🔧 Not Supported (By Design)

Zephyr **does not currently support**:

Scheduler Type	Reason
Round-robin across priorities	Breaks real-time guarantees
Deadline or EDF scheduling	Not deterministic enough for embedded RTOS use cases
Completely dynamic/pluggable schedulers	Adds complexity, unsuitable for embedded resource limits

🧪 Summary Table

Scheduling Type	Description	Config Option(s)
Preemptive	Always runs highest-priority thread	CONFIG_NUM_PREEMPT_PRIORITIES
Cooperative	Thread must yield voluntarily	CONFIG_NUM_COOP_PRIORITIES
Round-Robin	Same-priority thread sharing	CONFIG_TIMESLICING, CONFIG_TIMESLICE_SIZE
SMP Scheduling	Multi-core scheduling	CONFIG_SMP, CONFIG_SCHED_CPU_MASK
Scheduler Lock	Prevents switching temporarily	k_sched_lock(), k_sched_unlock()

✅ Best Practice

- Use **preemptive** threads for real-time tasks.
- Use **cooperative** threads for non-blocking continuous loops.
- Use **time slicing** for background service tasks that share the same priority.
- Enable **SMP scheduling** on multi-core platforms and use **CPU affinity** wisely.

Would you like a **diagram** that visualizes these scheduling types or a **code example** showing how they interact in a real application (e.g., cooperative + preemptive + time-slicing)?

🔗 **Source**: Zephyr Scheduling – CPU Idling

🧠 CPU Idling in Zephyr RTOS

✅ What is CPU Idling?

CPU **idling** is the mechanism by which Zephyr **puts the processor into a low-power state** when **no threads are ready to run**.

When the **scheduler finds no runnable threads**, instead of spinning or executing dummy code, the kernel **calls the idle thread**, which can put the CPU in a **power-saving idle mode** (like WFI, WFE, or hlt, depending on the architecture).

🧩 Core Concepts

Concept	Description
Idle Thread	A low-priority thread (typically the lowest) that runs when there is nothing else to execute
Power Efficiency	Allows the system to conserve energy during idle time
Architecture-Specific Instructions	Instructions like WFI (ARM), hlt (x86) are used to enter low-power state
Idle Hooks	Custom user-defined behavior during idle time
Multicore Support	On SMP systems, CPUs can idle independently

🔁 Idle Thread Behavior

1. The **scheduler** looks for a READY thread to run.
2. If **none** are available, it schedules the **idle thread**.
3. The idle thread:
   - Runs an **infinite loop**.
   - Calls a CPU instruction to **enter low-power mode**.
4. When an **interrupt** occurs (e.g. timer, GPIO), it wakes up the CPU.
5. Scheduler is re-invoked to select a READY thread.

🧱 Implementation Overview

Internally, the **idle thread** is a **kernel thread** with:

- The **lowest priority** (typically CONFIG_IDLE_THREAD_PRIORITY)
- Runs forever in a loop
- Architecture-specific code puts the core to sleep

⚙️ Configuration Options

Config	Description
CONFIG_ARCH_HAS_IDLE_ENTRY	Architecture provides custom idle entry code
CONFIG_TICKLESS_KERNEL	Reduces spurious wakeups (enhances CPU idle time)
CONFIG_SYS_POWER_MANAGEMENT	Enables system-wide power management
CONFIG_PM	Enables the Power Management subsystem (can override idle behavior)

🔄 Idle Thread on SMP (Multi-Core)

On systems with CONFIG_SMP=y:

- Each core has **its own idle thread**
- Cores can independently go into and out of idle mode
- Allows for **fine-grained power savings**

🔋 CPU Idle Entry Example (ARM Cortex-M)

```
void arch_cpu_idle(void){

  __asm__ volatile ("wfi");

}
```

This wfi instruction (Wait For Interrupt) **stops the CPU clock** until an interrupt occurs, conserving power.

📉 Benefits of CPU Idling

Benefit	Description
✅ Power Efficiency	Reduces CPU power consumption
✅ Thermal Management	Reduces heat generation on embedded SoCs
✅ Extended Battery Life	Crucial for IoT, wearable, and portable devices
✅ Responsiveness	Quickly wakes up on interrupt without polling

⚠️ Important Considerations

- Do not perform **blocking operations or heavy computation** in idle hooks.
- Idle thread should **never exit**—it must always be available.
- On systems with CONFIG_PM, the power management subsystem may override default idle behavior (e.g., enter deeper sleep modes).

🧪 Final Takeaways

- Zephyr **does not waste cycles** when idle — it enters **hardware-supported sleep modes**.
- CPU idling is key for **ultra-low power** embedded designs.
- Hooks allow **custom extensions** to idle behavior, but must be lightweight and non-blocking.
- On multicore (SMP), **each CPU can idle independently**.

Would you like:

- A sample program that demonstrates CPU idling using a low-power sensor thread?
- A visual representation of **Zephyr's thread states including idle**?

Let me know and I can generate it for you.

Great — you've shared a **sample Zephyr RTOS application** that creates two threads, both of which explicitly call k_cpu_idle() to **request CPU idle states** when there's no other runnable work. Let's now **analyze, annotate, and explain** the **CPU idle behavior** and its effect using this code.

🧠 Objective

This example demonstrates:

- How the **Zephyr scheduler** enters the **idle thread**.
- How **k_cpu_idle()** is used to **force CPU idling**.
- When and how **the CPU wakes up** (due to interrupts or sleep expiry).
- Differences between **k_cpu_idle()** and **k_sleep()**.

✅ Breakdown of the Code Behavior

📌 Main Function

```
void main(void){

   printk(" Main: CPU idling sample started with 2 threads\n");

   // Create both threads

}
```

- Simply initializes the two threads with equal priority.
- The scheduler will alternate between these threads **based on readiness and timing**.

🧵 Thread 1

```
void thread1(void *arg1, void *arg2, void *arg3){

   while (1) {

      printk(" Thread 1: Requesting CPU idle\n");

      k_cpu_idle();     // CPU is asked to enter idle

      printk(" Thread 1: Back from idle\n");

      k_sleep(K_SECONDS(1));

   }

}
```

- k_cpu_idle() → CPU **enters low-power idle mode** (if no other thread is READY).
- Wakes up on any **interrupt** (like system timer or another thread becoming ready).
- After waking, prints again and **sleeps for 1 second**.

🧵 Thread 2

```
void thread2(void *arg1, void *arg2, void *arg3){

   while (1) {
```

```
        printk(" Thread 2: Requesting CPU idle\n");

        k_cpu_idle();

        printk(" Thread 2: Back from idle\n");

        k_sleep(K_SECONDS(2));

    }

}
```

- Same as thread 1, but with **longer sleep**, creating **more idle time**.

🧰 Core Concept: k_cpu_idle() vs k_sleep()

Function	What it does
k_cpu_idle()	Immediately enters **idle thread mode** (low power) if **no other threads are ready**
k_sleep()	Puts the **current thread to sleep**, and allows the **scheduler** to pick other threads (or idle)

✅ Summary:

- k_cpu_idle() hints **"CPU is not needed now"**.
- It **only enters idle mode** if nothing else is ready.
- It's **not a sleep** — it's a **system-level hint**, not thread-specific suspension.

⚙️ How the Scheduler Handles This

1. Both threads start running, printing idle requests.
2. They call k_cpu_idle():
   - The **scheduler checks**: are any other threads READY?
   - If not → calls **arch_cpu_idle()**, which executes WFI or similar.
3. CPU halts until:
   - A **timer interrupt** wakes up sleeping thread.
   - Or any **external interrupt** arrives.
4. k_sleep() causes the thread to go into SLEEPING state.
   - It is removed from READY queue.
   - **Idle thread runs** in the meantime.

🔍 Example Console Output (What You Might See)

Main: CPU idling sample started with 2 threads

Thread 1: Requesting CPU idle

Thread 1: Back from idle

Thread 2: Requesting CPU idle

Thread 2: Back from idle

Thread 1: Requesting CPU idle

Thread 1: Back from idle

...

- **No output** is shown during k_cpu_idle() — CPU is in low-power wait.
- Wakeup occurs from **timer tick interrupt** when k_sleep() expires.
- Idle time is maximized especially when both threads are sleeping.

🔋 Power Optimization Aspect

When Zephyr detects no runnable threads, thanks to k_cpu_idle() or natural sleep:

- It enters arch_cpu_idle() → Executes WFI / hlt / platform-specific power-down.
- On a **battery-powered** system, this can **dramatically reduce consumption**.

This is critical for:

- IoT devices
- Wearables
- Sensor nodes
- Any duty-cycled embedded system

✅ Summary of Learning

Feature	Effect
k_cpu_idle()	Enters CPU idle if nothing else is ready
k_sleep()	Sleeps thread and triggers re-scheduling
Idle Thread	Runs when no threads are ready
Scheduler	Decides who runs next or enters idle
Power Saving	Enabled via CPU idle when system is not active

🟦 **Reference Link**: System Threads in Zephyr

This section in the Zephyr documentation describes **system threads**—threads that are **created and managed by the Zephyr kernel** internally. They're essential for internal operations like initialization, system logging, and power management.

🧵 What Are System Threads?

System threads are **built-in kernel threads** that provide background services needed by the OS itself. Unlike application threads (which you create with k_thread_create()), system threads are:

- Automatically created by Zephyr at boot.
- Have **predefined stack sizes, names, and priorities**.
- Used internally by **subsystems like logging, system initialization, and idle processing**.

🔍 List of Key System Threads

Here are the **most common system threads** in Zephyr:

System Thread	Description
main thread	Entry point for the application. Runs main()
idle thread(s)	Always present; runs when no other threads are ready. Enters CPU idle state.
logging thread	Processes deferred logging if CONFIG_LOG_PROCESS_THREAD is enabled
system workqueue	Processes work submitted to the kernel workqueue
init thread	Runs initialization routines post-boot
shell thread	Runs command-line interface if the shell is enabled
Bluetooth threads	If BT is enabled, there may be threads like bt_rx, bt_tx, etc.

✅ Core Concepts

1. **Main Thread**

- Runs the user-defined main() function.
- Created during kernel boot.
- **Default priority**: CONFIG_MAIN_THREAD_PRIORITY
- **Stack size**: CONFIG_MAIN_STACK_SIZE

```
void main(void) {

   // This is run inside the main thread

}
```

2. **Idle Thread**

- Each CPU core has **its own idle thread**.
- Lowest possible priority (0).
- Enters **low-power CPU idle state** (via k_cpu_idle()).
- Used when **no other thread is ready to run**.

Configurable by:

- CONFIG_IDLE_STACK_SIZE
- CONFIG_MULTITHREADING

3. **System Workqueue Thread**

- Handles background work like **deferred tasks, drivers, and services**.
- Runs k_work_submit(), etc.
- Only present if CONFIG_SYSTEM_WORKQUEUE is enabled.

Configurable by:

- CONFIG_SYSTEM_WORKQUEUE_STACK_SIZE
- CONFIG_SYSTEM_WORKQUEUE_PRIORITY

4. **Logging Thread**

- If logging is enabled (CONFIG_LOG), a separate thread may be created to **process log buffers** asynchronously.

Triggered by:

- CONFIG_LOG_PROCESS_THREAD
- Helps keep ISR and application fast (offloads log processing).

5. **Init Thread**

- Runs SYS_INIT() and DEVICE_INIT() handlers.
- Used for initializing drivers, subsystems, etc.
- Removed once initialization is complete.

6. **Shell Thread (Optional)**

- If CONFIG_SHELL is enabled, a thread handles shell input/output.
- Useful for CLI interaction over UART, USB, etc.

🔁 Thread Lifecycle Diagram (with System Threads)

⚙️ Configuration Options

Here are some important Kconfig symbols associated with system threads:

Symbol	Description
CONFIG_MAIN_STACK_SIZE	Stack size for main thread
CONFIG_MAIN_THREAD_PRIORITY	Priority for main thread
CONFIG_SYSTEM_WORKQUEUE_STACK_SIZE	Stack size for system workqueue thread
CONFIG_SYSTEM_WORKQUEUE_PRIORITY	Priority of workqueue
CONFIG_IDLE_STACK_SIZE	Stack size per idle thread
CONFIG_LOG_PROCESS_THREAD	Enables log processing thread
CONFIG_SHELL	Enables shell thread
CONFIG_MULTITHREADING	Must be enabled for most system threads

✅ Why System Threads Matter

- **Power Management**: Idle threads trigger CPU power-saving states.
- **Performance**: Log processing and deferred work can run in background threads.
- **Clean Startup**: Initialization thread ensures orderly bring-up of subsystems.
- **Maintainability**: Keeps your main() function clean — system services are isolated.

🧪 Debugging Tip

You can **list system threads at runtime** using:

```
#include <zephyr/kernel.h>


void print_threads(void) {
    struct k_thread *t;
    K_THREAD_FOREACH(t) {
        printk("Thread name: %s, Priority: %d, State: %d\n",
```

```
            k_thread_name_get(t),
            k_thread_priority_get(t),
            t->base.thread_state);
    }
}
```

Or via CONFIG_THREAD_MONITOR, which allows listing all thread info.

📝 Summary

System Thread	Purpose
Main	Runs user code (main())
Idle	Low-power wait when CPU is free
Logging	Async logging
Workqueue	Background tasks
Init	One-shot initialization
Shell	Optional CLI support

======

Absolutely! Let's go through all the details of **Zephyr Workqueues** from the official documentation link:

What Are Workqueues in Zephyr?

Workqueues in Zephyr are **mechanisms for deferring work** to be executed **outside of interrupt context**, typically in a **dedicated thread**.
They help offload time-consuming tasks from ISRs or main logic to avoid blocking or missing deadlines.

✅ Core Concepts of Workqueues

1. **k_work**

- The basic unit of deferred work in Zephyr.
- You define a work item (struct k_work) and associate it with a handler function.
- The handler runs in a thread context when submitted.

struct k_work my_work;

2. **System Workqueue**

- Provided by Zephyr out-of-the-box.
- Always available when CONFIG_SYSTEM_WORKQUEUE is enabled.
- Work items can be submitted via k_work_submit().

k_work_submit(&my_work);

3. **User-defined Workqueues**

- You can create your own workqueue (thread + queue).
- Useful if your system needs **task isolation**, **different priority**, or **real-time separation**.

struct k_work_q my_work_q;

K_THREAD_STACK_DEFINE(my_stack, STACK_SIZE);

k_work_q_start(&my_work_q, my_stack, STACK_SIZE, PRIORITY);

4. **Delayed Work**

- Schedule work to be executed after a delay.
- Useful for **timeouts**, **debouncing**, **retries**, etc.

struct k_work_delayable my_delayed_work;

k_work_schedule(&my_delayed_work, K_MSEC(1000)); // Runs after 1 second

🔧 API Functions

Basic Work

API	Description
k_work_init()	Initialize a work item
k_work_submit()	Submit work to system queue
k_work_busy_get()	Check if work is pending

Delayed Work

API	Description
k_work_schedule()	Schedule work with a delay
k_work_cancel_delayable()	Cancel delayed work
k_work_delayable_is_pending()	Check pending state

User Workqueue

API	Description
k_work_q_start()	Start a custom workqueue
k_work_queue_init()	Initialize queue before starting

⚙️ Kconfig Options

Option	Description
CONFIG_SYSTEM_WORKQUEUE	Enables system workqueue
CONFIG_SYSTEM_WORKQUEUE_STACK_SIZE	Stack size for workqueue thread
CONFIG_SYSTEM_WORKQUEUE_PRIORITY	Thread priority

🟦 Example Code

Simple Work Submission Example

```
#include <zephyr/kernel.h>
#include <zephyr/sys/printk.h>

// Work handler function
void work_handler(struct k_work *work) {
    printk("Running work handler\n");
}

// Work item definition
K_WORK_DEFINE(my_work, work_handler);

void main(void) {
    printk("Submitting work\n");
    k_work_submit(&my_work);
}
```

Delayed Work Example

```
#include <zephyr/kernel.h>
```

```
#include <zephyr/sys/printk.h>

// Delayed work handler
void delayed_handler(struct k_work *work) {
    printk("Delayed work executed\n");
}

K_WORK_DELAYABLE_DEFINE(my_delayed_work, delayed_handler);
void main(void) {
    printk("Scheduling delayed work for 2 seconds\n");
    k_work_schedule(&my_delayed_work, K_SECONDS(2));
}
```

📊 System vs Custom Workqueue

Feature	System Workqueue	Custom Workqueue
Setup	Ready-to-use	Needs initialization
Priority Control	Fixed (via Kconfig)	Full control
Stack Size	Configurable	Customizable
Isolation	Shared for all tasks	Can be isolated
Use Case	Simple offloads	RT separation, partitioned logic

⚠️ Important Notes

- **Work handlers run in thread context**, not ISR.
- Do not block indefinitely in a work handler.
- Delayed work is **not periodic** — reschedule it manually.

- Work cannot be re-submitted until it's completed unless canceled.

📍 Real-World Use Cases

Scenario	How Workqueues Help
Button press debounce	Schedule delayed work to confirm stability
Sensor polling	Submit periodic work from timer ISR
Logging	Offload to keep main loop responsive
Networking	Process packets in background thread

🧠 Summary

Concept	Purpose
k_work	Basic unit of deferred task
k_work_q	User-defined workqueue
k_work_submit()	Submit to system queue
k_work_schedule()	Delay work execution
k_work_cancel_delayable()	Stop a pending delayed task
CONFIG_SYSTEM_WORKQUEUE	Enables default kernel queue

🧵 What Is a Workqueue in Zephyr?

A **workqueue** is a kernel object that allows you to defer execution of tasks (called **work items**) to a dedicated thread. This is especially useful when:

- An **ISR (Interrupt Service Routine)** or high-priority thread needs to offload non-critical work.
- You want to avoid blocking or delaying time-sensitive operations.
- You need structured, asynchronous task execution.

🧩 Core Components

1. **Work Item (k_work)**

A work item is a unit of work that contains:

- A **handler function** to execute.
- Internal state (e.g., pending, running, canceled).
- Optional delay (for delayable work).

You must initialize it using:

k_work_init(&work_item, handler_function);

2. **Workqueue (k_work_q)**

A workqueue is a thread that:

- Waits for work items.
- Executes them in FIFO order.
- Yields between items to avoid starvation.

You can use the **system workqueue** or create your own:

struct k_work_q my_work_q;

k_work_queue_init(&my_work_q);

⚙️ How It Works

🔄 Lifecycle of a Work Item

1. **Initialization**: Set up the work item with a handler.
2. **Submission**: Submit to a workqueue using:
3. k_work_submit(&work_item); // System queue
4. k_work_submit_to_queue(&my_work_q, &work_item); // Custom queue
5. **Execution**: The workqueue thread runs the handler.
6. **Completion**: The item is marked as done.

🕒 Delayable Work

You can schedule work to run **after a timeout** using k_work_delayable:

k_work_init_delayable(&dwork, handler);

k_work_schedule(&dwork, K_SECONDS(5));

This is useful for:

- Retry mechanisms
- Deferred cleanup
- Time-based triggers

🧠 Best Practices

- Use workqueues to **offload non-urgent tasks** from ISRs.
- Avoid using k_work_busy_get() or k_work_is_pending() for synchronization — they can cause race conditions.
- Prefer k_work_flush() or k_work_cancel_sync() for safe handling.

🧪 Sample Use Case

Imagine a sensor ISR that captures data quickly and defers processing:

```
void sensor_isr(void) {

   k_work_submit(&process_sensor_data_work);

}
```

This keeps the ISR fast and responsive, while the workqueue handles the heavy lifting.

🧵 System Idle Integration

Workqueues integrate well with **system idle**:

- When no threads or work items are ready, the system enters idle.
- Workqueue threads sleep when their queue is empty.
- This helps conserve power in embedded systems.

🔍 What This Example Demonstrates

Context	What Happens
SYS_INIT	Prints from early init (before threads exist)
ISR	Triggered by button press, runs in no-thread
Thread	Runs regularly every 2s, prints thread context

🟦 Notes

- SYS_INIT(..., PRE_KERNEL_1, ...) ensures the init function runs **before kernel scheduling**.
- The button ISR uses k_current_get() to confirm it's running outside thread context.

Would you like a **visual diagram** of these execution contexts (SYS_INIT, ISR, thread), or a **version with UART interrupt** instead of GPIO?

Here's a complete Zephyr RTOS program that demonstrates the use of **initialization levels**: PRE_KERNEL_1, PRE_KERNEL_2, and POST_KERNEL. These levels are part of Zephyr's **system initialization framework**, which allows you to register functions that run at specific stages during system boot.

✅ Program Overview

This example registers three initialization functions at different levels:

- PRE_KERNEL_1: Runs before most kernel subsystems are initialized.
- PRE_KERNEL_2: Runs after basic kernel services but before devices.
- POST_KERNEL: Runs after the kernel is fully initialized and multitasking is enabled.

🧵 Zephyr Initialization Demo

🧠 Core Concepts Explained

🔹 SYS_INIT()

This macro registers a function to run during system boot. It takes three arguments:

SYS_INIT(function, level, priority);

- function: The init function to run.
- level: One of PRE_KERNEL_1, PRE_KERNEL_2, POST_KERNEL, APPLICATION.
- priority: Determines order within the level (lower = earlier).

🔹 Initialization Levels

Level	Description
PRE_KERNEL_1	Runs before kernel services (e.g., memory setup)
PRE_KERNEL_2	Runs after basic kernel services but before devices
POST_KERNEL	Runs after kernel and device drivers are initialized
APPLICATION	Runs after main() starts

🔗 [Zephyr Docs - No Threading]

🧩 What is “No Threading” Mode in Zephyr?

Zephyr RTOS typically runs **multiple threads** to manage concurrency. But sometimes—for **tiny embedded systems** with **limited memory or no need for concurrency**—you want the **simplest possible system**.

That's where CONFIG_MULTITHREADING=n comes in.

🚫 CONFIG_MULTITHREADING=n: Core Concept

When you **disable multithreading**:

- Zephyr runs **only the main() function**.
- No scheduler, no kernel threads, no ISRs spawning threads.
- Your program becomes **single-threaded**, like a **bare-metal loop**.

🎯 Why Use This Mode?

This configuration is great for:

Use Case	Why Use No-Threading Mode?
Tiny MCUs (8/16-bit or minimal 32-bit)	Saves RAM & flash; no scheduler overhead
Hard real-time loops	You want predictable execution timing
Startup-time testing	Use bare main() for test apps or boot logic
Secure Bootloaders	Minimal, predictable execution paths

🔧 What Works in No-Thread Mode?

Even without threading, many **kernel services** still work:

✅ Supported Features

Feature	Available?	Notes
main() execution	✅	Your logic runs inside this
Interrupts (ISRs)	✅	ISRs still work and can use most APIs
Spinlocks	✅	Safe for critical sections
Atomic operations	✅	For memory-safe flags and counters
Memory allocation	✅	k_malloc, k_free work
Device drivers	✅	Work if they don't depend on threads
Logging (non-threaded)	✅	Use polling or interrupt-based output

🚫 Not Supported

Feature	Status	Why
Threads (k_thread_create)	❌	Requires scheduler
Workqueues	❌	Threads are required to run work
Semaphores, mutexes	❌	These are thread synchronization primitives
Message queues, FIFOs	❌	Same reason—threads are needed
Thread APIs (priority, sleep)	❌	No thread context

🧠 How to Enable No-Threading Mode?

In your **prj.conf**:

CONFIG_MULTITHREADING=n

You can also **disable optional thread-related features** to reduce size further:

CONFIG_THREAD_NAME=n

CONFIG_THREAD_STACK_INFO=n

CONFIG_THREAD_CUSTOM_DATA=n

⚙️ Execution Flow

🔁 Instead of scheduling, Zephyr will:

- Boot into main()
- Call any init functions or ISRs triggered
- Remain inside main() or sleep-loop
- All work is done sequentially or via ISRs

🛠️ Example Code (No Threads)

```
#include <zephyr/kernel.h>
#include <zephyr/sys/printk.h>
#include <zephyr/device.h>
void main(void){
  printk("Zephyr running in no-thread mode\n");

  while (1) {
    printk("Main loop doing work...\n");

    // Simulate workload
    for (volatile int i = 0; i < 100000; i++);

    // Idle - since k_sleep is not supported, you can do NOP or use a timer manually
  }
}
```

💡 Integration with ISRs (Very Important!)

Since threads are disabled, **interrupt service routines (ISRs)** are the only form of asynchronous execution.

Example:

- ISRs must be short and fast.
- Cannot spawn threads or use k_work_submit.

🚀 Memory and Flash Size Savings

Disabling multithreading can **reduce kernel footprint significantly**:

Feature	Approximate Saving
Threading & scheduler	3–6 KB RAM
Workqueues, timers, sync	2–4 KB Flash
Context switching	Reduced overhead

This mode is ideal for ultra-constrained systems.

📋 Summary Table

Component	Supported in No-Thread Mode?	Notes
main()	✅	Single-entry execution
ISRs	✅	Only form of async code
Threads (k_thread_*)	❌	Scheduler removed
Workqueues	❌	No background thread context
Timers (k_timer)	❌	Requires scheduling
Logging (polling)	✅	Use non-threaded backends
Memory Allocation	✅	k_malloc, k_free available

- A **comparison** between CONFIG_MULTITHREADING=n vs y?

Zephyr RTOS stands out from other real-time operating systems (RTOSes) due to its **modular architecture**, **scalability**, and **deep integration with modern development tools**. Here's a detailed comparison highlighting how **Zephyr differs from other RTOSes** like FreeRTOS, ThreadX, and RTEMS:

🔍 1. Architecture & Design Philosophy

Feature	**Zephyr RTOS**	FreeRTOS / ThreadX / RTEMS
Modular kernel	Highly configurable via Kconfig	Less modular (FreeRTOS is monolithic)
Multi-architecture support	Supports 32+ architectures (x86, ARM, RISC-V, ARC, etc.)	FreeRTOS supports many, but not as unified
Device Tree usage	✅ Yes (like Linux)	❌ No (FreeRTOS, ThreadX use static config)
POSIX compatibility	Partial	FreeRTOS: minimal, ThreadX: limited

🧩 2. Threading & Scheduling

Feature	Zephyr	FreeRTOS	ThreadX	RTEMS
Preemptive scheduling	✅	✅	✅	✅
Cooperative threads	✅	✅	✅	✅
Thread priorities	0 (highest) to 255 (lowest)	0 (highest) to N	0 (highest) to 31	Configurable
Thread states	Rich: ready, running, sleeping, pending, suspended, dead	Simpler model	Similar to Zephyr	Similar

🧵 3. Kernel Services

Feature	Zephyr	FreeRTOS	ThreadX	RTEMS
Semaphores, Mutexes, Queues	✅	✅	✅	✅
Workqueues	✅ (system & custom)	❌ (manual task creation)	✅	✅
Timers	✅ (kernel + user)	✅	✅	✅
Memory management	Slab, heap, pool	Heap only	Byte pool	Region-based

🔐 4. Security & Safety

Feature	Zephyr	FreeRTOS	ThreadX	RTEMS
Memory protection (MMU/MPU)	✅	✅ (MPU only)	✅	✅
User/kernel mode separation	✅	❌	❌	✅
Trusted Execution (PSA)	✅	Partial	❌	❌
Safety certifications	In progress (ISO 26262, IEC 61508)	FreeRTOS+Safety	ThreadX: IEC 61508	RTEMS: DO-178C

🌐 5. Connectivity & Protocols

Zephyr includes a **rich networking stack**:

- IPv4/IPv6, TCP/UDP
- BLE, 802.15.4, LoRa, CAN
- MQTT, CoAP, LwM2M, HTTP(S)

FreeRTOS and others often require **external libraries** (e.g., FreeRTOS+TCP, LwIP).

🧪 6. Development Ecosystem

Feature	Zephyr	FreeRTOS	ThreadX	RTEMS
Build system	CMake + Kconfig	Make/CMake	Proprietary	Autotools
Device Tree	✅	❌	❌	❌
Tooling	West (meta-tool), Zephyr SDK	AWS tools	Azure RTOS Studio	GCC toolchain
Testing framework	Twister	Unity	Azure Test	Custom

🧠 7. Unique Features in Zephyr

- **Device Tree + Kconfig**: Linux-style hardware abstraction and configuration.
- **Unified build system**: CMake + Python-based west tool.
- **Multi-domain support**: Supports secure/non-secure domains (e.g., ARM TrustZone).
- **Subsystems**: Power management, logging, shell, file systems, USB, etc.

📘 Learn More

- 🔗 Zephyr RTOS Overview
- 🔗 Zephyr vs FreeRTOS Comparison (Community)

Would you like a **visual comparison chart** or a **recommendation** based on your use case (e.g., automotive, IoT, wearables)?