# RTOSes for IoT

# Agenda

What we will cover in this lecture

- IoT constraints and what an RTOS must provide
- The IoT OS landscape (FreeRTOS, RIOT, NuttX, Contiki-NG)
- Zephyr architecture: kernel + OS services + connectivity stacks
- Development workflow: west, CMake, Kconfig, Devicetree
- Security, firmware updates, and power management
- Two short demos: Blinky + networking (Thread/IPv6/CoAP concepts)

# Part 1

What IoT demands from an RTOS

# Typical IoT constraints

Why “tiny + connected + safe” is hard

## Constrained resources

- RAM/flash are limited
- CPU budgets vary (MCUs → SoCs)
- Devices often need long battery life

## Connectivity is non-optional

- IPv6/UDP/TCP + security (TLS/DTLS)
- IoT app protocols (CoAP, MQTT, LwM2M)
- Multiple link layers (802.15.4, BLE, Wi-Fi, CAN)

## Modern expectations

- Secure boot + signed updates
- Isolation (where hardware supports it)
- Testability, CI, reproducible builds
- Observability (logging/tracing/metrics)
- Fast bring-up across many boards

# IoT protocol stacks (quick refresher)

From radios to application protocols

Key idea: enable only what you need to reduce RAM/flash usage.

| | |
|---|---|
| **Application** | CoAP • MQTT • HTTP • LwM2M |
| **Security** | DTLS • TLS |
| **Transport** | UDP • TCP |
| **Network** | IPv6 • RPL |
| **Adaptation** | 6LoWPAN |
| **Link/PHY** | 802.15.4 • BLE • Wi-Fi • CAN |

# Kernel & RTOS design patterns

A few concepts that matter on MCUs

## Scheduling models

- Priority-based scheduling
- Preemptive vs cooperative threads
- Optional time slicing for equal priorities
- Specialized policies (e.g., EDF)

## Memory & safety

- Static vs dynamic allocation
- Stack sizing + overflow detection
- Optional user mode (MPU/MMU backed)
- Fault handling & assertions

## Power-aware idling

- Tickless idle (event-driven)
- System + device power management
- Minimize wake-ups, wake latency tradeoffs

## Portability levers

- Hardware description (Devicetree)
- Feature selection (Kconfig)
- HAL/driver layers
- Consistent tooling (build/flash/debug)

# Part 2

RTOS landscape

# A quick landscape view

Different trade-offs for different products

## FreeRTOS

Great when you want a small kernel + choose your own ecosystem.
Often paired with vendor SDKs and separate stacks.

## Zephyr

“Batteries included”: kernel + drivers + networking + security services.
Highly configurable via Kconfig + Devicetree.

## RIOT

IoT-friendly networking (6LoWPAN/IPv6/RPL/CoAP, etc).
Strong community and modular design.

## Apache NuttX

POSIX-leaning RTOS with standards compliance focus.
Scales from tiny MCUs to larger systems.

## Contiki-NG

Research-friendly OS with low-power IPv6 stacks (RPL/CoAP/LwM2M).
Strong for constrained network experiments.

## Rule of thumb

Choose the smallest thing that still meets your connectivity + security + maintainability targets.

# Zephyr snapshot

What the project looks like today

## Current release: 4.3 (GA)

Next releases move to a bi-yearly cadence in 2026.

Permissive Apache-2.0 licensing.

### Why teams pick Zephyr for IoT products

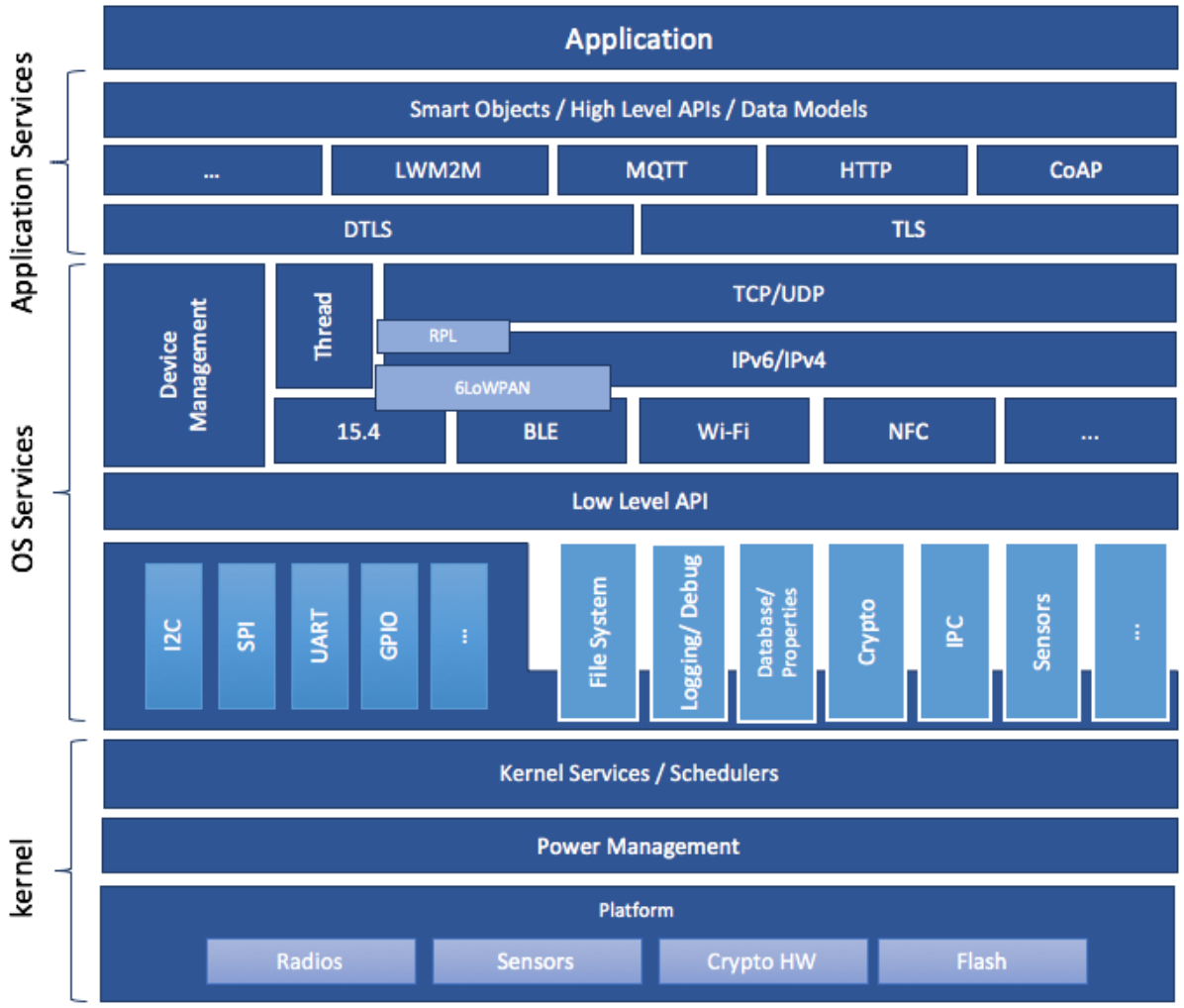
- Built-in connectivity stacks (IPv4/IPv6, 6LoWPAN, etc.)
- Devicetree + Kconfig for portability and feature selection
- west tooling: multi-repo, build/flash/debug workflows
- Security building blocks: TF-M integration + firmware upgrade subsystems
- Strong upstream documentation + active release process

# Part 3

Zephyr deep dive

# Zephyr system architecture

Kernel + OS services + connectivity stacks




## Kernel services

Threads, scheduling, interrupts, synchronization primitives,
timers, memory domains (optional user mode).

## OS services

Drivers, file systems, logging, IPC, crypto,
device management + power management.

## Connectivity

Modular IP stack (IPv4/IPv6) + multiple L2 options.
App protocols like CoAP/MQTT can be enabled as needed.

# Kernel scheduling in Zephyr

Priority-based with multiple policies

## Thread types

- Cooperative threads (non-preemptible)
- Preemptible threads (priority-based preemption)
- Meta-IRQ / bottom-half style options
- Optional EDF scheduling

## Time slicing

Time slicing can be enabled for preemptible threads of equal priority, with a configurable slice size.

Use this when fairness matters more than strict single-thread latency.

## Design takeaway

In embedded systems, scheduling decisions tie directly to: latency, power (wakeups), and memory (stack sizes).
Zephyr lets you configure policies to match your product constraints.

# Portability: Devicetree vs Kconfig

Hardware description vs software feature selection

## Devicetree (hardware)

- Describes the board/SoC hardware graph
- Pins, buses, peripherals, compatibles
- Overlays tweak board defaults without editing upstream files

**Devicetree overlay example**

```
/* boards/<board>.overlay (Devicetree) */
&uart0 {
  current-speed = <115200>;
  status = "okay";
};

```

## Kconfig (software)

- Select features to compile into the image
- Drivers, protocols, logging levels, etc.
- Typically controlled through prj.conf and menuconfig-like interfaces

**Kconfig overlay example**

```
/* prj.conf (Kconfig symbols) */
CONFIG_GPIO=y
CONFIG_LOG=y
CONFIG_NET_IPV6=y

```

# A Zephyr “hello world” pattern

Devicetree-backed device handles

GPIO DT spec makes application code portable across boards:

blinky/src/main.c

```c
#include <zephyr/kernel.h>
#include <zephyr/drivers/gpio.h>

#define LED0_NODE DT_ALIAS(led0)
static const struct gpio_dt_spec led =
  GPIO_DT_SPEC_GET(LED0_NODE, gpios);

int main(void) {
  if (!gpio_is_ready_dt(&led)) return 0;
  gpio_pin_configure_dt(&led, GPIO_OUTPUT_ACTIVE);
  while (1) {
    gpio_pin_toggle_dt(&led);
    k_msleep(500);
  }
}

```

## Some key points

- LED alias comes from the board Devicetree
- gpio_dt_spec captures pin + controller
- gpio_is_ready_dt() catches missing drivers
- k_msleep() uses kernel timing services

# Tooling workflow

west + CMake provide a consistent developer experience

## west (meta-tool)

- Manages multiple repositories
- Provides build/flash/debug commands
- Supports custom extension commands

## CMake build system

- Two-stage configure + build
- Integrates app + Zephyr kernel
- Works across host platforms

**Command-line workflow**

```
# Typical session
west init -m
https://github.com/zephyrproject-
rtos/zephyr
west update
west zephyr-export

# Build + flash (example board)
west build -b nrf52840dk_nrf52840
samples/basic/blinky
west flash

```

Tip: Use “west build -t menuconfig” for interactive Kconfig changes.

# Networking in Zephyr

Modular IP stack with multiple L2 options

## Core idea

Zephyr’s IP stack is modular and can be configured at build time.
You can minimize memory usage by enabling only required features.

## Common IoT profiles

- 802.15.4 + 6LoWPAN + IPv6 + RPL
- Thread (IPv6 mesh over 802.15.4)
- BLE (incl. BLE Mesh in supported setups)
- Wi-Fi + TCP/UDP + TLS/DTLS
- CoAP/MQTT application protocols

## Thread example stack

- Application: CoAP
- Security: DTLS
- Transport: UDP
- Network: IPv6 + RPL
- Adaptation: 6LoWPAN
- Link: IEEE 802.15.4

Thread is an IPv6-based standard using 6LoWPAN over 802.15.4.

# Security & firmware updates

From root-of-trust to OTA

## Trusted Firmware-M (TF-M)

TF-M provides PSA Root of Trust services. Its secure boot image verifies secure and non-secure images and helps protect the firmware update process via public signing keys.

## DFU / OTA in Zephyr

The DFU subsystem provides frameworks to upgrade Zephyr-based
applications at runtime. OTA typically downloads a new image to an update slot, then upgrades using the MCUboot process.

## Key points

- “Secure boot” is a chain of trust
- Updates must be authenticated and rollback-safe
- Separate concerns: bootloader vs app vs transport
- Hardware features matter (TrustZone-M, MPU, flash layout)

Also: Zephyr is permissively licensed under Apache-2.0.

# Power management

Tickless idling + system/device PM

## Tickless idle (event-driven)

A tickless kernel can sleep until the next event instead of waking periodically for tick interrupts (“race to idle”).
This reduces unnecessary wake-ups and saves energy.

## System + device PM

Zephyr’s PM subsystem supports system power states and can suspend devices when the SoC goes to sleep (with care around ongoing transactions).

## Practical angle

- Power is a system property, not a driver flag
- Measure: current draw vs latency vs duty cycle
- Choose wake sources and budgets early
- Configuration: what’s enabled affects power (protocols, logging, timers, peripherals)

Zephyr PM docs: system PM + device PM.

# Testing & developer experience

Why “upstream” matters

## Twister (test harness)

Zephyr includes a large automated test suite. Recent releases have added tools to validate on-target behaviors such as display output via captured frames (“display harness”).

## Better diagnostics

Zephyr 4.3 highlights include Devicetree diagnostics (dtdoctor) and tools to inspect Kconfig provenance (traceconfig).

## Key development points

- Read build logs
- Show how to debug Devicetree errors
- Track RAM/ROM changes when enabling stacks
- Use reproducible builds (same commands, same results)

Takeaway: good tooling reduces “RTOS friction”.

# Part 4

Hands-on: two small demos

# Demo 1 — Blinky (portable)

Goal: blink the “led0” alias on any supported board

## Steps

1) Install toolchain + west (per Getting Started)
2) Initialize workspace (west init/update)
3) Build for your board
4) Flash and observe LED toggle

## Teaching points

- Board selected with “-b <board>”
- led0 alias comes from Devicetree
- prj.conf enables needed subsystems

Terminal

```
# Build + flash
west build -b <your_board>
samples/basic/blinky
west flash

# Optional: open config UI
west build -t menuconfig

```

# Demo 2 — Networking (conceptual path)

Goal: show how features are enabled and composed

## Option A: IP over Wi-Fi/Ethernet

Use Zephyr sockets (TCP/UDP) and a library protocol (CoAP/MQTT).
This is easiest to demo on dev boards with Wi-Fi/Ethernet.

## Option B: Thread mesh (802.15.4)

Thread is IPv6-based and uses 6LoWPAN over 802.15.4.
A Thread Border Router connects the mesh to the wider IP network.

**Enable only what you need**

```
# Build networking sample
west build -b <board> samples/net/sockets/echo_server

# Example config knobs
CONFIG_NET_IPV6=y
CONFIG_NET_SOCKETS=y
# + enable desired L2: Wi-Fi / Ethernet / 802.15.4
```

Lesson: networking is “lego bricks” — stacks are compiled in, not installed at runtime.

# Choosing the right OS

A pragmatic decision matrix

Ask these questions first:

- Do you need a full networking stack (IPv6/6LoWPAN/Thread/BLE/Wi-Fi)?
- Do you need secure boot + robust OTA?
- How many boards/SoCs must you support?
- Is POSIX compatibility important?
- How much of the ecosystem do you want “included” vs assembled?

## Typical picks

- Zephyr: “integrated” IoT stack + portability
- FreeRTOS: minimal kernel, bring your own stacks
- NuttX: stronger POSIX/Unix-like interfaces
- RIOT / Contiki-NG: networking-centric research/IoT stacks

# References & further reading

Primary docs worth bookmarking

- Zephyr docs: Introduction + Getting Started
- west: workspaces, build/flash/debug
- Devicetree vs Kconfig + GPIO peripheral guide
- Networking overview + Thread API docs
- TF-M integration + DFU/OTA docs
- Power management subsystem docs
- Zephyr 4.3 release highlights (tooling improvements)