

# Multiple Cores, One Zephyr

---

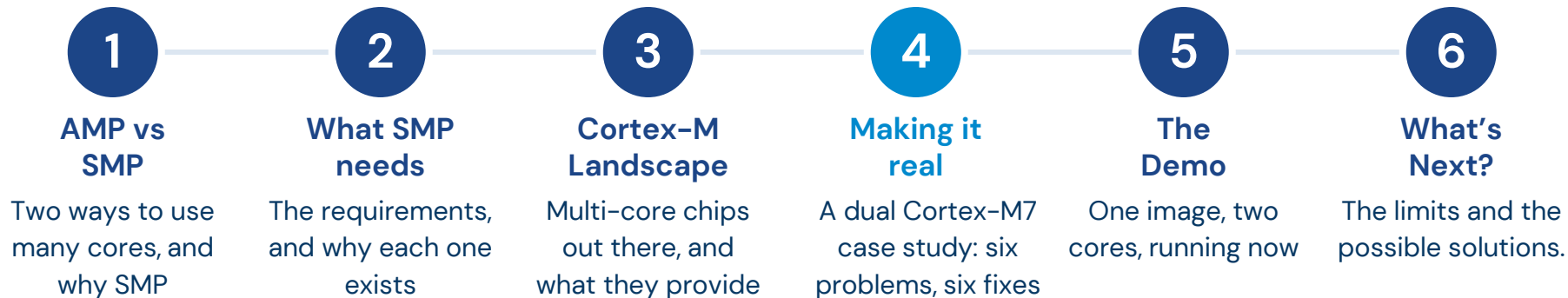
The Path to Cortex-M SMP in Zephyr

# whoami

---

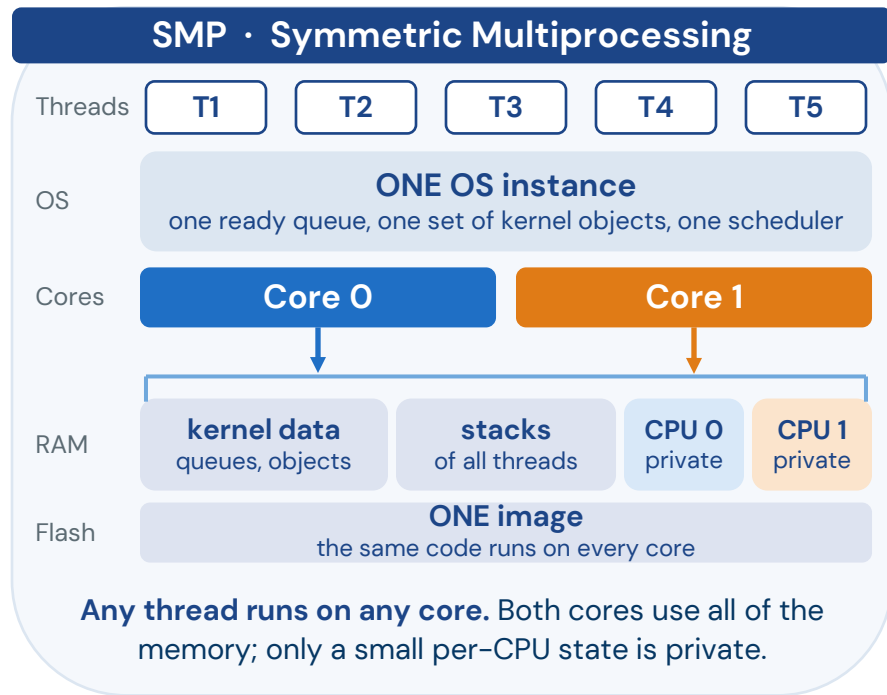
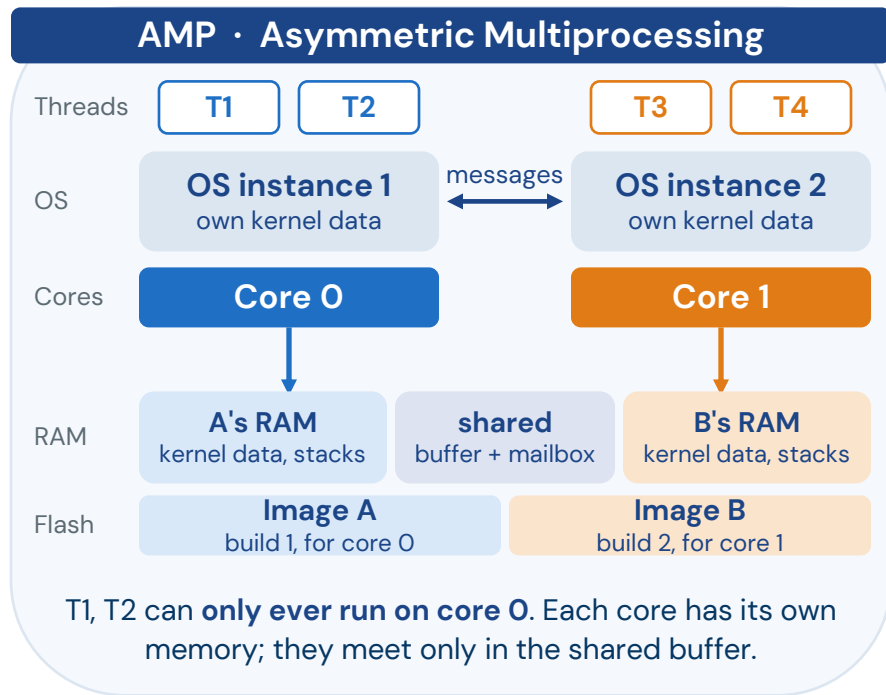
- Sanjay Vallimanalan, Embedded Software Engineer @ Linumiz.
- Primarily works on Zephyr RTOS, contributing to low-level platform enablement and the open-source ecosystem.
- Current upstream activities involve bringing Infineon TRAVEO™ T2G platform support to Zephyr.

# Agenda



# AMP vs SMP

**SMP:** one OS instance on all cores, over one shared memory; any core runs any thread, kernel included.



■ used by core 0 only ■ used by core 1 only ■ shared by both cores

# Why SMP is worth it

## What SMP gives you

- 1 **Real parallelism.** Threads run at the same instant on different cores, instead of taking turns on one.

Two threads, each with 2 units of work:

|             |    |    |    |    |                      |
|-------------|----|----|----|----|----------------------|
| 1 core      | T1 | T2 | T1 | T2 | take turns: $t = 4$  |
| SMP, core 0 | T1 |    |    |    | in parallel: $t = 2$ |
| SMP, core 1 |    | T2 |    |    |                      |

- 2 **Same programming model.** Existing threads, priorities, semaphores, queues and mutexes work across cores unchanged.
- 3 **No static partitioning of peripherals.** Any core could use any available peripherals in a single OS instance without the need for static partitioning.
- 4 **One copy, reachable from every core.** One kernel, one driver per peripheral, one memory pool, all usable from any core.
- 5 **One timer interrupt for all cores.** Only core 0 takes the tick; core 1 takes no timer interrupt, where AMP runs a tick on every core.
- 6 **No stranded capacity.** SMP uses any free core; AMP's design-time split can overload one core while the other idles.

## What it costs

- 1 **Cross-core atomics.** Kernel data is guarded by spinlocks; their atomics must hold across cores.
- 2 **Coherent memory.** Every core must see the same bytes, so caches need coherency or care.
- 3 **Inter-processor interrupts.** Cores must interrupt each other to wake up or reschedule.

# What SMP needs, and why

## THE HARDWARE MUST PROVIDE

H1 Compatible cores, one shared memory

H2 Cross-core atomic operations

H3 Coherent shared memory

H4 Inter-processor interrupts (IPI)

H5 A shared timebase

Zephyr  
SMP kernel

## THE ARCHITECTURE PORT MUST PROVIDE

A1 Core identity

A2 Secondary core start-up

A3 A context switch safe on every core

A4 Interrupt routing

# How Zephyr asks for it

All implemented by the Zephyr port (arch/, soc/ and timer-driver code), grouped by purpose.

## 1 Identity and start-up

Which core am I? Bring the other cores up.

```
static inline struct _cpu *arch_curr_cpu(void);
static inline uint32_t arch_proc_id(void);
int arch_smp_init(void);
void arch_cpu_start(int cpu_num, k_thread_stack_t *stack, int sz,
                    arch_cpustart_t fn, void *arg);
```

<zephyr/arch/arch\_interface.h>

## 2 Scheduling

Switch threads on every core; make another core reschedule.

```
static inline void arch_switch(void *switch_to, void **switched_from);
void arch_sched_broadcast_ipi(void);
void arch_sched_directed_ipi(uint32_t cpu_bitmap);
```

kernel\_arch\_interface.h · arch\_interface.h · SMP depends on USE\_SWITCH · SCHED\_IPI\_SUPPORTED

## 3 Synchronization \*

Atomics that hold across cores; spinlocks build on them.

```
bool atomic_cas(atomic_t *target, atomic_val_t old_value,
                atomic_val_t new_value);
```

<zephyr/sys/atomic\_arch.h> · CONFIG\_ATOMIC\_OPERATIONS\_ARCH (the full atomic\_\* set)

## 4 Time

Start the timer on each secondary core as it joins.

```
extern void smp_timer_init(void);
```

kernel\_internal.h · rule: any CPU may set a timeout; the driver announces each tick once

\* **Not SMP-only:** atomics exist in every build. SMP needs them to be **atomic across cores**, because every spinlock is built on `atomic_cas()`.

# What Cortex-M provides and what a multi-core SoC adds

| Hardware capability                      | The Cortex-M core provides                     | A multi-core SoC must add                                                        |
|------------------------------------------|------------------------------------------------|----------------------------------------------------------------------------------|
| <b>1</b> Compatible cores, shared memory | Standard AHB / AXI bus interfaces.             | Compatible cores on one bus matrix, with one memory map.                         |
| <b>2</b> Cross-core atomics              | LDREX/STREX with a local monitor (v7-M, v8-M). | A global monitor, or hardware spinlocks.                                         |
| <b>3</b> Coherent memory                 | Cache maintenance by address; MPU attributes.  | Uncached shared memory, or the kernel cleans and invalidates caches in software. |
| <b>4</b> Inter-processor IRQ             | An NVIC that accepts any SoC interrupt.        | A mailbox / IPC interrupt into every core.                                       |
| <b>5</b> Shared timebase                 | Per-core SysTick; can read any SoC timer.      | A timer every core can read.                                                     |
| <b>6</b> Core ID                         | CPUID: the core's type and revision.           | A register that tells each core its ID.                                          |
| <b>7</b> Core start-up                   | A relocatable vector table (VTOR).             | Per-core reset and boot-address control.                                         |
| <b>8</b> Interrupt routing               | A private NVIC per core.                       | Per-core interrupt wiring, or a routing mux.                                     |

# Multi-core Cortex-M parts today

| Chip                                 | Cores         | Cross-core lock                               | Core-to-core IRQ             | In Zephyr today                    |
|--------------------------------------|---------------|-----------------------------------------------|------------------------------|------------------------------------|
| Identical cores: SMP candidates      |               |                                               |                              |                                    |
| Raspberry Pi RP2040                  | 2× M0+        | 32 hardware spinlocks (v6-M has no LDREX)     | Inter-core FIFOs             | Board support; core 0 only         |
| Raspberry Pi RP2350                  | 2× M33        | Global exclusive monitor + hardware spinlocks | Doorbells, inter-core FIFOs  | Board support; one core            |
| Infineon TRAVEO T2G (CYT4DN, CYT4BF) | 2× M7 (+ M0+) | IPC hardware locks                            | IPC interrupt structures     | One image per core; SMP: this work |
| Infineon XMC7200                     | 2× M7 (+ M0+) | IPC hardware locks                            | IPC interrupt structures     | One image per core                 |
| Sony Spresense (CXD5602)             | 6× M4F        | SRAM exclusive monitor + hardware semaphores  | Per-core software interrupts | Not supported                      |

## Mixed cores or separate memories: AMP only

STM32H745/H747 (M7 + M4) · i.MX RT1170 (M7 + M4) · PSoC Edge E84 (M55 + M33) · RA8P1 (M85 + M33) · nRF5340 (2× M33, separate memories)

# The case study SoC: TRAVEO T2G CYT4DN



One bus matrix: every core sees the same flash, SRAM and peripherals

Code flash

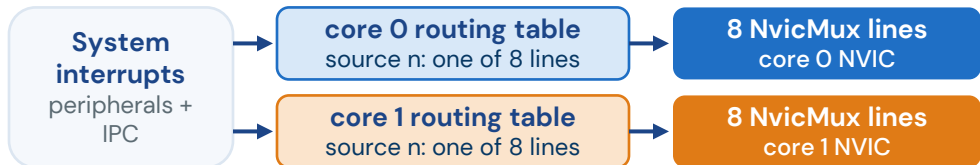
SRAM

Peripherals

## What this SoC adds for SMP



## Interrupt routing **8**



## Capabilities of this SoC

- 1 Compatible cores, shared memory**  
Two identical M7s on one bus matrix
- 2 Cross-core atomics**  
IPC hardware locks; no global exclusive monitor
- 3 Coherent memory**  
No coherency: shared data runs uncached
- 4 Inter-processor IRQ**  
One IPC interrupt structure per core
- 5 Shared timebase**  
TCPWM 32-bit counter (run at 10 MHz)
- 6 Core ID**  
CPUSS\_IDENTITY register
- 7 Core start-up**  
M0+ releases both M7s at one vector base
- 8 Interrupt routing**  
Per-core routing table into 8 NvicMux lines

# Booting Multiple Cortex-M Cores

Boot

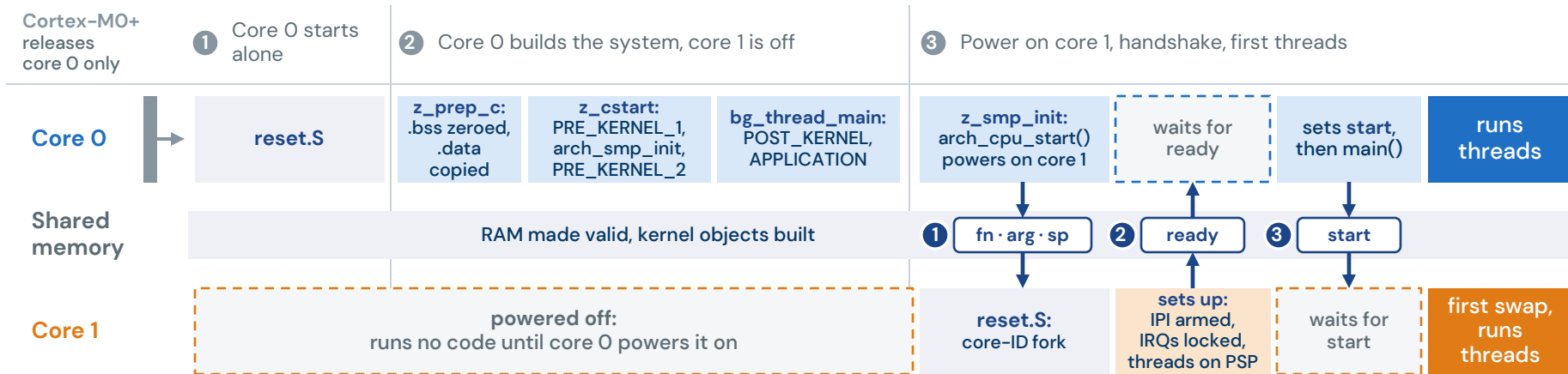
Atomsics

Time

IPI

Context switch

Interrupts



# Cross-core Atomics: LDREX / STREX

Boot

Atomsics

Time

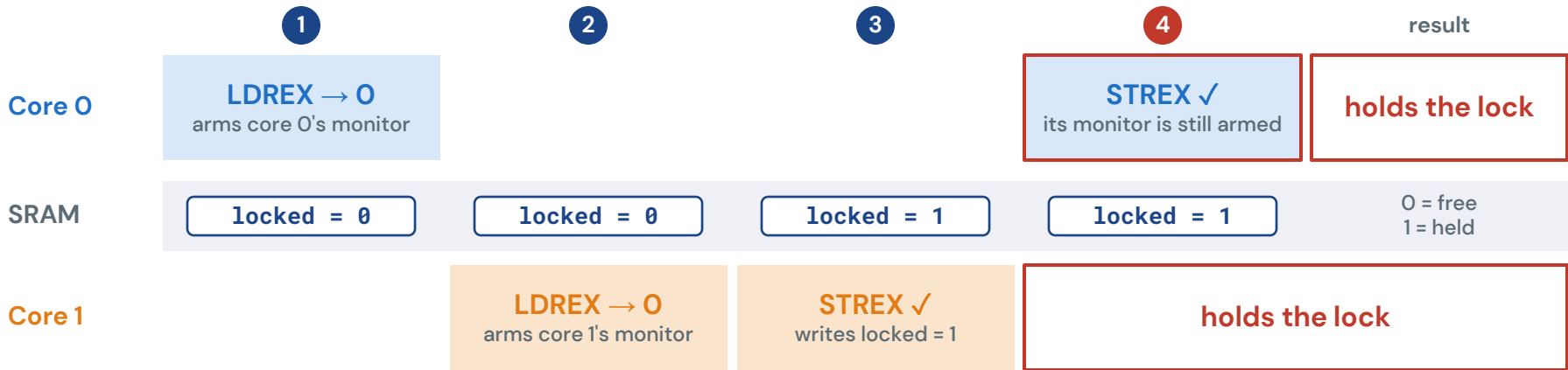
IPI

Context switch

Interrupts

## X Both cores take the same spinlock

`k_spin_lock()` → `atomic_cas(&locked, 0, 1)` → LDREX / STREX (default Cortex-M atomics)



### No global monitor

STREX checks only its own core's monitor. Nothing between the two M7s sees core 1's store at 3, so core 0's STREX at 4 still succeeds: both CASes succeed.

# Cross-core Atomics: The IPC Guard

Boot

Atomics

Time

IPI

Context switch

Interrupts

## ✓ Exactly one core takes the spinlock

`atomic_cas()` → `atomic_ipc.c`: hold one IPC channel only while the atomic runs



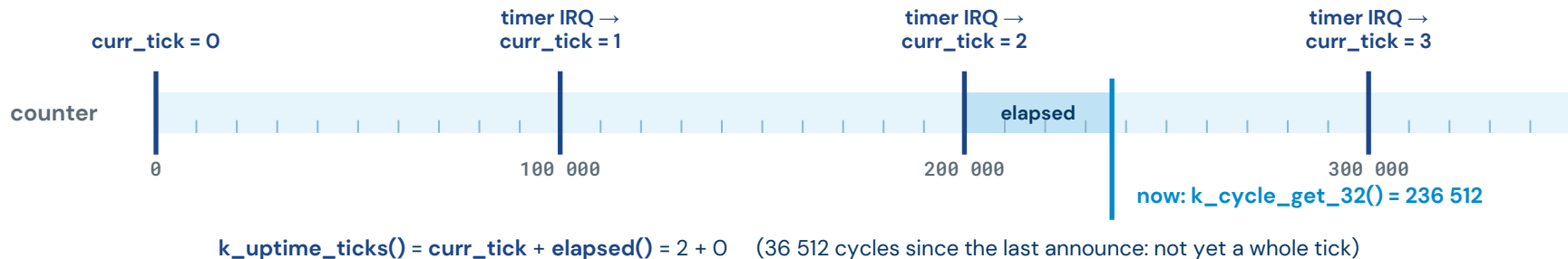
### IPC ACQUIRE is the test-and-set

Reading it returns SUCCESS to exactly one core. With it under `atomic_cas()`, every `k_spinlock` works unchanged.

# Ticks and Cycles

[Boot](#)[Atomsics](#)[Time](#)[IPI](#)[Context switch](#)[Interrupts](#)

Example (`TICKSS_KERNEL = n`) : One hardware counter at 10 MHz, seen two ways (this build: 100 ticks per second)



## Ticks

kernel time · 1 tick = 10 ms

**curr\_tick**: one kernel variable

Moved only by `sys_clock_announce()`, from the timer IRQ

**Used for**: timeouts (`k_sleep`, `k_msleep`, `k_timer`, `k_sem_take`),  
time slicing (20 ms = 2 ticks), uptime

## Cycles

the counter itself · 1 cycle = 100 ns

The hardware counter, 10 000 000 per second

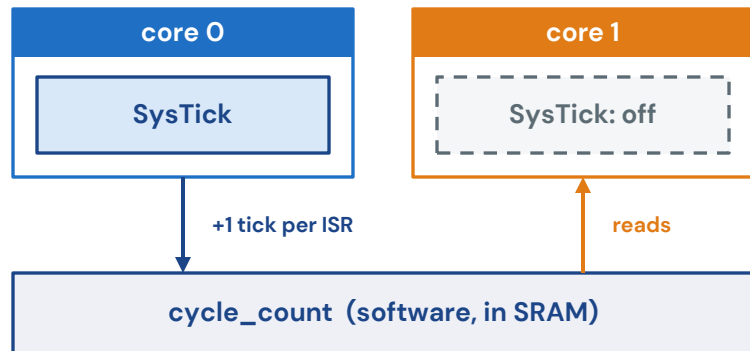
**Read on demand**: no interrupt, no kernel state

**Used for**: `k_cycle_get_32/64()`, `k_busy_wait()`, runtime stats

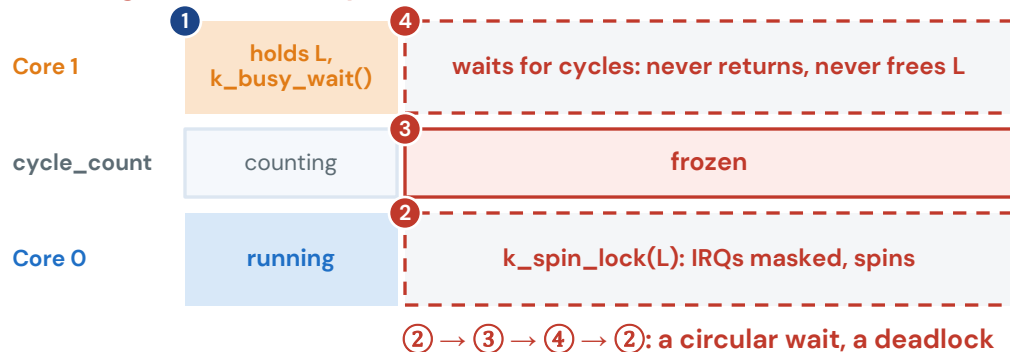
# One SysTick, Two Cores: A Deadlock

[Boot](#)[Atomics](#)[Time](#)[IPI](#)[Context switch](#)[Interrupts](#)

SysTick is private to each core



The hang (tests/kernel/spinlock: bounce test)



k\_busy\_wait(): spin on cycles

kernel/busy\_wait.c

```
void z_impl_k_busy_wait(uint32_t usec_to_wait)
{
    ...
    uint32_t start_cycles = k_cycle_get_32();
    uint32_t cycles_to_wait = k_us_to_cyc_ceil32(usec_to_wait);
    for (;;) {
        uint32_t current_cycles = k_cycle_get_32();
        if ((current_cycles - start_cycles) >= cycles_to_wait) {
            break;
        }
    }
    ...
}
```

on core 1: never exits

k\_cycle\_get\_32() with one SysTick

cortex\_m\_systick.c

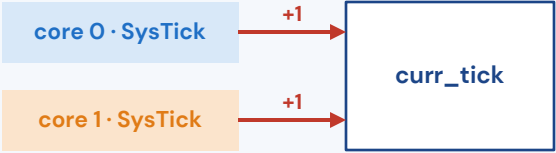
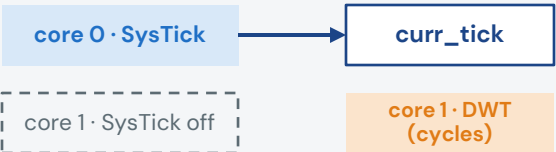
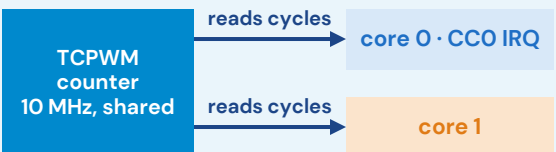
```
static uint32_t elapsed(void)
{
    if (IS_ENABLED(CONFIG_SMP) && (arch_curr_cpu()->id != 0U)) {
        return 0U;
    }
    ...
}
uint32_t sys_clock_cycle_get_32(void)
{
    ...
    uint32_t ret = cycle_count;
    ret += elapsed();
    ...
}
```

core 1: always 0

frozen while core 0 spins

# One Clock for Two Cores

[Boot](#)[Atomics](#)[Time](#)[IPI](#)[Context switch](#)[Interrupts](#)

| Design                                                                                                                                   | Ticks: who announces?                                                                | Cycles: which counter?                                             | Tickless?                                               |
|------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------|--------------------------------------------------------------------|---------------------------------------------------------|
| <p><b>1 Per-core SysTick (the Cortex-M default)</b></p>  | <p>✗ Both cores announce: every tick <b>counted twice</b>, time slices corrupted</p> | <p>✗ One cycle count fed by <b>two unsynchronized</b> SysTicks</p> | <p>— <b>Never reached</b>: ticks were already wrong</p> |
| <p><b>2 Core 0 SysTick + DWT (what we tried)</b></p>     | <p>✓ <b>Core 0's SysTick only</b>; core 1's is switched off</p>                      | <p>✗ Core 1 reads its own DWT: <b>two unrelated clocks</b></p>     | <p>✗ Core 1 has <b>no timer to arm</b></p>              |
| <p><b>3 One shared timer/counter</b></p>                 | <p>✓ CCO compare interrupt on core 0: <b>announced once</b></p>                      | <p>✓ One counter, 10 MHz, <b>read by both cores</b></p>            | <p>✓ CCO armed for the <b>next deadline</b></p>         |

# How IPIs help in SMP

[Boot](#)[Atomsics](#)[Time](#)[IPI](#)[Context switch](#)[Interrupts](#)

Without IPIs: A and B are high priority, C is low. Core 0 runs A, core 1 runs C, B is asleep. One tick fires on core 0.

Core 0

runs A

timer IRQ: B's sleep ends

A keeps running: B has the same priority

Shared  
kernel  
state

B made ready

ready queue: B (high priority, waiting for a core)

Core 1

runs low-priority C, while high-priority B waits: priorities broken

Every time one core decides something another core must act on

A higher-priority thread becomes ready

e.g. B's sleep ends on core 0

**without an IPI:**

core 1 keeps running a lower-priority thread

`z_ready_thread()`

Core 1's time slice ends

only core 0 has the timer

**without an IPI:**

the thread keeps its core forever: no round-robin

`slice_timeout()`

A thread becomes ready, core 1 is idle

core 1 is asleep in WFI

**without an IPI:**

core 1 sleeps while the thread waits in the queue

`z_ready_thread()`

Abort or suspend a thread running on core 1

`k_thread_abort()` on core 0

**without an IPI:**

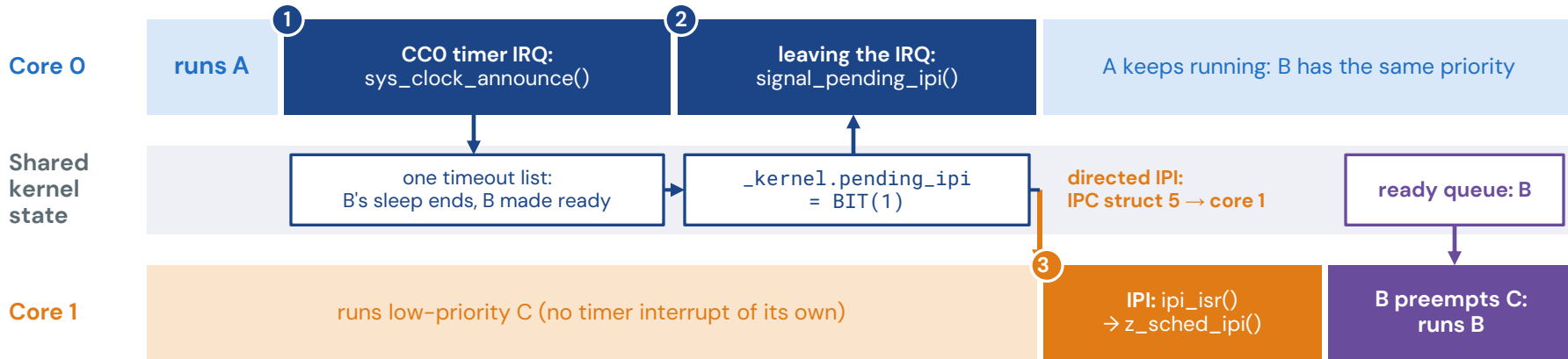
the thread keeps running

`z_thread_halt()`

# IPI: Telling the Other Core

[Boot](#)[Atomics](#)[Time](#)[IPI](#)[Context switch](#)[Interrupts](#)

A and B are high priority, C is low. Core 0 runs A, core 1 runs C, B is asleep. Then one tick fires on core 0.



## 1 One announcer, one timeout list

Core 0's tick fires every timeout, whichever core the woken thread belongs on. Readying B flags the other core, because it runs a lower-priority thread.

```
z_ready_thread(B)
→ flag_ipi(ipi_mask_create(B))
```

## 2 Flag now, send on the way out

Timeout handlers only set a bit in `_kernel.pending_ipi`. Core 0 sends the IPI when it leaves the interrupt, still holding the scheduler lock.

```
signal_pending_ipi()
→ arch_sched_directed_ipi(BIT(1))
```

## 3 Core 1 is told, not timed

The IPI makes core 1 enter its kernel; on the way out its scheduler sees that B outranks C and switches. An idle core asleep in WFI is woken the same way.

```
ipi_isr() → z_sched_ipi()
then the switch to B on IRQ exit
```

# Why SMP Needs CONFIG\_USE\_SWITCH

Boot   Atomics   Time   IPI   **Context switch**   Interrupts

Legacy: `arch_swap()` + PendSV, as if it ran on two cores

Core 0

B made ready →  
`update_cache(): cache = B`

\_kernel

2

`ready_q.cache = B`

one global slot: written early, read later

1

`z_swap() → PendSV:`  
`current = cache = B` ✗

Core 1

its scheduler takes B and runs it

⇒ B runs on both cores

CONFIG\_USE\_SWITCH: `arch_switch()`, the same moment

Core 0

under the lock: pick next  
for this core = B

3

`arch_switch(B, &A->switch_handle)`

runs B

Core 1

under the lock: pick next  
= D (B left the queue)

`arch_switch(D, &C->switch_handle)`

runs D

decide and switch together, per core: the choice is passed as an argument

## 1 No argument

`z_swap()` switches to a decision cached when the scheduler last ran. The other core may already be running it.

## 2 Hard-wired to CPU 0

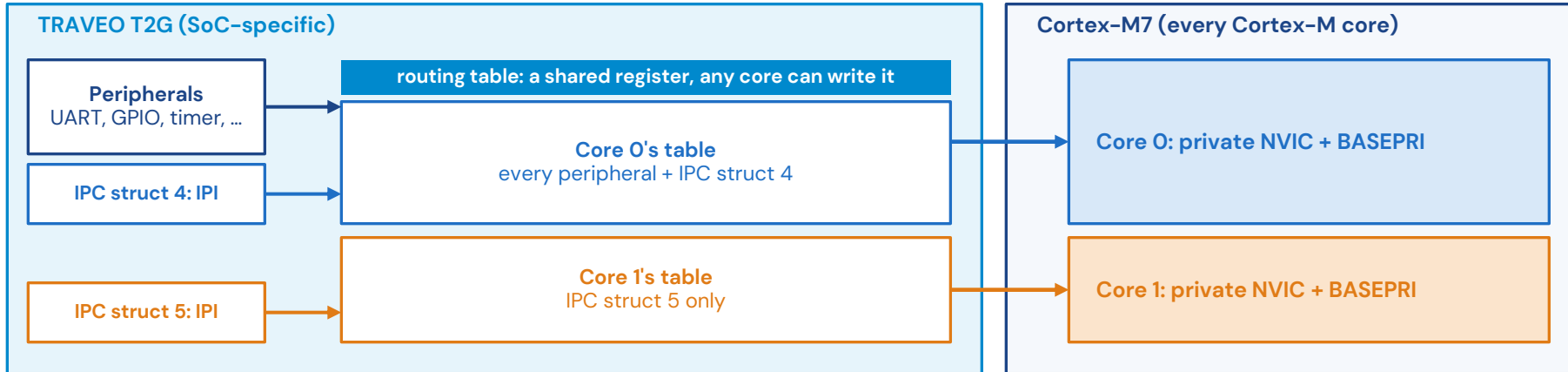
The PendSV code reads and writes `_kernel.cpus[0].current` and the one `ready_q.cache`.

## 3 Decide and switch together

`arch_switch(to, &from)` uses the choice made under the lock, and publishes the old handle once the thread is saved.

kernel/smp/Kconfig:   `config SMP`   ...   depends on `USE_SWITCH`

# Interrupts with a Private NVIC per Core

[Boot](#)[Atomics](#)[Time](#)[IPI](#)[Context switch](#)[Interrupts](#)

## Cortex-M: each core has a private NVIC

**One core only.** Enable, disable, pend and priority reach your own NVIC, never the other core's.

**Masking is per core.** BASEPRI blocks interrupts on one core only, so under SMP the kernel turns `irq_lock()` into a global spinlock.

**No way to interrupt the other core.** The IPI needs something outside the core.

## This SoC: one shared routing table in front

**All peripherals go to core 0.** `irq_enable()` and `irq_disable()` from either core write core 0's entry, so an interrupt's owner never moves.

**Core 1 gets only its IPI.** IPC structs 4 and 5 give each core its own IPI.

**Each core sets its own IPI priority.** The lines power up at 0, which BASEPRI cannot mask.

# Demo of Cortex-M SMP

```
T1 on CPU 1
T0 on CPU 1
T2 on CPU 1
T1 on CPU 0
T0 on CPU 0
T2 on CPU 0
T1 on CPU 1
T0 on CPU 1
T2 on CPU 1
T1 on CPU 0
T0 on CPU 0
T2 on CPU 0
T1 on CPU 1
T0 on CPU 1
T2 on CPU 1
T1 on CPU 0
T0 on CPU 0
T2 on CPU 0
T1 on CPU 1
T0 on CPU 1
T2 on CPU 1
T1 on CPU 0
T0 on CPU 0
T2 on CPU 0
T1 on CPU 1
T0 on CPU 1
T2 on CPU 1
T1 on CPU 0
*** Booting Zephyr OS build v4.4.0-10060-gf50f3fc81771 ***

CYT4DN dual-M7 SMP demo (num_cpus=2)
T0 on CPU 1
T1 on CPU 0

2 on CPU 0
;32uartBTN0: thread running on CPU 0, waiting for presses
BTN1: thread running on CPU 0, waiting for presses
:~$ T0 on CPU 1
T1 on CPU 0
T2 on CPU 1

CTRL-A Z for help | 115200 8N1 | NOR | Minicom 2.10 | VT102 | Offline | ttyACM0
```

# What's Next

## Still to do in the port

### Cache coherency

Today: CONFIG\_DCACHE=n (caches do not snoop)

→ **CONFIG\_KERNEL\_COHERENCE**

- kernel data uncached, thread stacks cached
- mark the uncached region (one MPU region)
- flush / invalidate stacks on context switch

### Interrupt balancing

Today: every device interrupt on core 0

→ **Static partitioning**

- one owner core per interrupt, from devicetree
- route it and enable it on that core

## Road to upstream

### 1 Cortex-M SMP in arch/arm

per-core switch scratch, a SoC hook for arch\_curr\_cpu(), per-core irq\_offload, a secondary-core entry hook in reset.S

### 2 CYT4DN SoC support

secondary start, IPC atomics and IPIs, interrupt routing

### 3 TCPWM shared-timebase timer

one counter both cores read, tickless capable

### 4 RFC, then a second SoC

XMC7200 has the same IPC block; RP2350 has a global exclusive monitor

Visit us on Zephyr Booth

**Zephyr Booth 33, Floor 2**

**Open Source Summit, Europe 2026**

**7–9 October 2026, Prague Congress Centre, Prague**

# Thank You!

---

Any questions?  
Contact us: [contact@linumiz.com](mailto:contact@linumiz.com)

