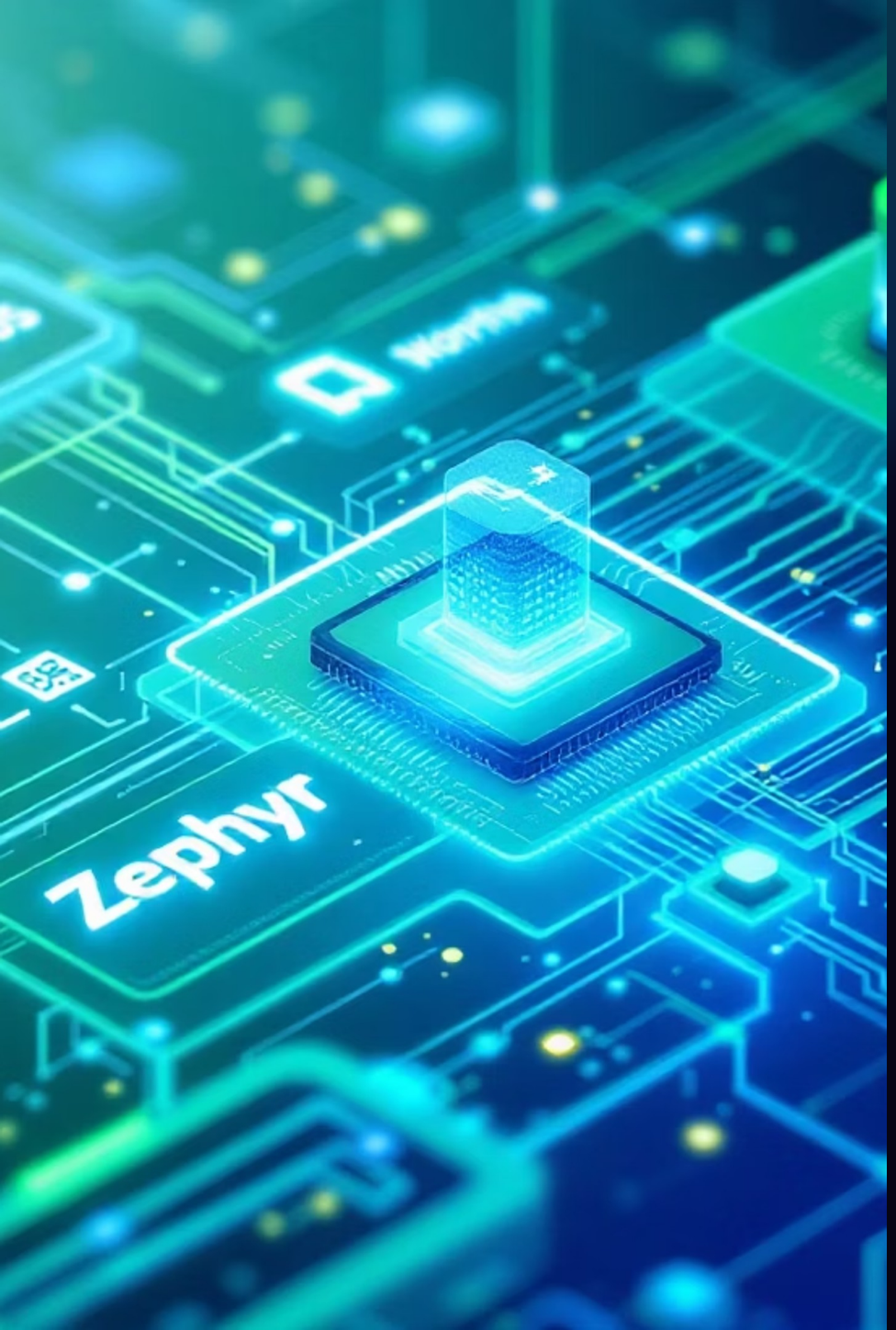


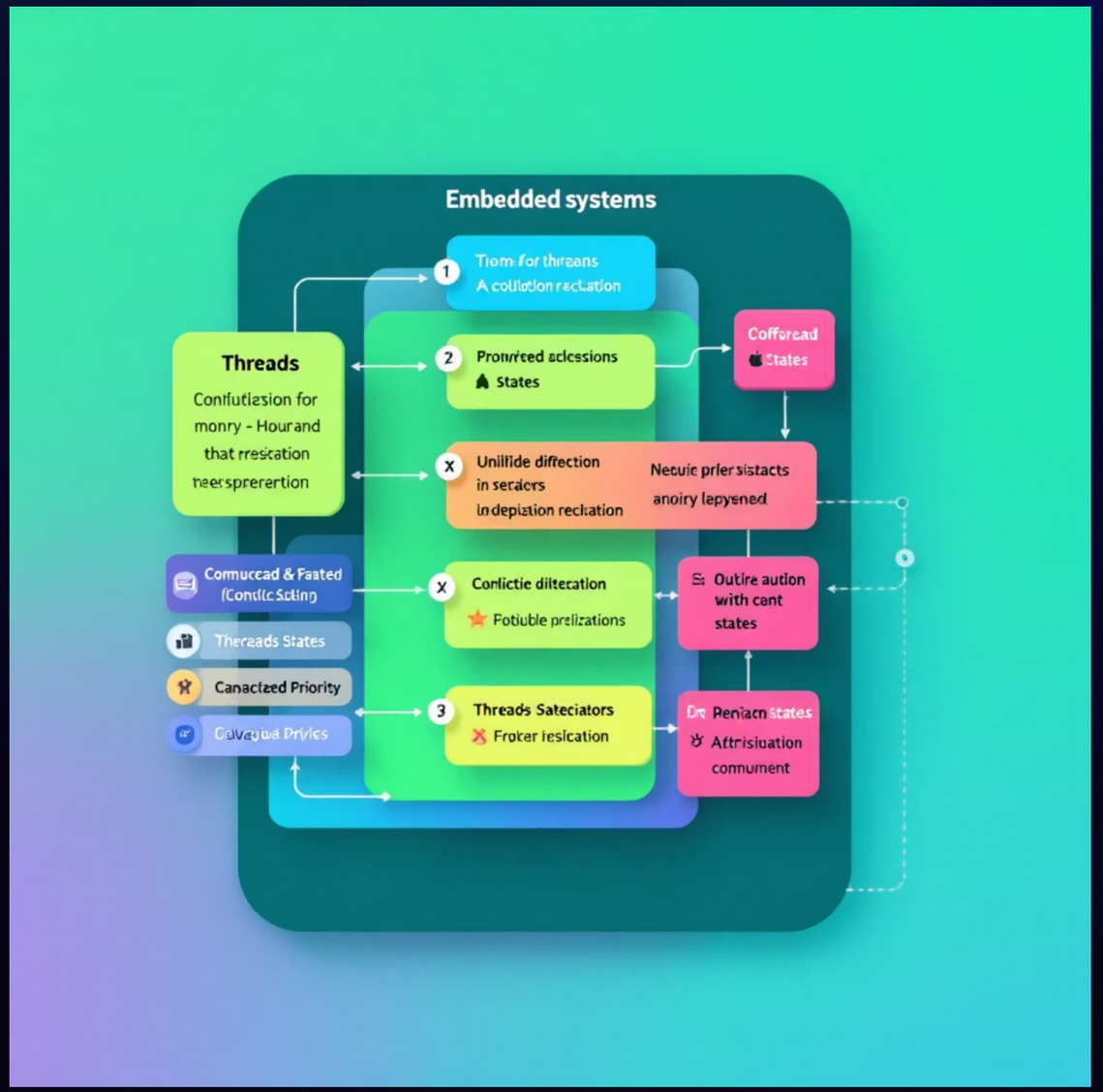
Zephyr Workshop – Threads & Workqueues

An in-depth exploration of Zephyr's threading model and workqueue system for efficient embedded application development

What is a Thread in Zephyr?

- Basic unit of execution in Zephyr
- Each thread has: stack, priority, entry point
- Can be preemptive or cooperative
- Used for independent, long-running tasks

Threads are ideal for independent, long-running tasks that may block or require significant processing time without affecting system responsiveness.



Thread API

Creation

```
k_thread_create()
```

Runtime thread creation with dynamic allocation. Thread parameters include stack pointer, entry function, and priority.

Definition

```
K_THREAD_DEFINE()
```

Static thread definition at compile time. Optimizes memory usage and startup performance.

Control

```
k_sleep()
```

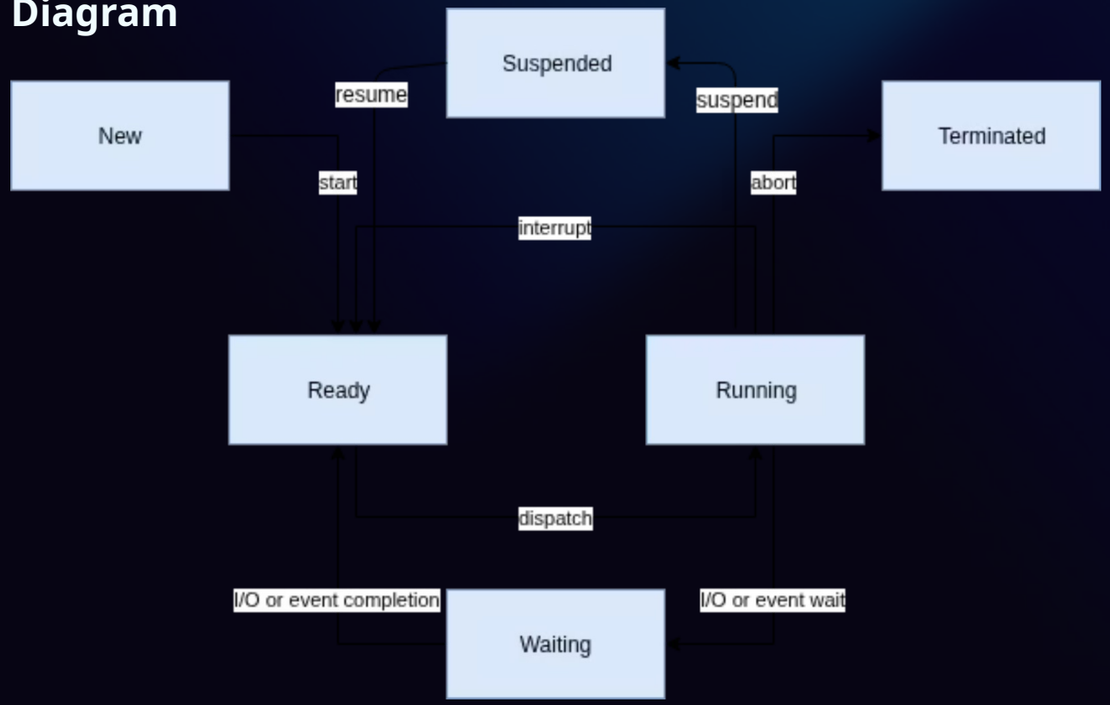
Pause thread execution for a specified duration. The scheduler will run other threads during this time.

Scheduling

Thread priority determines execution order. Higher priority threads preempt lower priority ones unless cooperative scheduling is used.

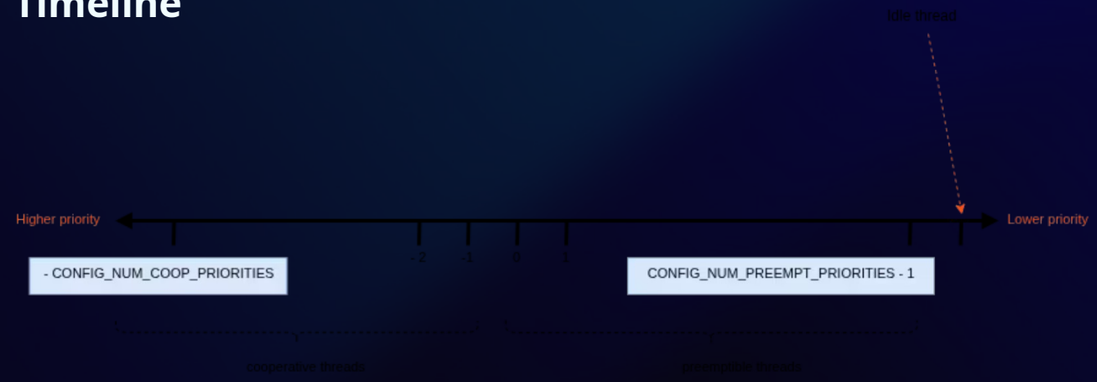
Thread States and Transitions

Thread State Diagram



Thread states include: Ready, Running, Pending, and Suspended. Transitions between states are managed by the scheduler based on priority and system events.

Thread Execution Timeline



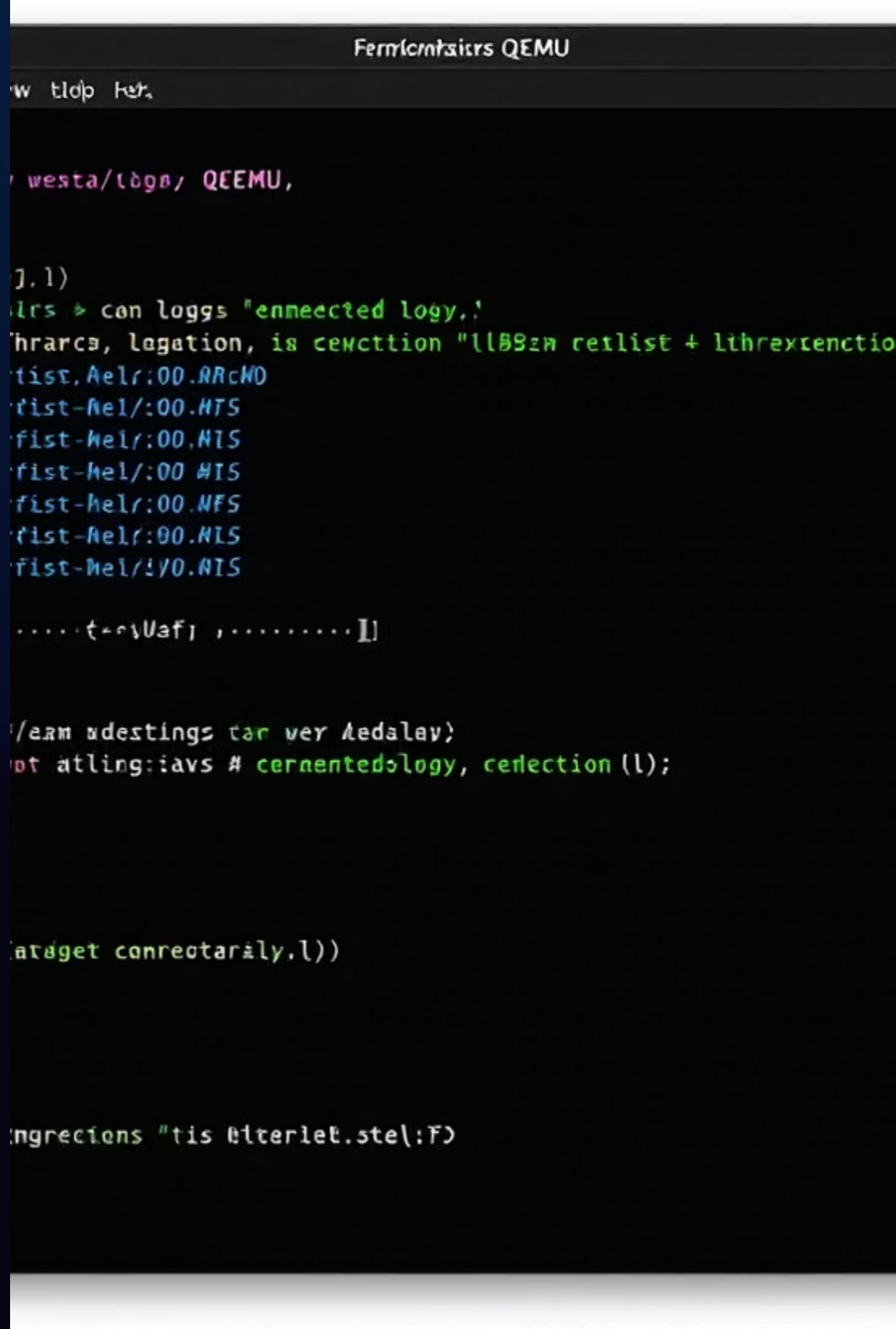
Zephyr's preemptive scheduler ensures high-priority threads receive immediate processor time, while cooperative threads voluntarily yield execution.

Basic Synchronization Demo

Running the Demo

```
west build -b qemu_x86 samples/synchronization
west build -t run
```

This demo runs on QEMU directly on your development machine, eliminating the need for physical hardware during initial learning.



What is a Workqueue?

Kernel Object

A workqueue is a kernel object that uses a dedicated thread to process work items in a FIFO (First In, First Out) manner.

Efficient

Avoids context switching overhead by batching multiple small tasks into a single thread execution context.

Deferred Execution

Provides a mechanism for scheduling small tasks to run later, often from interrupt context.

Lightweight

Uses significantly less resources than creating separate threads for small, periodic tasks.

Zephyr provides a system workqueue by default, but custom workqueues can be created with different priorities and stack sizes for specialized needs.

Workqueue API Concepts

Work Item Types

`struct k_work`: Basic work item for immediate or one-time deferred execution

`struct k_work_delayable`: Enhanced work item that can be scheduled to run after a specified time period

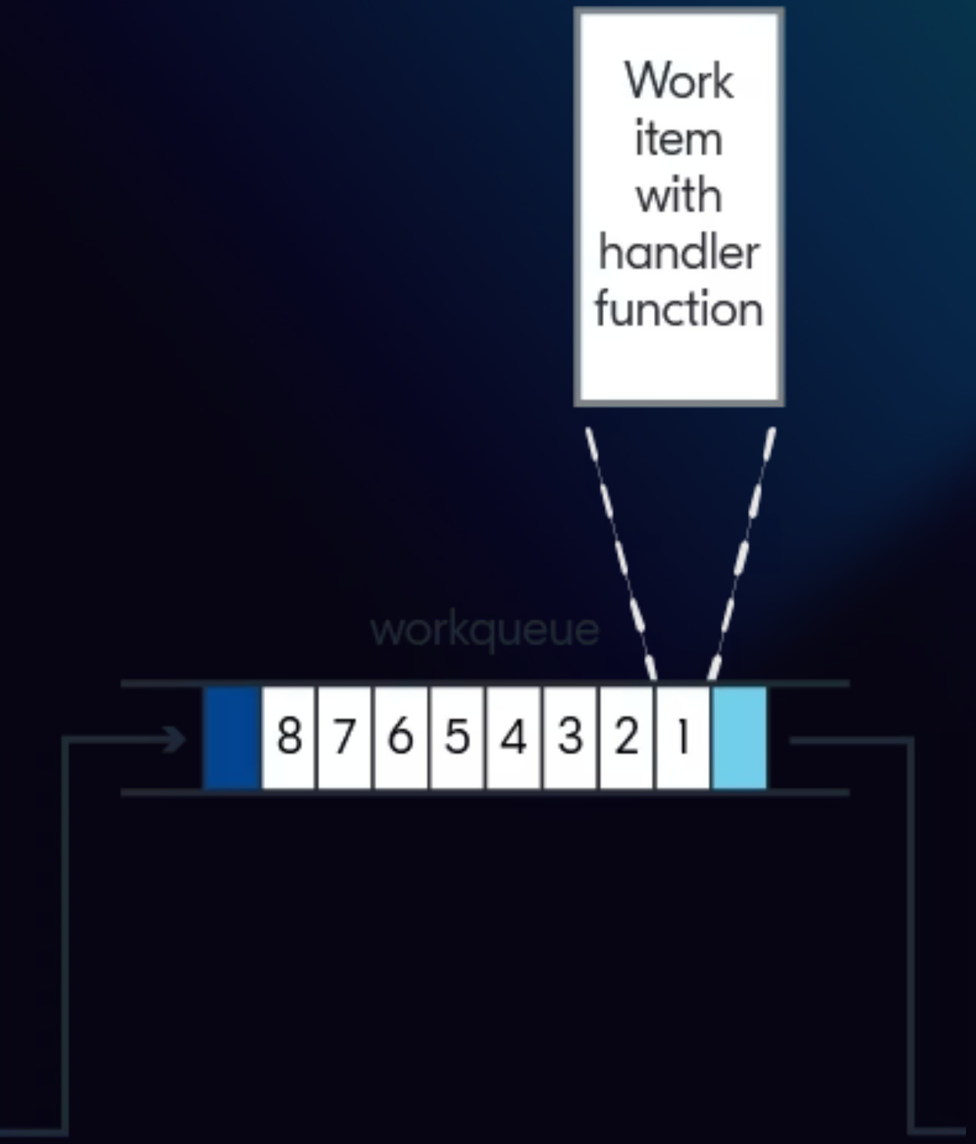
Both types execute their handler functions in the context of the workqueue thread.

Core Functions

```
k_work_init(&work, handler_function);k_work_submit(&work);k_work_schedule(&dwork, K_MSEC(100));
```

The handler function signature must be:

```
void handler(struct k_work *work);
```



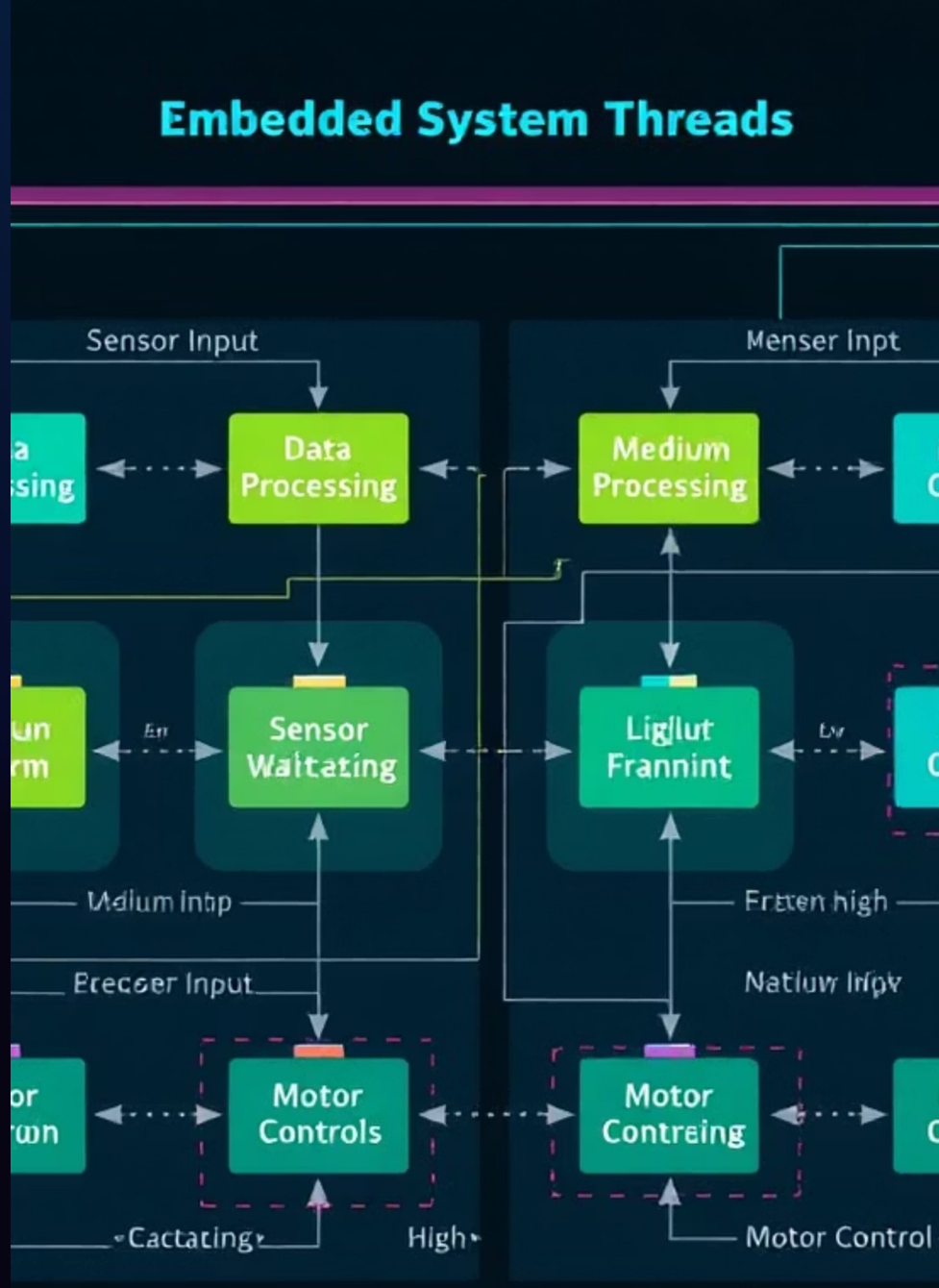
Workqueue Implementation

The diagram illustrates how work items flow through the system. When submitted, they enter a FIFO queue and are processed sequentially by the workqueue thread. Delayable work items are managed by a timing wheel that efficiently tracks when each item should be executed.

The system workqueue runs at priority 0 (highest) by default, making it suitable for time-sensitive operations that must execute promptly but are too short to justify a dedicated thread.

When to Use Threads

- Long-running background operations
- Independent state machines
- Polling or waiting logic



When to Use Workqueues

Short, Non-blocking Tasks

Brief operations that complete quickly and don't block. Examples include updating status indicators or triggering state changes.

Interrupt Deferral

Moving processing out of interrupt context to prevent extended ISR execution. Perfect for sensor data processing, message handling, or flash operations.

Periodic Maintenance

Regularly scheduled lightweight tasks like watchdog refreshing, connection keepalives, or system health checks.

Example Implementation

```
/* Define work item */K_WORK_DEFINE(process_work, process_handler);
/* In ISR */void sensor_isr(void) {
/* Read sensor data */ read_sensor_data();
/* Defer processing to workqueue */k_work_submit(&process_work);
}
/* Work handler */void process_handler(struct k_work *work) {
/* Process data outside ISR context */ analyze_sensor_data();
update_display();
}
```