Embedded and Real-Time Systems course

Zephyr Project Introduction

SOTR 2022/2023
Paulo Pedreiras
DETI/UA/IT

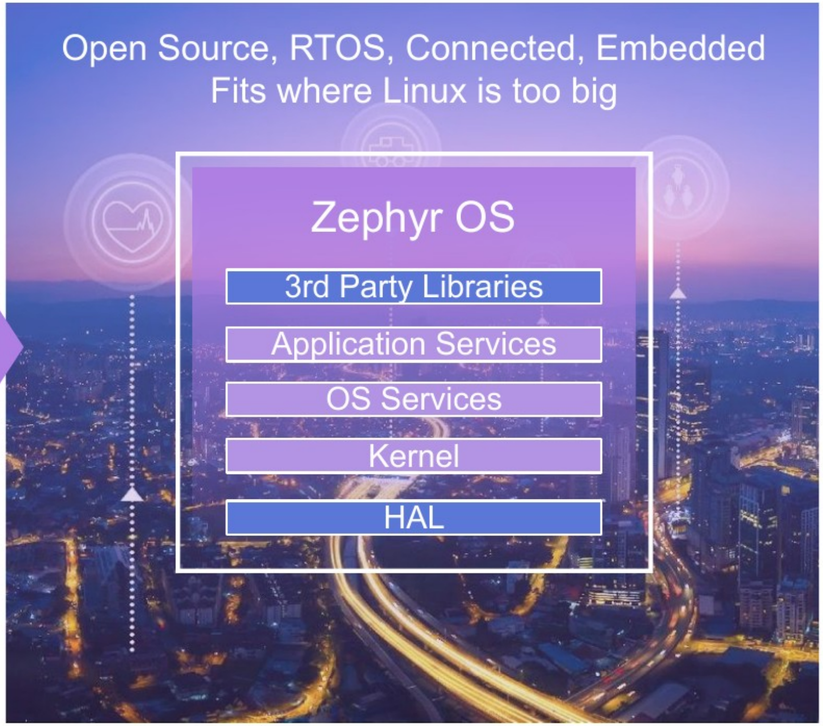
Vision

Quoted from the Linux Foundation documentation:

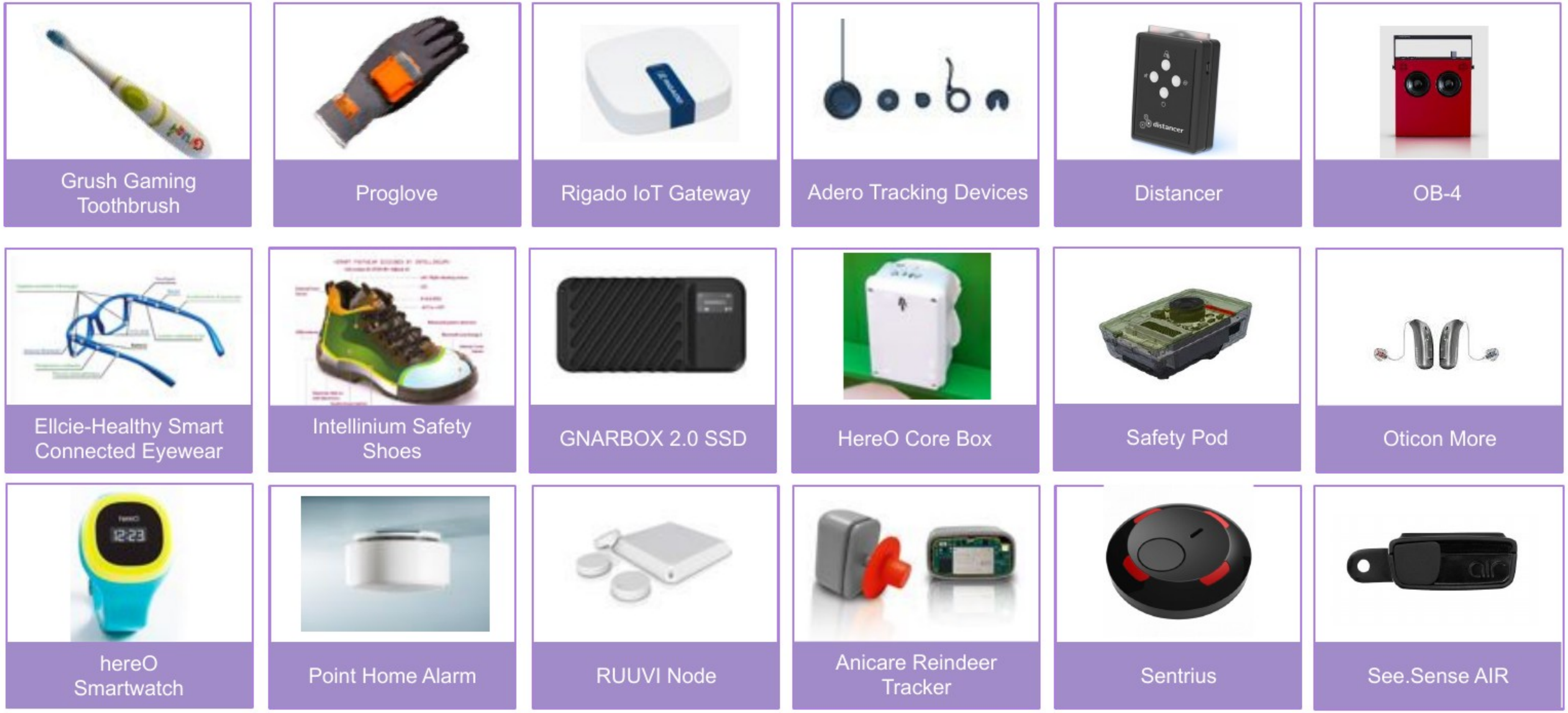
“The Zephyr Project strives to deliver the best-in-class RTOS for connected resource-constrained devices, built to be secure and safe.”

Zephyr Project Overview

- **Open source real time operating system**
- Vibrant **Community** participation
- Built with **safety** and **security** in mind
- **Cross-architecture** with broad SoC and development board support.
- **Vendor Neutral governance**
- **Permissively licensed** - Apache 2.0
- **Complete, fully integrated, highly configurable and modular (for flexibility)**
- Product development ready using **LTS** includes **security updates**
- **Auditable codebase,** for certification



Some products using Zephyr



Zephyr vs other RTOS

- There are many other RTOS
- Selecting the “best one” depends on the criteria used (i.e. the user, the application, the context, ...).
- Among the most popular “competitors”
  - FreeRTOS (one of my favorites)
    - Provides essentially scheduling services and IPC
    - Much lower abstraction level regarding peripherals and comm stacks when compared with Zephyr
    - Backed by Amazon ...
  - Mbed OS
    - Also aims at IoT, complete ecosystem, etc.,
    - But it is owned by ARM and supports only ARM Cortex-M hardware
  - Azure RTOS (ThreadX)
    - Also aims at IoT, complete ecosystem, etc.,
    - But backed/controlled by Microsoft

Supported Boards

- Wide variety of supported architectures and boards
  - x86
  - ARM/ARM64
    - Hundreds of boards from NXP, ST, Nordic, BBC, TI, Microchip, Renesas, Digilent, Adafruit, Arduino, …
  - MIPS
  - NIOS II
  - RISC-V
  - XTENSA
  - SPARC
- It can even run as a native Linux application

Cortex-M, Cortex-R & Cortex-A

X86 & x86_64

32 & 64 bit

Xtensa

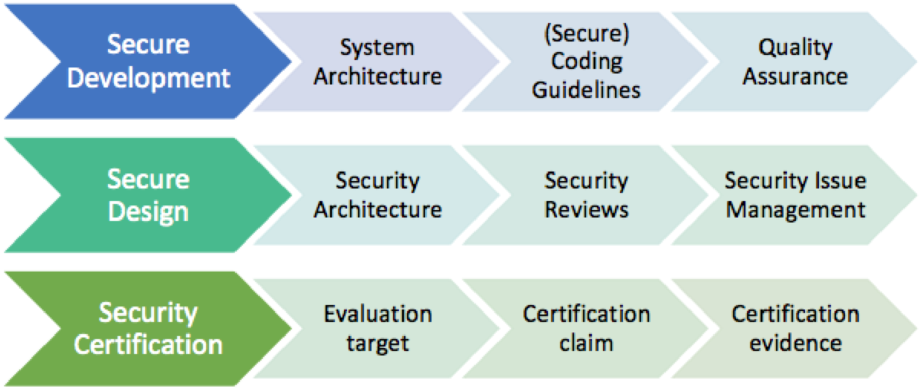
Build and configuration system

- The build and configuration system of Zephyr is unique in comparison to other RTOSs
  - The build system is Cmake, which is common in the (embedded) Linux world
  - The configuration system is Kconfig, which is common in the Linux world as well (it is the Kernel’s configuration system).

- Advantages
  - Based on solid, well established and widely disseminated tools.
    - Requires a significant effort to learn, but the knowledge acquired can be useful in many contexts and is long-lasting.
  - Embedded Linux developers will feel at home when working with the Zephyr RTOS, or, people that learn Zephyr will feel at home when moving to embedded Linux ...
  - Eases setup and maintenance of software products

Security

- Security is becoming a critical aspect of embedded and real-time applications
- The Zephyr project maintains a Security Overview which summarizes the measures taken to make the code base as secure as practically possible.



- There are Secure Coding Guidelines
  - https://docs.zephyrproject.org/latest/security/secure-coding.html

Safety

- Many embedded/real-time applications are subject to **safety requirements**
  - That is the case e.g. when a **malfunction** device can cause **significant economical impact or endanger human lives**
  - Common requirements in areas such as medical, industrial, automotive, ...
- Zephyr aims to be the first safety-certified free, open source RTOS

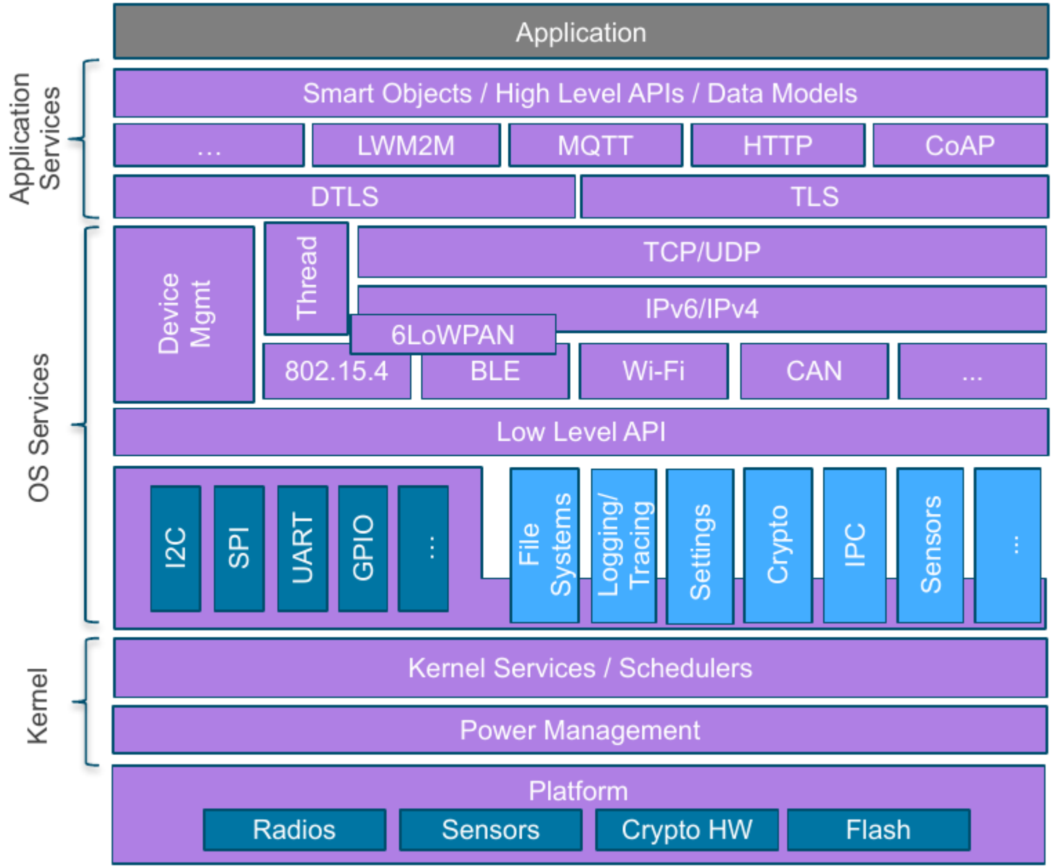
https://www.zephyrproject.org/zephyr-project-rtos-first-functional-safety-certification-submission-for-an-open-source-real-time-operating-system/

- There are certified RTOS, but they are not FOSS
  - E.g. FreeRTOS has a certified version SAFERTOS, which is not FOSS

Architecture

Main characteristics

- Highly Configurable and Modular
- Cooperative and Preemptive Threading
- Memory and Resources are typically statically allocated
- Integrated device driver interface
- Memory Protection: Stack overflow protection, Kernel object and device driver permission tracking, Thread isolation
- Bluetooth® Low Energy (BLE 5.1) with both controller and host, BLE Mesh
- 802.15.4 OpenThread
- Native, fully featured and optimized networking stack



Focusing on Kernel and scheduling

Kernel services for development:

- **Multi-threading Services** for cooperative, priority-based, non-preemptive, and preemptive threads with optional round robin time-slicing. Includes POSIX pthreads compatible API support.
- **Interrupt Services** for compile-time registration of interrupt handlers.
- **Memory Allocation Services** for dynamic allocation and freeing of fixed-size or variable-size memory blocks.
  - Note that for **Real-Time static memory allocation is preferred**
- **Inter-thread Synchronization Services** include binary semaphores, counting semaphores, and mutex semaphores.
- **Inter-thread Data Passing Services** include basic message queues, enhanced message queues, and byte streams.
- **Power Management Services** such as tickless idle and an advanced idling infrastructure.

Focusing on Kernel and scheduling

Zephyr provides a comprehensive set of thread scheduling choices:

- Cooperative and Preemptive Scheduling
- Sort of “Earliest Deadline First” (EDF)
  - When enabled, EDF is applied to jobs that have the same fixed priority. Works like a second-level scheduler.
- Meta IRQ scheduling implementing “interrupt bottom half” or “tasklet” behavior
- Timeslicing:
  - Enables time slicing between preemptible threads of equal priority
- Multiple ready-queue management strategies:
  - Simple linked-list ready queue: unsorted, small # of tasks
  - Red/black tree ready queue (self-balanced binary tree. Some memory and overhead. Use for 20+ tasks and scales well)
  - Traditional multi-queue ready queue (array of lists, one per priority)

Threads/Tasks

- **A short note in terminology**
  - The terms "**task**" and "**thread**" are often used **interchangeably**
  - Both refer to an object that has certain properties (e.g. periodicity, deadline, priority, state, stack) and executes some work (a C function) that consumes some time to complete
  - Usually tasks/threads share the same addressing space (unlike processes)
- **Threads/tasks are on the base of the real-time model**
  - A **real-time application** is structured as a **set of (potentially) cooperating tasks** that:
    - Carry out a specific processing (well defined inputs and/or outputs)
    - Are activated at appropriate instants (periodically, in response to an event)
    - Are scheduled for execution according to rules that assure that the requirements are met (e.g. response time, precedence, etc.)
      - E.g. when a user presses an emergency button a machine stops in no more than 100 ms

Threads/Tasks

- **In Zephyr any number of threads can be defined by an application**
  - Limit is the available RAM.
  - Each thread is referenced by a thread id that is assigned when the thread is created
- **Main properties of a thread**
  - A (private) stack area. Size should be adapted according to the space required by thread local variables
  - A thread control block for private kernel bookkeeping of the thread’s metadata (e.g. the current state)
  - An entry point (a C function), that corresponds to the work to be carried out
  - A scheduling priority
  - A set of thread options (depend on architecture, e.g. Floating Point Unit)
  - A start delay (how long the kernel should wait before starting the thread)
  - Execution mode (supervisor or user mode).
    - By default threads run in supervisor mode (access to privileged CPU instructions, the entire memory address space, and peripherals). User mode threads have a reduced set of privileges.
      Support to User Mode threads is optional (CONFIG_USERSPACE)

Threads/Tasks

- **Thread lifecycle and states**
  - A thread must be created before use
  - Threads typically execute forever, but they can also be terminated
    - A thread terminates if it returns from its C function
    - In such case it must release all owned shared resources (e.g. mutexes, dynamically allocated memory), as the the kernel does not reclaim them automatically.
    - A thread can be aborted if triggers a fatal condition (e.g. dereferencing a null pointer) or by an explicit call to k_thread_abort() (by itself or other thread)

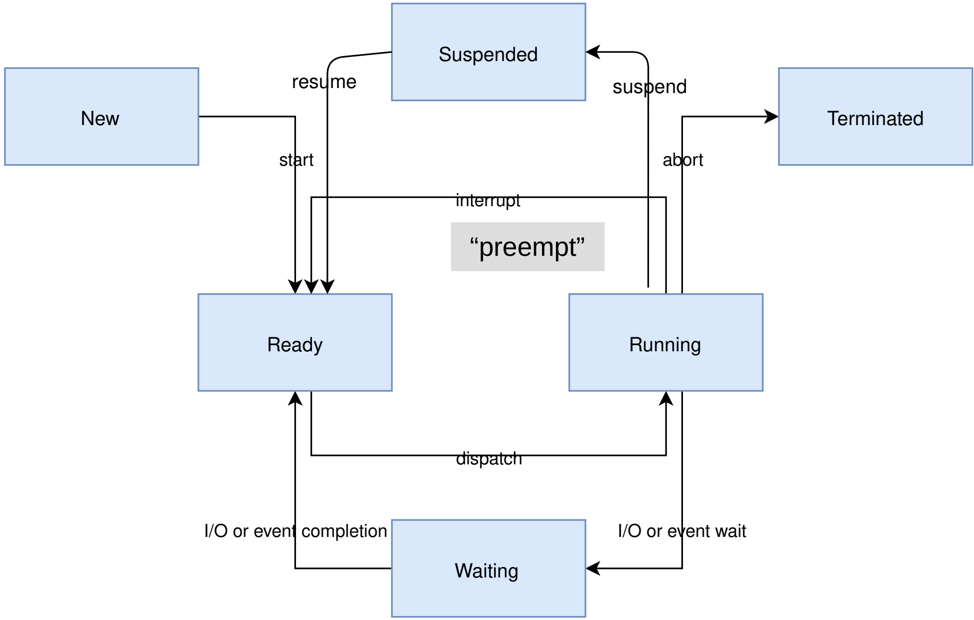


Image from: https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/zephyr/reference/kernel/threads/index.html

Threads/Tasks

- **Thread lifecycle and states**
  - A task is **Ready** if it is eligible for execution
    - The scheduler decides which ready tasks are granted with the CPU
    - Preemption can occur
  - A **Suspended** task is prevented from executing for an indefinite time.
    - Calls to k_thread_suspend() and k_thread_resume() mange the Suspended state
  - A task is on the **Waiting** state if is waiting for event, e.g. a semaphore that is unavailable or a timeout to occur

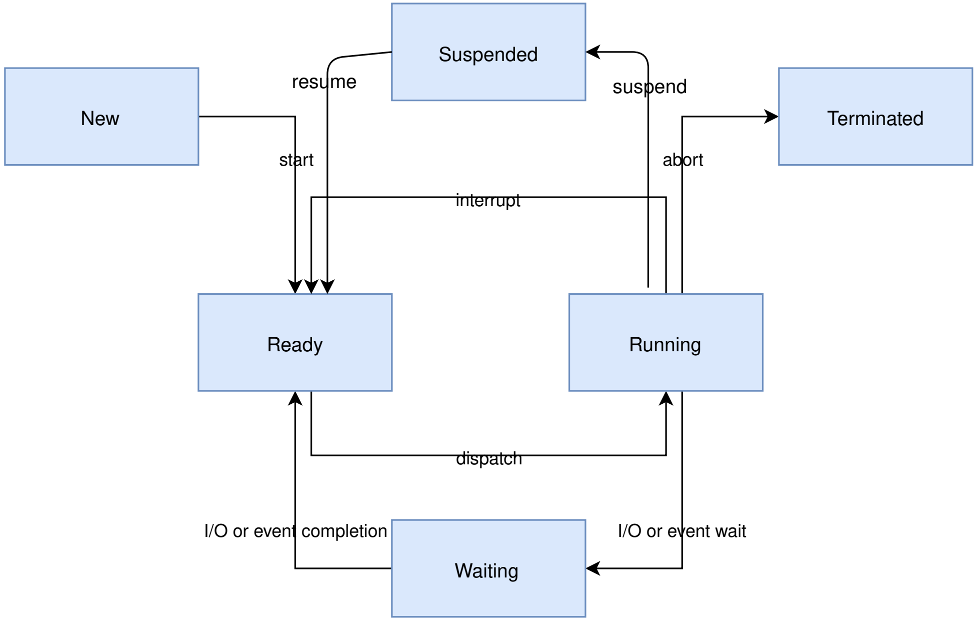


Image from:
https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/zephyr/reference/kernel/threads/index.html

Threads/Tasks

Thread Stack objects

- Each thread requires its own stack buffer for the CPU to push context. Depending on configuration, there are several constraints that must be met:
  - Additional memory may be reserved for memory management structures
  - If guard-based stack overflow detection is enabled, a small write-protected memory management region must immediately precede the stack buffer to catch overflows.
  - If userspace is enabled, a separate fixed-size privilege elevation stack must be reserved to serve as a private kernel stack for handling system calls.
  - If userspace is enabled, the thread's stack buffer must be appropriately sized and aligned such that a memory protection region may be programmed to exactly fit.
- Moreover alignment constraints can be quite restrictive, for example some MPUs require their regions to be of a power of two in size and word-aligned.
- Because of this, **portable code can't simply pass an arbitrary character buffer to k_thread_create()**. Special macros exist to instantiate stacks, prefixed with K_KERNEL_STACK and K_THREAD_STACK.

Threads/Tasks

Thread Stack objects

- **Kernel-only Stacks**
  - If it is known that a thread will never run in user mode, or the stack is being used for special contexts like handling interrupts, it should be defined via K_KERNEL_STACK macros
  - These stacks minimize memory use (e.g. the kernel doesn’t need need to reserve additional room for the privilege elevation stack)
  - Attempts from user mode to use stacks declared in this way will result in a fatal error for the caller.
- **Thread stacks**
  - If it is known that a stack will need to host user threads, or if this cannot be determined, define the stack with K_THREAD_STACK macros. This may use more memory but the stack object is suitable for hosting user threads.

If CONFIG_USERSPACE is not enabled, K_THREAD_STACK and K_KERNEL_STACK macros become equivalent.

Threads/Tasks

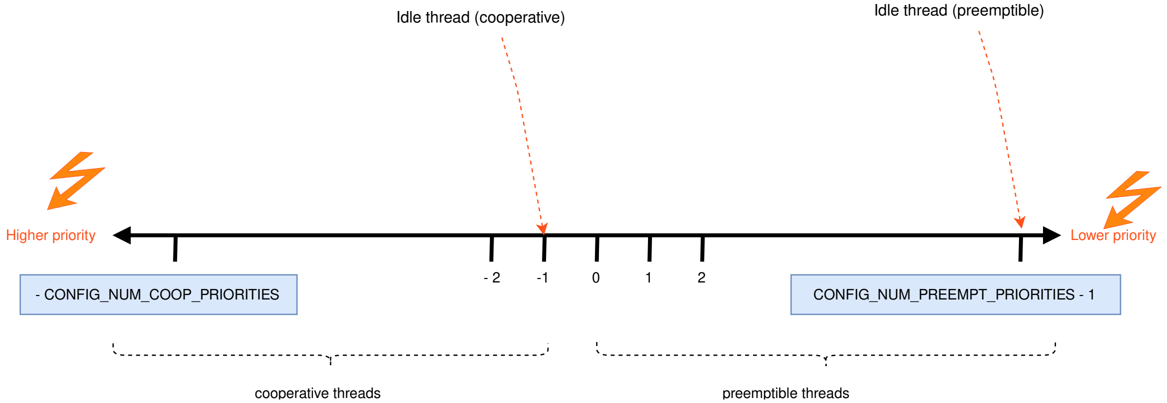
Thread Priorities

- Thread **priority** is an **integer value** (negative or non-negative).
  - Lower priority numbers correspond to higher priority levels
- The scheduler allows **two classes of threads**, based on priority:
  - A **cooperative** thread has a **negative priority value**.
    - Once it becomes the current thread, a cooperative thread remains the current thread until it performs an action that makes it not ready (Suspended or Waiting).
    - **Not well suited for real-time**
  - A **preemptible** thread has a **non-negative priority value**.
    - Once it becomes the current thread, a preemptible thread may be preempted at any time if a cooperative thread, or a preemptible thread of higher or equal priority, becomes ready.
- A thread's initial priority value can be altered dynamically. In particular, a preemptible thread may become a cooperative thread, and vice versa, by changing its priority.

Threads/Tasks

Thread Priorities

- The kernel supports a virtually unlimited number of thread priority levels.
- The configuration options **CONFIG_NUM_COOP_PRIORITIES** and **CONFIG_NUM_PREEMPT_PRIORITIES** specify the number of priority levels for each class, resulting in the following usable priority ranges:
  - Cooperative threads: (-CONFIG_NUM_COOP_PRIORITIES) to -1
  - Preemptive threads: 0 to (CONFIG_NUM_PREEMPT_PRIORITIES - 1)



Threads/Tasks

Implementing tasks/threads

```
#define TASK1_STACK_SIZE 500
#define TASK1_PRIORITY 5
...
void Task1_Code(void *arg1, void *arg2, void *arg3)
{
     /* Task code here */
}
...

K_THREAD_STACK_DEFINE(Task1_stack_area, TASK1_STACK_SIZE);
struct k_thread Task1_thread_data;
k_tid_t Task1_tid;
...

Task1_tid = k_thread_create(&Task1_thread_data, Task1_stack_area,
                          K_THREAD_STACK_SIZEOF(Task1_stack_area),
                          Task1_Code,          /* Pointer to code, i.e. the function name */
                          NULL, NULL, NULL,   /* Three optional arguments */
                          Task1_PRIORITY,
                          0,                  /* Thread options. Arch dependent */
                          K_NO_WAIT);         /* or delay in milliseconds */
```

Threads/Tasks

Implementing tasks/threads

- **Typical structure of a task/thread**

```
void Task1_Code(void *arg1, void *arg2, void *arg3)
{
      /* Initializations (executed only once) */
      ...

      /* Task body – usually never ends */
      while (1) {
            ...
            Do Computations();
            ...
            Some form of Suspend or Wait(); /* Tasks usually don’t run all the time */

      }

    /* Thread terminates if execution reaches this point */
}
```

Threads/Tasks

Implementing tasks/threads

- **Periodic tasks** are common in RT systems
- Simple (but poor, why?) method:

```
void Task1_Code(void *arg1, void *arg2, void *arg3)
{
      /* Initializations (executed only once) */
      ...

      /* Task body – usually never ends */
      while (1) {
            ...
            Do Computations();
            ...
            k_msleep(TASK1_PERIOD);
      }

    /* Thread terminates if execution reaches this point */
}
```

Threads/Tasks

Implementing tasks/threads

- **Sporadic tasks** are also common in RT systems
  - Usually there is a primitive that puts the thread in Waiting/Suspend state

```
void Task1_Code(void *arg1, void *arg2, void *arg3)
{
     /* Initializations (executed only once) */
     ...

     /* Task body – usually never ends */
     while (1) {
          ...
          Do Computations();
          ...
          Blocking call to a semaphore/mutex/event/condition_variable/...
     }

    /* Thread terminates if execution reaches this point */
}
```

Threads/Tasks

Thread runtime statistics

- Runtime statistics can be gathered and retrieved if CONFIG_THREAD_RUNTIME_STATS is enabled
- Example of statistics is the total number of execution cycles of a thread.
- By default runtime statistics are gathered using the default kernel timer. Some platforms have higher resolution timers. The use of these timers can be enabled via CONFIG_THREAD_RUNTIME_STATS_USE_TIMING_FUNCTIONS.
- Example:

```
k_thread_runtime_stats_t rt_stats_thread;

...

k_thread_runtime_stats_get(k_current_get(), &rt_stats_thread);

printk("Cycles: %llu\n", rt_stats_thread.execution_cycles);

...
```

Bibliography

- Kate Stewart, “Zephyr Project: Unlocking Innovation with an Open Source RTOS”, The Linux Foundation, 2021.
- Nordic documentation for kernel services
  - https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/zephyr/index.html#zephyr-project-documentation
  - https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/zephyr/introduction/index.html#introduction
  - https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/zephyr/kernel/services/scheduling/index.html#scheduling
  - https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/zephyr/kernel/services/threads/index.html#threads
  - https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/zephyr/kernel/timeutil.html#time-utilities
  - https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/zephyr/kernel/timeutil.html#time-utilities