Zephyr Shell Architecture Overview

The Zephyr Shell is a modular, Unix-like command-line interface designed for embedded systems. It enables developers to interact with their applications in real-time, facilitating tasks such as debugging, configuration, and monitoring.

Key Components:

1. **Shell Core:** Manages command parsing, execution, and user interaction.
2. **Backends:** Interfaces for input/output operations (e.g., UART, RTT, Bluetooth).
3. **Command Handlers:** Functions registered to handle specific shell commands.
4. **Logging Integration:** Allows shell output to be directed to Zephyr's logging system.

🔄 Command Processing Flow

The lifecycle of a shell command involves several stages:

1. **Initialization:**
    - The shell is initialized using shell_init(), specifying the backend and configuration parameters.
    - Backends are set up to handle input/output operations.
2. **Input Handling:**
    - User inputs are captured via the configured backend.
    - The shell processes input characters, handling features like line editing and command history.
3. **Parsing and Execution:**
    - Upon receiving a complete command line (typically terminated by Enter), the shell parses the input into command and arguments.
    - It searches for a matching command handler in the registered command set.
    - If found, the corresponding handler function is invoked with the parsed arguments.
4. **Output:**
    - Command output is sent back through the backend to the user.
    - Additionally, output can be routed through Zephyr's logging system for centralized logging.

🧱 Command Registration and Structure

- **Static Commands:** Defined at compile-time using macros like SHELL_CMD_REGISTER().
- **Dynamic Commands:** Registered at runtime, useful for applications requiring flexible command sets.

Example:

```
SHELL_CMD_REGISTER(version, NULL, "Prints the kernel version", cmd_version);
```

This registers a version command that, when invoked, calls the cmd_version handler function.

🔌 Backend Integration

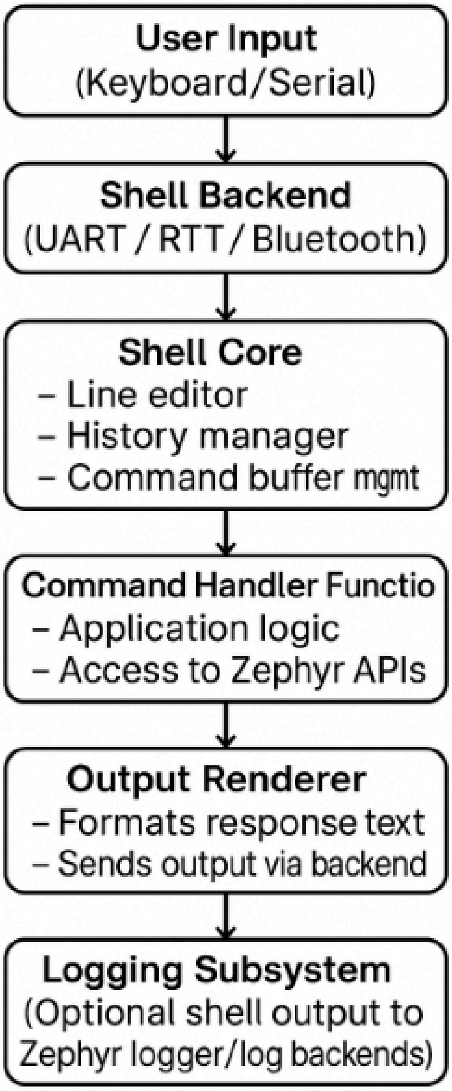
Zephyr Shell supports multiple backends, enabling diverse interaction methods:

- **UART:** Traditional serial communication.
- **RTT (Real-Time Transfer):** For debugging via SWD without using UART.
- **Bluetooth:** Allows wireless shell access over BLE .

Each backend is initialized and managed through specific APIs, and multiple backends can be active simultaneously, each with its own shell instance.

🧰 Advanced Features

- **Command Completion:** Pressing Tab auto-completes commands or lists possible options.
- **Command History:** Navigate through previous commands using arrow keys.
- **Built-in Commands:** Includes commands like help, clear, and history.
- **Conditional Command Registration:** Commands can be included or excluded based on configuration flags .



- **Official Shell Documentation:** Zephyr Project Documentation
- **Shell API Reference:** Zephyr Project Documentation
- **Custom Shell Module Example:** Zephyr Project Documentation

Full Command Execution Cycle (Step-by-Step)

1. **User Input**
    - Command typed into a terminal connected over serial, RTT, or Bluetooth.
2. **Backend**
    - Receives raw character input, handles input buffering and echoing.
3. **Shell Core**
    - Manages user interaction:
        - Input editing (arrow keys, backspace)
        - Maintains command history
        - Triggers parsing on newline (Enter)
4. **Command Parser**
    - Breaks input into tokens (command + arguments)
    - Matches command from a static/dynamic table
    - Verifies argument count/format
5. **Command Handler**
    - Registered C function executes the logic
    - Can perform hardware control, data display, configuration, etc.

6. **Output Renderer**
    - Formats result text or logs
    - Sends it back through the shell backend
7. **Logging Subsystem (Optional)**
    - Shell can redirect output to Zephyr's log system
    - Enables log buffering, remote logging, and filtering

Zephyr Shell Subsystem Overview

The Zephyr Shell provides a command-line interface for interacting with the system, facilitating tasks like debugging, configuration, and monitoring. Its architecture comprises several key components:

1. **Shell Core**: Manages command parsing, execution, and user interaction.
2. **Backends**: Interfaces for input/output operations (e.g., UART, RTT, Bluetooth).
3. **Command Handlers**: Functions registered to handle specific shell commands.
4. **Logging Integration**: Allows shell output to be directed to Zephyr's logging system.

🔄 Detailed Component Flow and Source Mapping

1. **Shell Core**

implements the **core logic of Zephyr's shell subsystem**. This core is responsible for managing user interactions, parsing commands, executing registered command handlers, and coordinating input/output through backends (e.g., UART, RTT).

It connects backends, manages user input/output, interprets commands, and invokes the right handlers. While backends handle the communication, and command handlers define functionality, shell.c ties everything together into a cohesive interactive CLI framework.

Responsibilities:

- Initializes and manages shell instances.
- Handles user input, command parsing, and execution.
- Manages features like command history and auto-completion.

Key Source Files and Functions:

- **File**: subsys/shell/shell.c
- **Functions**:
    - shell_init(): Initializes the shell with specified backend and configuration.
    - shell_start(): Starts the shell instance, enabling it to process input.
    - shell_process(): Processes input characters and executes commands.
    - shell_execute_cmd(): Executes a given command string.

Usage Example:

const struct shell *shell = shell_backend_uart_get_ptr();

shell_start(shell);

---

2. **Backends**

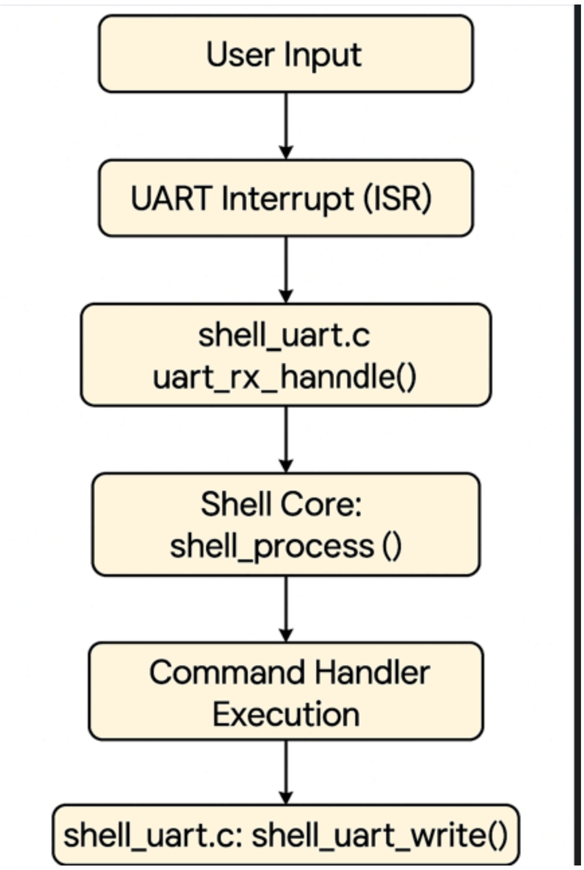
The Zephyr Shell supports multiple backends for I/O operations, with **UART** and **RTT** being the most widely used. Facilitate input/output operations between the user and the shell.

1. UART Backend

Responsibilities:

- **Hardware Initialization**: Configures UART pins, baud rate, and interrupts .
- **Input/Output Buffering**: Manages FIFO buffers for asynchronous data transfer.
- **Thread Management**: Spawns dedicated RX/TX threads for non-blocking I/O .
- **Protocol Handling**: Translates shell commands into UART frames (e.g., ANSI escape codes)

File	Location	Description
shell_uart.c	subsys/shell/backends/shell_uart.c	**Implements UART backend logic (initialization, I/O threads).**
shell_uart.h	include/zephyr/shell/shell_uart.h	**Defines UART-specific macros and APIs (e.g., shell_backend_uart_get_ptr()).**



Function	**Purpose**
uart_rx_handle()	**Handles incoming UART characters (invoked by the RX interrupt).**
shell_uart_write()	**Writes shell output over UART using the Zephyr UART driver.**
shell_uart_enable()	**Called when the shell is started to configure UART and enable RX.**
shell_uart_init()	**Initializes buffers and transport structures.**
SHELL_BACKEND_DEFINE(shell_uart_backend, ...)	**Registers the UART backend with the shell core.**

Core Responsibilities

- **Initialization of UART Shell Backend**
    - Initializes UART hardware for shell usage.
    - Sets up ring buffers for handling UART RX and TX.
    - Registers the UART backend with the shell core system.
- **Handling UART Input (RX)**
    - Receives characters via UART interrupts or polling.
    - Buffers incoming characters in a ring buffer.
    - Passes complete lines or commands to the shell parser for execution.
- **Handling UART Output (TX)**
    - Sends shell output (e.g., command results, prompts) to UART.
    - Uses a transmit buffer for efficient output streaming.
    - Supports synchronous or asynchronous transmission depending on UART driver.
- **UART Callback Mechanisms**
    - Uses interrupt-driven mechanisms to detect RX-ready events.
    - May handle TX-ready interrupts for efficient non-blocking output.
- **Shell Backend API Implementation**
    - Implements functions required by the shell_backend_api (like enable, write, update, uninit).
    - Enables the shell core to interact with UART as a communication channel.
- **Support for Multithreading**
    - Uses atomic operations and proper synchronization to handle concurrent access to buffers in ISR and main context.

Usage Example:

```
const struct shell *shell = shell_backend_uart_get_ptr();

shell_start(shell);
```

---

3. **Command Handlers**

Responsibilities:

- Define the functionality executed when a specific shell command is invoked.

Command handlers define the functionality executed when a shell command is invoked in Zephyr RTOS. They bridge user input (via the shell) to application logic.

Key Source Files and Macros:

- **Header**: include/zephyr/shell/shell.h
- **Macros**:
    - SHELL_CMD_REGISTER(): Registers a static shell command.
    - SHELL_CMD_ARG_REGISTER(): Registers a shell command with arguments.

Usage Example:

Custom Command

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

// Handler for "adc read <channel>"

static int cmd_adc_read(const struct shell *sh, size_t argc, char **argv) {

  uint8_t channel = (uint8_t)strtol(argv[1], NULL, 10);

  const struct device *adc = device_get_binding("ADC_0");

  int32_t val;

  adc_read(adc, channel, &val);  // Zephyr ADC API

  shell_print(sh, "ADC%d: %d mV", channel, val);

  return 0;

}

// Register command

SHELL_CMD_REGISTER(adc, NULL, "Read ADC channel", cmd_adc_read);
```

Usage:

```
uart:~$ adc read 3

ADC3: 3245 mV
```

---

4. **Logging Integration**

Redirect Shell Output to Logging System
 Sends logs and shell messages to the Zephyr Logging Subsystem (which can output to UART, RTT, memory, or custom backends).

Zephyr Logging System (log subsystem) with the **Shell subsystem**. It acts as a **log backend**, which means it enables **log messages** (like LOG_INF, LOG_ERR, etc.) to be **displayed via the active shell backend** (e.g., UART or RTT).

Log Level Control
 Allows filtering logs (e.g., only show INFO and above) and dynamic runtime level adjustment.

Formatted Output
Ensures log messages follow structured formatting defined by the shell or developer.

Responsibilities:

- Redirect shell outputs to Zephyr's logging system.
- Manage log levels and formatting.

Key Source Files and Functions:

- **File**: subsys/shell/shell_log_backend.c
- **Functions**:
    - shell_log_backend_enable(): Enables the shell log backend.
    - shell_log_backend_disable(): Disables the shell log backend.

Function	Purpose
shell_log_backend_enable()	Enables the shell log backend and connects it to the shell instance.
shell_log_backend_disable()	Disables the shell log backend and disconnects from the shell.
shell_log_backend_output_func()	Core function that sends formatted logs to the shell backend.
shell_log_backend_set_filter()	Adjusts log filtering (e.g., log level, module).

Configuration Options:

Set these in your prj.conf file to enable and configure shell log backend:

Option	Description
CONFIG_SHELL_LOG_BACKEND=y	**Enables the shell as a log backend.**
CONFIG_LOG_BACKEND_UART=y	**Enables UART as another log backend (independent of shell).**
CONFIG_LOG	**Enables Zephyr's logging subsystem.**
CONFIG_LOG_MODE_IMMEDIATE=n	**Ensures logs are processed through backend (not printed directly).**
CONFIG_LOG_PROCESS_THREAD=y	**Needed for logs to be handled in a separate thread (recommended for shell).**

```
#include <zephyr/logging/log.h>
#include <zephyr/kernel.h>
LOG_MODULE_REGISTER(main, LOG_LEVEL_INF);
void main(void)
{
   LOG_INF("System initialized");
   LOG_WRN("This is a warning!");
   LOG_ERR("This is an error!");
}
```

CONFIG_SHELL_LOG_BACKEND=y, these messages will appear in the shell output window (e.g., UART terminal), formatted according to log settings

---

List all the zephry shell commands

Zephyr's shell provides a variety of built-in commands to interact with the system. Here are some commonly used ones:

- **clear** – Clears the screen.

- **shell** – Provides useful shell-related commands.
- **colors** – Adjusts shell color settings.
- **echo** – Prints text to the shell.
- **history** – Displays command history.
- **resize** – Adjusts terminal screen size.
- **device** – Commands related to device management.
- **devmem** – Read/write physical memory.
- **i2c** – Interact with the I2C bus.
- **kernel** – Commands related to kernel operations.
- **led_ctrl** – Change LED states.
- **threads** – List kernel threads.
- **uptime** – Display system uptime.
- **version** – Show kernel version.

Where Zephyr does NOT use RPC

- **Shell subsystem**: It does not use RPC; command handlers are function calls local to the device.
- **Logging, Device Drivers, Networking APIs**: These are direct API interfaces, not using RPC.
- **Bluetooth, CAN, UART, I2C**: These use standard HAL/drivers—not RPC.

Shell Subsystem (No RPC)

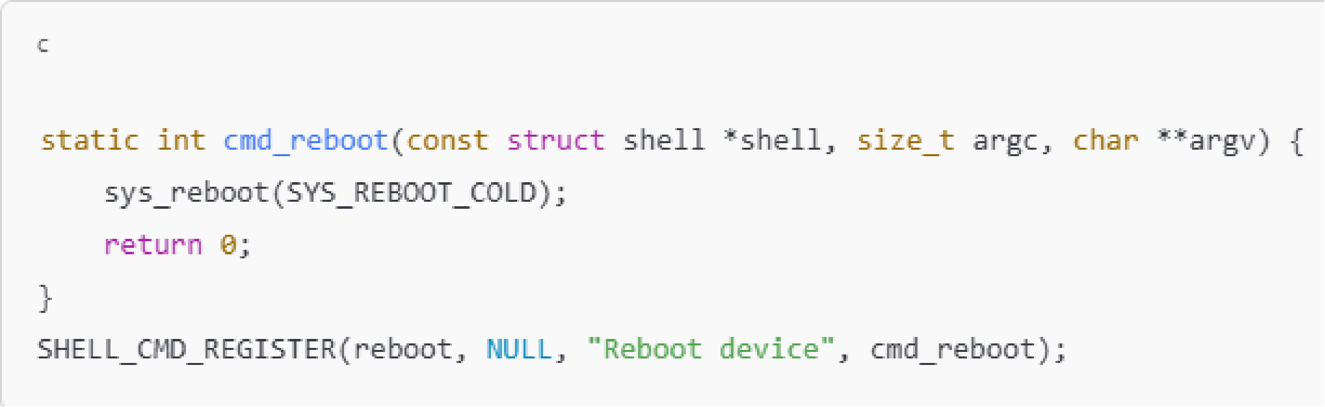
🔧 How it works:

- When you type a shell command (e.g., reboot), Zephyr:
    1. Parses the command using shell.c
    2. Looks up the handler function in its **shell command tree** (registered via SHELL_CMD_REGISTER)
    3. Calls the corresponding function **directly** in the **same thread/context**

Mechanism:

- **Direct function pointers**
- **No serialization, no remote calls**
- Commands are executed as normal C functions

🧠 Example:



```c
c

static int cmd_reboot(const struct shell *shell, size_t argc, char **argv) {
    sys_reboot(SYS_REBOOT_COLD);
    return 0;
}
SHELL_CMD_REGISTER(reboot, NULL, "Reboot device", cmd_reboot);
```

SHELL_CMD_REGISTER(reboot, NULL, "Reboot device", cmd_reboot);

Zephyr RTOS scheduler

The Zephyr RTOS scheduler is a key component of the Zephyr kernel responsible for managing task execution across threads, ensuring real-time responsiveness and efficiency. Here's a detailed breakdown of how Zephyr's scheduler works:

🧠 1. Scheduling Model

Zephyr supports preemptive, cooperative, and timeslicing scheduling:

Mode	**Description**
Preemptive	**Higher-priority threads can preempt lower-priority ones at any time.**
Cooperative	**Threads must yield control voluntarily; no preemption by scheduler.**
Timeslicing	**Scheduler enforces time limits on same-priority threads in round-robin way.**

⚙️ 2. Thread States in Zephyr

Each thread (or task) can be in one of these states:

- READY: Can run immediately when scheduled.
- RUNNING: Currently executing.
- PENDING: Waiting for an event or resource.
- SUSPENDED: Inactive, must be resumed explicitly.
- DEAD: Terminated and will be cleaned up.

🏗️ 3. Scheduler Architecture

Core Components:

Component	Description
k_thread	Structure representing a thread (in include/zephyr/kernel_structs.h).
z_ready_q	The ready queue, ordered by priority.
z_sched	Main scheduler function (in kernel/sched.c).
z_reschedule()	Called to determine if the current thread should be preempted.
z_swap()	Context switch between threads.
k_yield()	Voluntary yield by cooperative thread.
k_sleep()	Puts the thread into sleeping state for a duration.

🔄 4. Thread Priority & Timeslicing

- Priorities:
  - Zephyr allows 0 (highest) to CONFIG_NUM_PREEMPT_PRIORITIES - 1 for preemptive threads.
  - Priorities >= CONFIG_NUM_PREEMPT_PRIORITIES are cooperative.
- Timeslicing:
  - Enabled by CONFIG_TIMESLICING.
  - Same-priority threads share CPU in a round-robin fashion.
  - Controlled via CONFIG_TIMESLICE_SIZE (e.g., 10 ms).

🧵 5. Multithreading and SMP (Symmetric Multiprocessing)

- Zephyr supports SMP on multi-core platforms (e.g., x86_64, Arm Cortex-A).
- A separate scheduler instance runs per CPU.
- Threads can be pinned to specific CPUs using thread attributes.

📦 6. Key Kconfig Options

Option	Purpose
CONFIG_TIMESLICING	Enables round-robin scheduling.
CONFIG_NUM_PREEMPT_PRIORITIES	Number of preemptible priorities.
CONFIG_SMP	Enables multi-core scheduling.
CONFIG_PRIORITY_CEILING	Enables priority ceiling protocol (for mutexes).

🔧 7. Context Switching

Performed via:

- PendSV (on ARM) or Software Interrupts (on x86).
- z_swap() saves context of current thread and restores context of the next thread.

🔍 8. Source Code References

Function	Location	Purpose
z_ready_thread()	kernel/sched.c	Moves thread to ready queue.
z_reschedule()	kernel/sched.c	Decides if a thread switch is needed.
z_swap()	kernel/include/kernel_internal.h	Triggers context switch.
z_thread_priority_set()	kernel/sched.c	Changes thread priority.

🧪 Example: Creating Threads with Different Priorities

```
void my_thread(void *a, void *b, void *c) {
   while (1) {
      printk("Running high-priority thread\n");
      k_sleep(K_MSEC(500));
   }
}


K_THREAD_DEFINE(my_tid, 1024, my_thread, NULL, NULL, NULL, 1, 0, 0);
// Priority 1: Higher than default (priority 0)
```

📌 Summary

- Zephyr uses a priority-based preemptive scheduler.
- Supports cooperative, preemptive, and timesliced scheduling.
- Uses ready queue to track schedulable threads.
- Can run on single or multi-core (SMP) systems.
- Optimized for real-time and embedded applications.

Zephyr Scheduling: Overview

The Zephyr kernel scheduler is responsible for selecting which thread runs at any given time. It supports preemptive, cooperative, and optional time-sliced scheduling with deterministic behavior—ideal for real-time systems.

🔄 Scheduling Types in Zephyr

1. ✅ Preemptive Scheduling

- Default behavior for threads with priority < CONFIG_NUM_PREEMPT_PRIORITIES.
- A higher-priority thread can interrupt a running lower-priority one.

- Enables responsiveness for critical tasks.

2. 🤝 Cooperative Scheduling

- Threads with priority ≥ CONFIG_NUM_PREEMPT_PRIORITIES.
- These do not preempt each other—they must explicitly yield (via k_yield()).
- Useful for reducing overhead in non-critical threads.

3. ⏱ Time Slicing (Optional)

- Allows equal-priority threads to share CPU time in a round-robin manner.
- Controlled by:
  - CONFIG_TIMESLICING (enable/disable)
  - CONFIG_TIMESLICE_SIZE (in ticks)
- Only applies to threads with the same priority and preemptible.

🧩 Thread Priorities

- Zephyr supports multiple priority levels.
- Priority values: Lower number = higher priority.
- Priority ranges:
  - Preemptive: 0 to CONFIG_NUM_PREEMPT_PRIORITIES - 1
  - Cooperative: CONFIG_NUM_PREEMPT_PRIORITIES to CONFIG_NUM_COOP_PRIORITIES - 1

🔧 Scheduler Configuration (Kconfig Options)

Option	Description
CONFIG_NUM_PREEMPT_PRIORITIES	Max preemptive priorities (default: 15)
CONFIG_NUM_COOP_PRIORITIES	Max cooperative priorities (default: 16)
CONFIG_TIMESLICING	Enables round-robin for same-priority threads
CONFIG_TIMESLICE_SIZE	Time slice duration (ticks)

🧵 Thread States and Transitions

State	Description
READY	Eligible to run
RUNNING	Currently executing
PENDING	Waiting for an event
SUSPENDED	Disabled by API call
DEAD	Terminated (not runnable)

State transitions are managed by:

- Blocking calls (e.g., k_sleep(), k_sem_take())
- API calls like k_thread_suspend() or k_thread_abort()

🔄 Core Scheduler APIs

API	Function
k_yield()	Yield CPU voluntarily
k_sleep()	Sleep for time period
k_thread_create()	Create a thread dynamically
k_thread_priority_set()	Change thread priority
k_thread_suspend()	Suspend a thread
k_thread_resume()	Resume a suspended thread

🧠 SMP Considerations (Symmetric Multiprocessing)

- Zephyr supports multi-core scheduling via CONFIG_SMP.
- Threads can migrate between cores if not pinned.
- Each CPU has a scheduler instance, but work is coordinated globally.

🛠️ Implementation Details

File	Purpose
kernel/sched.c	Main scheduling logic
kernel/thread.c	Thread creation/management
include/zephyr/kernel.h	Public APIs

🔍 Source Documentation

- Scheduler Overview
- Thread API
- Kconfig Reference

✅ Summary

- Zephyr uses priority-based preemptive scheduling with optional time-slicing.
- Cooperative threads allow manual yield for reduced overhead.
- The system is configurable and supports both single-core and multi-core (SMP) platforms.
- Designed for deterministic behavior in real-time systems.

Let me know if you'd like a flowchart, thread scheduling diagram, or real-code demo!

Zephyr Scheduling: Overview

The Zephyr kernel scheduler is responsible for selecting which thread runs at any given time. It supports preemptive, cooperative, and optional time-sliced scheduling with deterministic behavior—ideal for real-time systems.

🔄 Scheduling Types in Zephyr

1. ✅ Preemptive Scheduling

- Default behavior for threads with priority < CONFIG_NUM_PREEMPT_PRIORITIES.
- A higher-priority thread can interrupt a running lower-priority one.
- Enables responsiveness for critical tasks.

2. 🤝 Cooperative Scheduling

- Threads with priority ≥ CONFIG_NUM_PREEMPT_PRIORITIES.
- These do not preempt each other—they must explicitly yield (via k_yield()).
- Useful for reducing overhead in non-critical threads.

3. ⏱️ Time Slicing (Optional)

- Allows equal-priority threads to share CPU time in a round-robin manner.
- Controlled by:
  - CONFIG_TIMESLICING (enable/disable)
  - CONFIG_TIMESLICE_SIZE (in ticks)
- Only applies to threads with the same priority and preemptible.

🧩 Thread Priorities

- Zephyr supports multiple priority levels.
- Priority values: Lower number = higher priority.
- Priority ranges:
  - Preemptive: 0 to CONFIG_NUM_PREEMPT_PRIORITIES - 1
  - Cooperative: CONFIG_NUM_PREEMPT_PRIORITIES to CONFIG_NUM_COOP_PRIORITIES - 1

🔧 Scheduler Configuration (Kconfig Options)

Option	Description
CONFIG_NUM_PREEMPT_PRIORITIES	Max preemptive priorities (default: 15)
CONFIG_NUM_COOP_PRIORITIES	Max cooperative priorities (default: 16)
CONFIG_TIMESLICING	Enables round-robin for same-priority threads
CONFIG_TIMESLICE_SIZE	Time slice duration (ticks)

🧵 Thread States and Transitions

State	Description
READY	Eligible to run
RUNNING	Currently executing
PENDING	Waiting for an event
SUSPENDED	Disabled by API call
DEAD	Terminated (not runnable)

State transitions are managed by:

- Blocking calls (e.g., k_sleep(), k_sem_take())
- API calls like k_thread_suspend() or k_thread_abort()

🔄 Core Scheduler APIs

API	Function
k_yield()	Yield CPU voluntarily
k_sleep()	Sleep for time period
k_thread_create()	Create a thread dynamically
k_thread_priority_set()	Change thread priority
k_thread_suspend()	Suspend a thread
k_thread_resume()	Resume a suspended thread

🧠 SMP Considerations (Symmetric Multiprocessing)

- Zephyr supports multi-core scheduling via CONFIG_SMP.
- Threads can migrate between cores if not pinned.
- Each CPU has a scheduler instance, but work is coordinated globally.

🛠️ Implementation Details

File	Purpose
kernel/sched.c	Main scheduling logic
kernel/thread.c	Thread creation/management
include/zephyr/kernel.h	Public APIs

🔍 Source Documentation

- Scheduler Overview
- Thread API
- Kconfig Reference

✅ Summary

- Zephyr uses priority-based preemptive scheduling with optional time-slicing.
- Cooperative threads allow manual yield for reduced overhead.
- The system is configurable and supports both single-core and multi-core (SMP) platforms.
- Designed for deterministic behavior in real-time systems.

Let me know if you'd like a flowchart, thread scheduling diagram, or real-code demo!

2/2

4o

You said:

expalain this code in details https://github.com/zephyrproject-rtos/zephyr/blob/main/kernel/sched.c

ChatGPT said:

You can view the full source code for sched.c here: kernel/sched.c on GitHub

Here’s a detailed explanation of what this file does:

🔧 Purpose of sched.c

This file implements Zephyr’s core scheduling logic. The scheduler is responsible for:

- Managing thread states (ready, running, pending, etc.)
- Selecting the highest-priority thread to run
- Managing preemption, cooperative behavior, and optional time-slicing
- Supporting single-core and symmetric multi-processing (SMP)

🧩 Major Components and Functions

1. Thread Selection

c

CopyEdit

```
struct k_thread *_sched_next_thread(void);
```

- Chooses the next thread to run based on priority and ready state.
- This is the core scheduler decision function.
- It considers cooperative and preemptive priorities.

2. Thread Queues and Ready Queue

c

CopyEdit

```
struct _ready_q {

   ...

};
```

- Maintains threads ready to run, sorted by priority.
- Implemented using a priority bitmap for fast lookup.
- Threads are added/removed when they become ready/unready.

3. Adding/Removing Threads from Scheduler

c

CopyEdit

```
void z_ready_thread(struct k_thread *thread);

void z_remove_thread_from_ready_q(struct k_thread *thread);
```

- z_ready_thread: Places a thread into the ready queue.
- z_remove_thread_from_ready_q: Removes a thread from the scheduler.
- These are used during blocking/unblocking, yielding, and thread lifecycle changes.

4. Time Slicing Logic

c

CopyEdit

```
void z_reset_time_slice(void);
```

- Supports round-robin scheduling when threads have the same priority.
- This function resets the time slice counter when needed.
- Controlled by CONFIG_TIMESLICING.

5. Scheduler Locking (Spinlocks)

c

CopyEdit

```
struct k_spinlock sched_spinlock;
```

- Used to ensure thread-safe operations in multi-core (SMP) scenarios.
- Prevents race conditions while modifying scheduler structures.

6. Context Switching

c

CopyEdit

```
void z_reschedule(struct k_spinlock *lock);
```

- Called when the current thread should be preempted or voluntarily yields.
- Performs context switch to the next thread.
- Ensures spinlock is properly released after rescheduling.

7. Idle Thread Management

c

CopyEdit

```
void z_idle(void);
```

- Called when no threads are ready to run.
- Puts CPU into a low-power idle state until an interrupt occurs.

8. Thread Scheduling Hooks

Zephyr uses weak functions or macros like:

c

CopyEdit

__weak void z_sched_lock(void);

__weak void z_sched_unlock(void);

- Used to hook into scheduler events.
- Can be overridden in architectures or extensions for platform-specific logic.

📁 Related Files

- [kernel/thread.c](): Handles thread creation, priority management.
- [include/zephyr/kernel_structs.h](): Defines thread structures.
- [include/zephyr/kernel.h](): Exposes public APIs for applications.

⚙️ Configurations That Affect This File

- CONFIG_TIMESLICING
- CONFIG_SMP
- CONFIG_NUM_PREEMPT_PRIORITIES
- CONFIG_NUM_COOP_PRIORITIES
- CONFIG_THREAD_RUNTIME_STATS

✅ Summary

kernel/sched.c is the heart of the Zephyr RTOS scheduler, implementing:

- Priority-based thread selection
- Cooperative and preemptive behavior
- Time slicing
- Scheduler locking for SMP
- Efficient ready queue handling
- Thread state transitions

If you want a visual diagram of thread scheduling, context switching, or time slicing logic, I can generate one. Let me know!

You can find the complete list of Zephyr's scheduler APIs in the official documentation here:
🔗 [Zephyr Scheduler API Reference]()

Here’s a consolidated list of key scheduler-related APIs in Zephyr:

🧵 Thread Scheduling & Control APIs

API	Description
k_yield()	Causes the current thread to yield execution to another of the same priority.
k_sched_lock()	Locks the scheduler to prevent context switching (cooperative only).
k_sched_unlock()	Unlocks the scheduler, re-enabling context switching.
k_thread_priority_set()	Sets the priority of a thread.
k_thread_priority_get()	Gets the current priority of a thread.
k_sleep()	Puts the current thread to sleep for a specified duration.
k_usleep()	Sleep for microseconds (when supported).
k_busy_wait()	Busy-wait for a specified time without yielding the CPU.

⏱️ Time Slicing

Config / Macro	Purpose
CONFIG_TIMESLICING	Enables time slicing across threads of the same priority.
CONFIG_TIMESLICE_SIZE	Defines the time slice size (in milliseconds).
CONFIG_TIMESLICE_PRIORITY	Minimum priority to apply time slicing.

🧰 Internal / Advanced APIs (kernel scope)

These are used internally but may be relevant for advanced development:

Function	Description
z_reschedule()	Triggers a scheduler decision.
z_ready_thread()	Marks a thread as ready to run.
z_remove_thread_from_ready_q()	Removes a thread from the ready queue.
z_reset_time_slice()	Resets the time slice counter.
_sched_next_thread()	Returns the next thread to run.

💡 Additional Concepts

- Scheduler domains in SMP (multi-core scheduling)
- Idle thread invoked when no other thread is ready
- Thread states: running, ready, suspended, pending