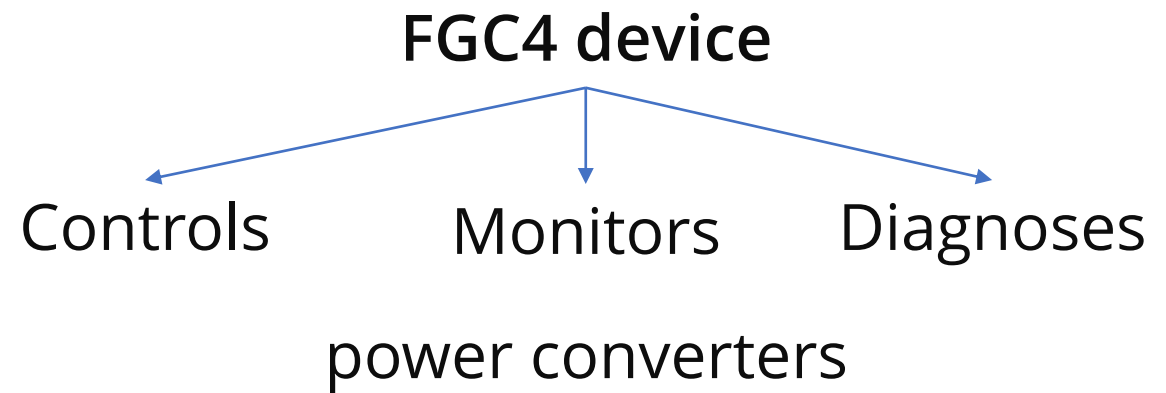
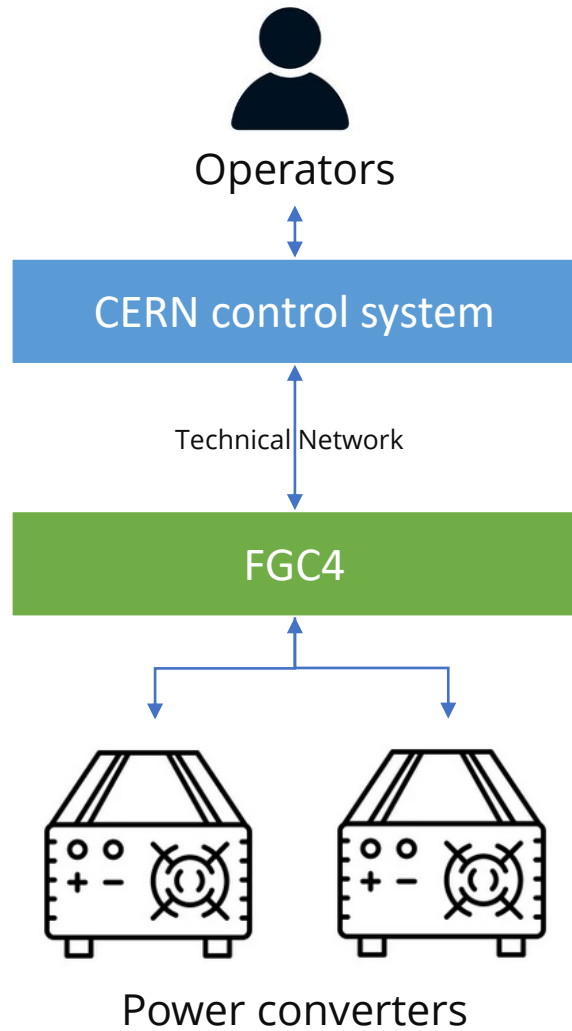


# Bare-metal Programming on Zynq UltraScale+ for the FGC4 Power Converter Controller

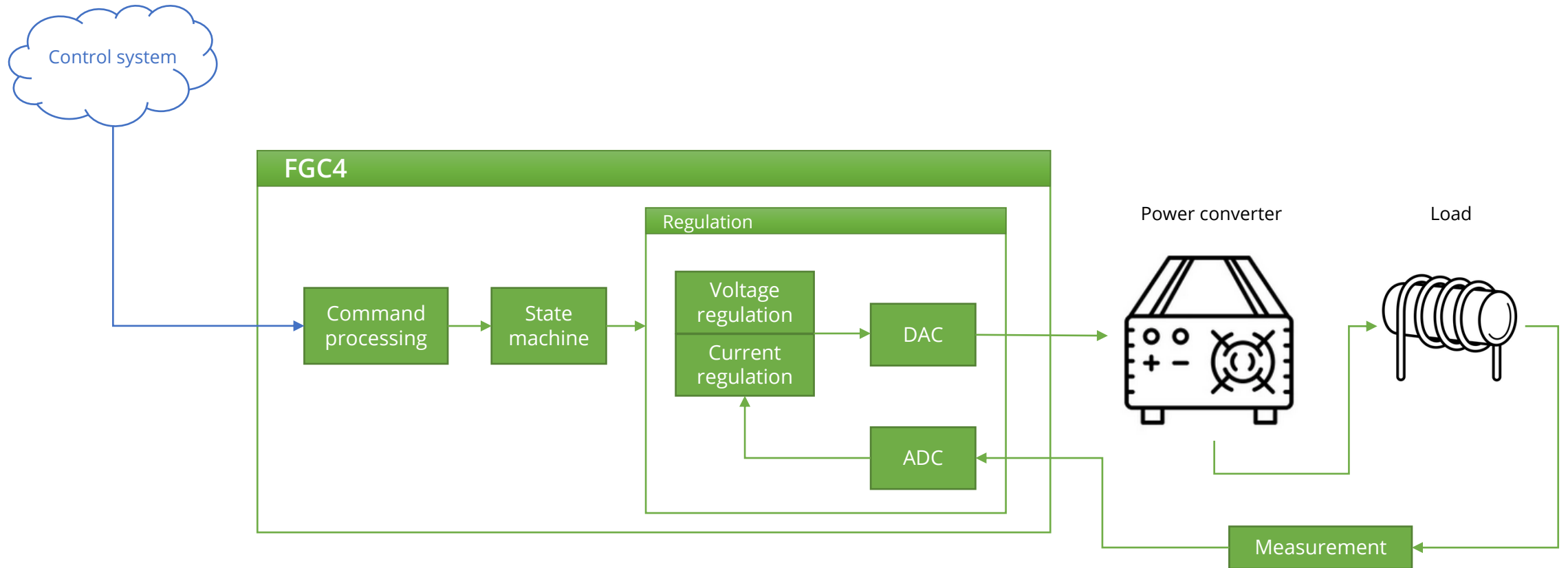
---

Martin Cejp, Dariusz Zielinski  
3rd CERN SoC Workshop 2023-10-04

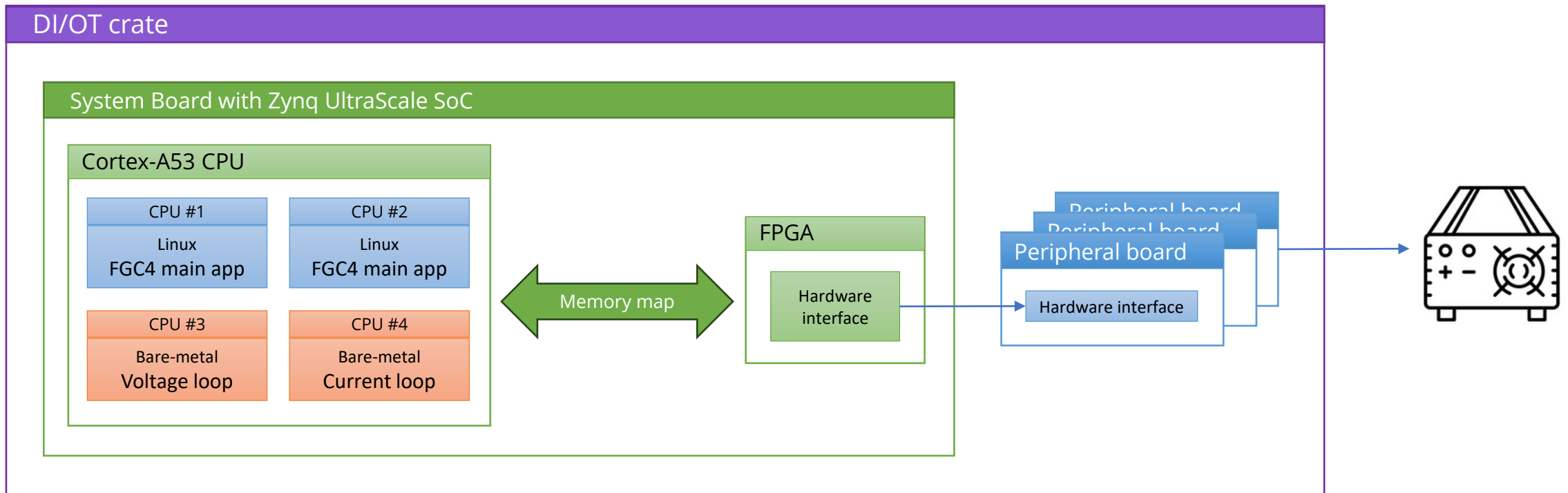
- Project introduction
- Architecture
- Our needs and constraints
- Possible solutions
- Bare-metal bootloader and utilities



# FGC4 project – basic principle



# FGC4 project – architecture



## We want to use Linux to:

- Reuse code with the FEC software.
- Take advantage of all available libraries, network stack, file system, etc.
- Make it easier to develop, debug and maintain our software.

## We want to use Linux to:

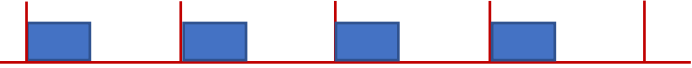
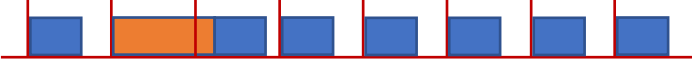
- Reuse code with the FEC software.
- Take advantage of all available libraries, network stack, file system, etc.
- Make it easier to develop, debug and maintain our software.

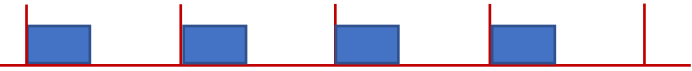
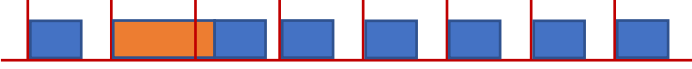
## But we also want to:

- Achieve around 100 kHz current regulation frequency.
- Be able to run hard-real-time code with absolute determinism.
- Be able to run lots of calculations in a short amount of time.

Problem type	What you need	Typical solution
Real-time problem	<p data-bbox="952 448 1212 486">Determinism</p>  <p data-bbox="1021 605 1467 629">Tasks have to finish at the deadline.</p>	<p data-bbox="1704 448 2333 586">Microcontrollers, DSPs - simpler CPUs (e.g. Cortex-M, Cortex-R), FPGAs</p> <p data-bbox="1704 619 2053 658">Bare-metal, RTOS</p>



Problem type	What you need	Typical solution
Real-time problem	<p data-bbox="952 448 1212 486">Determinism</p>  <p data-bbox="1021 605 1467 629">Tasks have to finish at the deadline.</p>	<p data-bbox="1704 448 2333 586">Microcontrollers, DSPs - simpler CPUs (e.g. Cortex-M, Cortex-R), FPGAs</p> <p data-bbox="1704 619 2053 658">Bare-metal, RTOS</p>
Number-crunching problem	<p data-bbox="952 733 1187 772">Throughput</p>  <p data-bbox="937 891 1556 943">With margin, even if a task overruns, it's still the same number of tasks per unit of time.</p>	<p data-bbox="1704 733 2346 815">SoCs, servers - more complex CPUs (e.g. Cortex-A, x86-64, etc.)</p> <p data-bbox="1704 891 1819 929">Linux</p>

Problem type	What you need	Typical solution
Real-time problem	<p data-bbox="952 448 1212 486">Determinism</p>  <p data-bbox="1021 605 1467 629">Tasks have to finish at the deadline.</p>	<p data-bbox="1704 448 2333 586">Microcontrollers, DSPs - simpler CPUs (e.g. Cortex-M, Cortex-R), FPGAs</p> <p data-bbox="1704 619 2053 658">Bare-metal, RTOS</p>
Number-crunching problem	<p data-bbox="952 733 1187 772">Throughput</p>  <p data-bbox="932 891 1556 943">With margin, even if a task overruns, it's still the same number of tasks per unit of time.</p>	<p data-bbox="1704 733 2346 815">SoCs, servers - more complex CPUs (e.g. Cortex-A, x86-64, etc.)</p> <p data-bbox="1704 891 1819 929">Linux</p>
High-frequency regulation	<p data-bbox="952 1019 1531 1058">Determinism and throughput</p>	<p data-bbox="1704 1019 1760 1058">???</p>

How to address these needs?

## Idea #1

Let's just use Linux...

## Frequency aim for the FGC4:



Preliminary tests showed that a regulation iteration takes:

**4-6 us** (*depending on a case*)

Which gives us a worst-case margin of **~4 us**  
(*not counting FPGA communication latency yet*)

## Can we force Linux to be deterministic?

### We tried:

- Applying real-time patch (RT patch).
- Setting CPU affinity for the process.
- Setting interrupts affinity.
- Memory locking
- Kernel tweaking (*isolcpus, nohz\_full, rc\_nocbs, irq\_affinity*)

→ Best achieved interruption time: 4 us.

While the available margin is 4 us....

The Kernel interrupts the process around every 4ms (250 Hz) which is the “Kernel tick” – present even in the “tickless” kernels.

### Conclusion

You cannot have hard real-time determinism on Linux

## In summary we tried:

- RT patch → Not enough
- Kernel config tweaking → Not enough
- Isolation patch → Not ready yet (*hobby project*)
- Xenomai project → Quite complex (*Jailhouse seems better*)
- Jailhouse project → Not so popular, difficult set-up

## Idea #2

Let's follow Xilinx recommendation  
and use Cortex-R5...



# Possible solutions - Idea #2



We run benchmarks on Cortex-A53:

Scenario	FPGA access	Logging	Mean rate [kHz]		Slowest rate [kHz]		Slowest iteration [ $\mu$ s]	
			NXP	Zynq	NXP	Zynq	NXP	Zynq
Idle	No	No	788	284	757	273	1.32	3.66
Idle	No	Yes	285	106	240	100	4.16	10.00
Idle	Yes	No	389	249	378	242	2.64	4.12
Idle	Yes	Yes	215	100	195	93	5.12	10.74
Direct	No	No	724	261	694	253	1.44	3.94
Direct	No	Yes	335	119	266	111	3.76	8.94
Direct	Yes	No	377	230	373	222	2.68	4.50
Direct	Yes	Yes	238	111	208	101	4.80	9.86

Benchmarks results.

We barely achieved 100 kHz on Cortex-A53 running 1.2 GHz.  
Cortex-R5 has a simpler microarchitecture and runs at ~500 MHz

## Conclusion

Cortex-R5 is too slow.

## Idea #3

Let's run bare-metal on Cortex-A53...

## Bare-metal approach:

- Limit the **RAM memory** and **number CPU of cores** visible in Linux (*by adjusting the Device Tree*).
- **Compile** the binary for **bare-metal target**.
- **Load the binary** at given memory address and **start CPU core** at this address.

## Bare-metal approach:

- Limit the **RAM memory** and **number CPU of cores** visible in Linux (*by adjusting the Device Tree*).
- **Compile** the binary for **bare-metal target**.
- **Load the binary** at given memory address and **start CPU core** at this address.

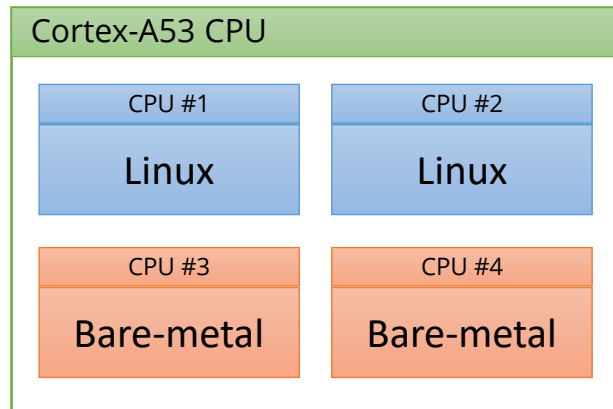
## Challenges:

- Proper compilation of bare-metal applications.
- Handling different Exception Level.
- Starting, interrupting, reloading and monitoring of the bare-metal program.
- Cache, MMU, and interrupts configuration.
- Inter-processor communication via shared memory.

How to run bare-metal app on Cortex-A53?

## Naive approach:

1. Write and compile code
2. Copy it to memory (through Linux)
3. Clear reset bit of BM core
4. Celebrate your success



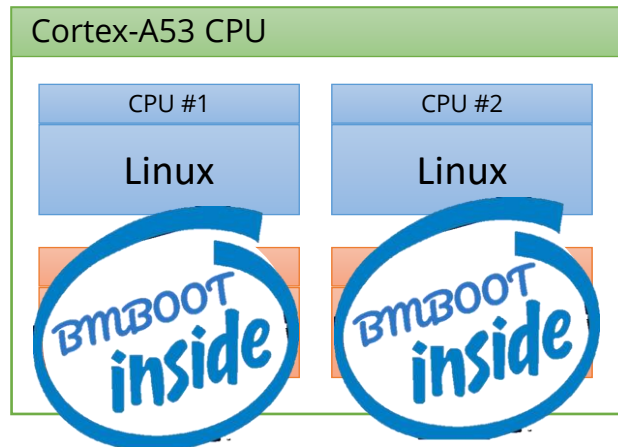
## Just a few obstacles...

1. Code may be erroneous → application must be able to restart after a crash, while protecting the rest of the system
2. Cannot reliably stop application once started
3. Some peripherals are shared between Linux and bare metal (interrupt controller) and must be protected accordingly

## Solution: OpenAMP?

## Our solution: Bmboot

- A minimalist loader & monitor for bare-metal code
- Executes on a high privilege level (EL3) to protect itself from errors in user code
- Zero overhead when user code executing, but can be called upon to intervene



## How to use it?

```
#include <bmboot/payload_runtime.hpp>
```

```
int main(int argc, char** argv)
{
    bmboot::notifyPayloadStarted();

    printf("Hello, world!\n");

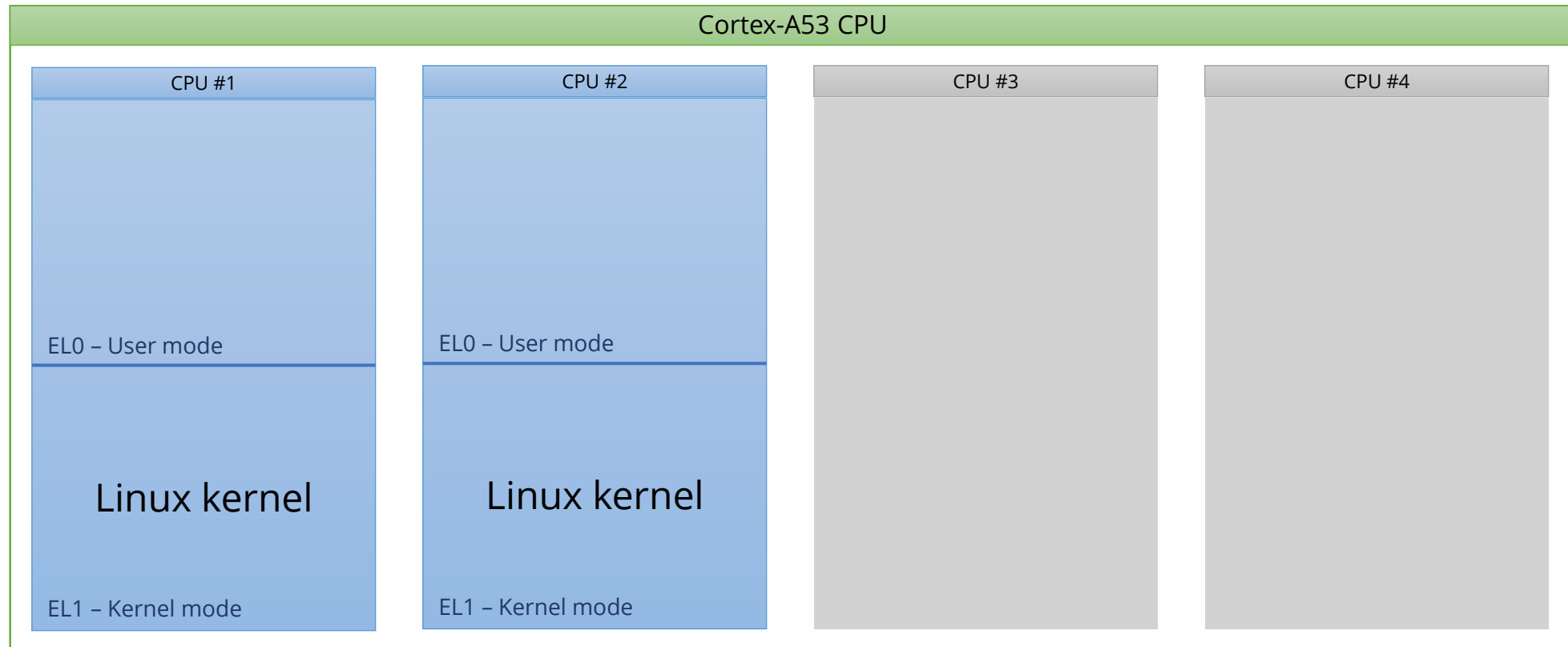
    for (;;) {}
}
```

```
root@diot:~# bmctl exec cpu3 hello_world.bin
Hello, world!
```

- Write C++ code (being aware of limitations of the bare-metal environment)
- Link to the Bmboot SDK
- Launch monitor + user application from Linux (via CLI or API)

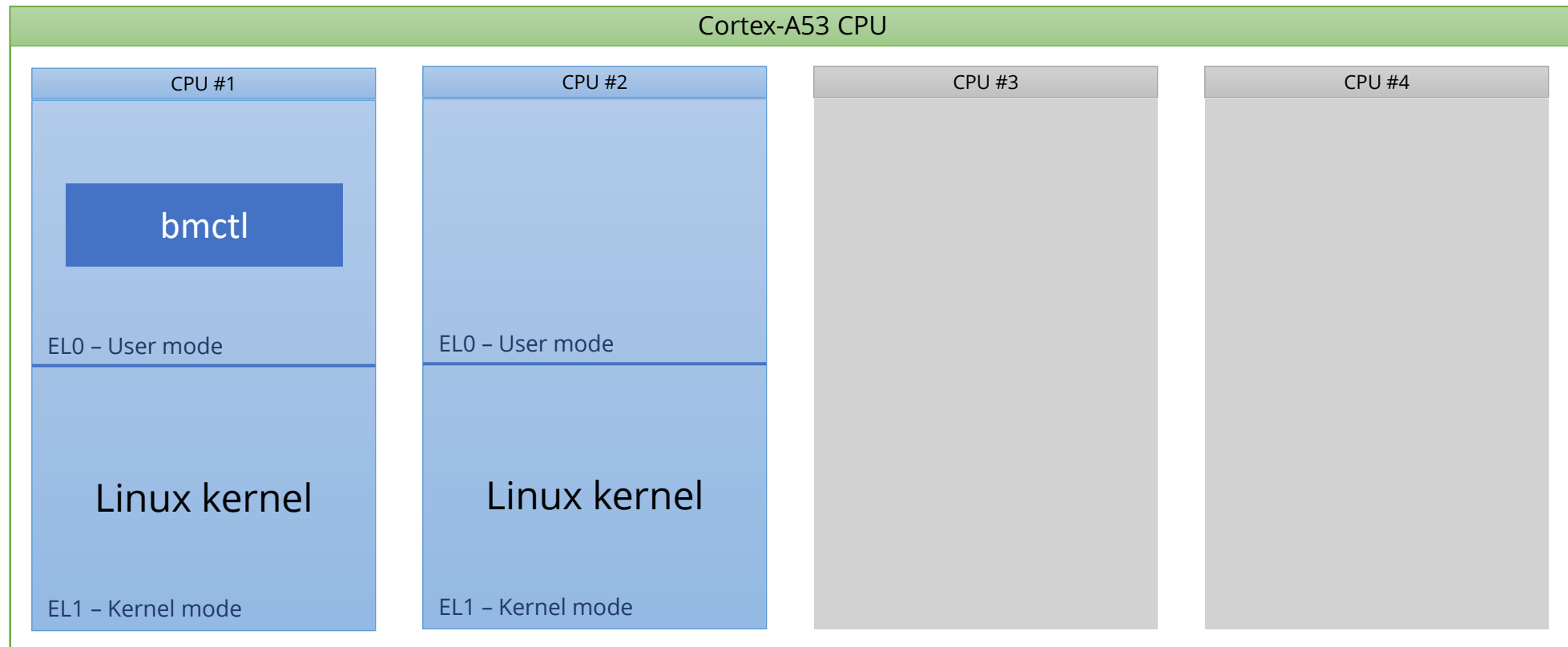


## How does it work?



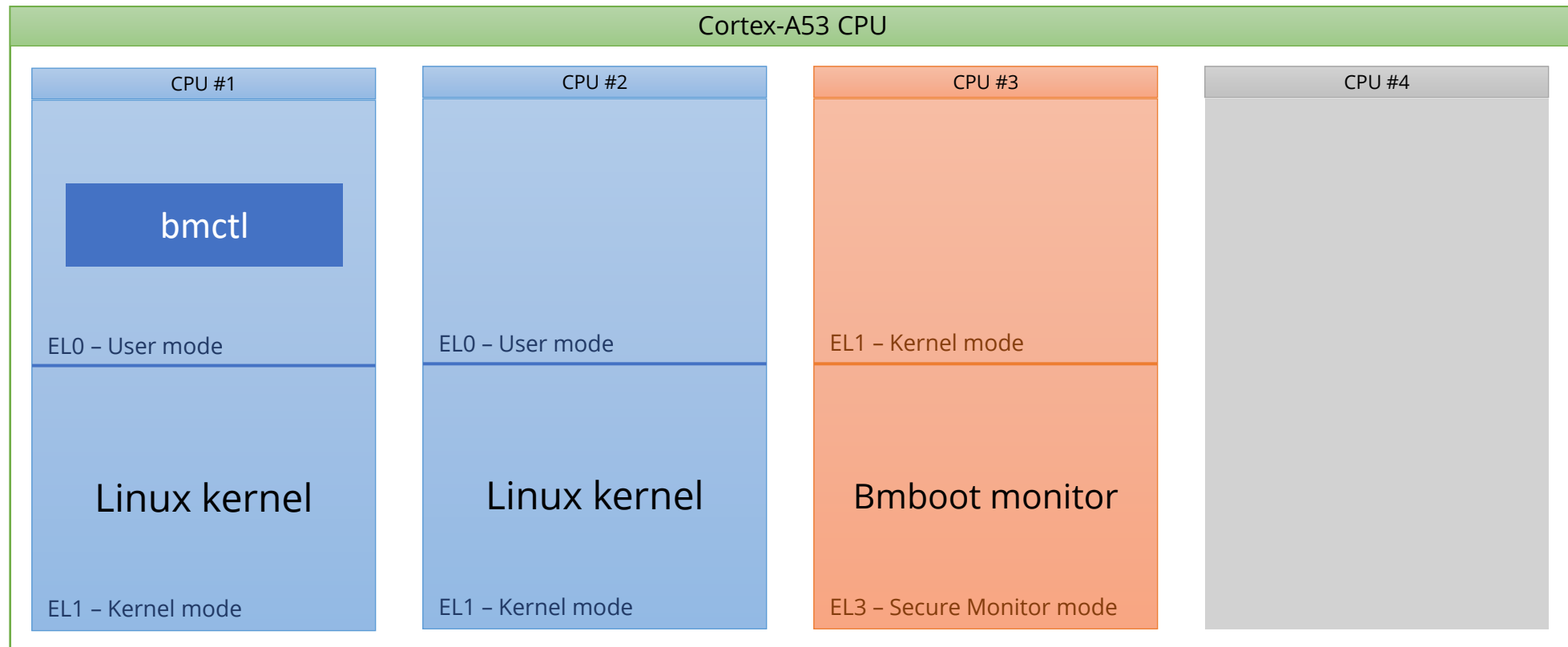
# How does it work?

```
$ bmctl startup cpu3
```



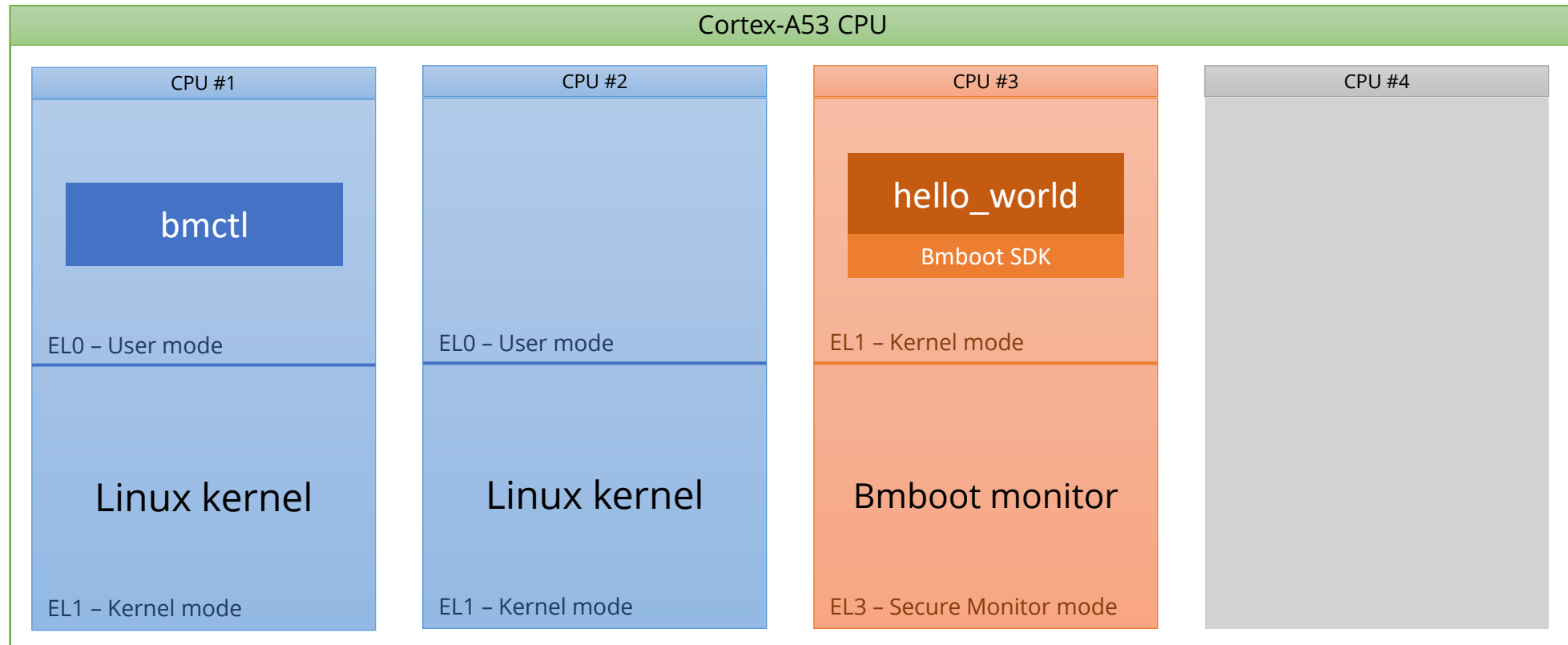
# How does it work?

```
$ bmctl startup cpu3
```



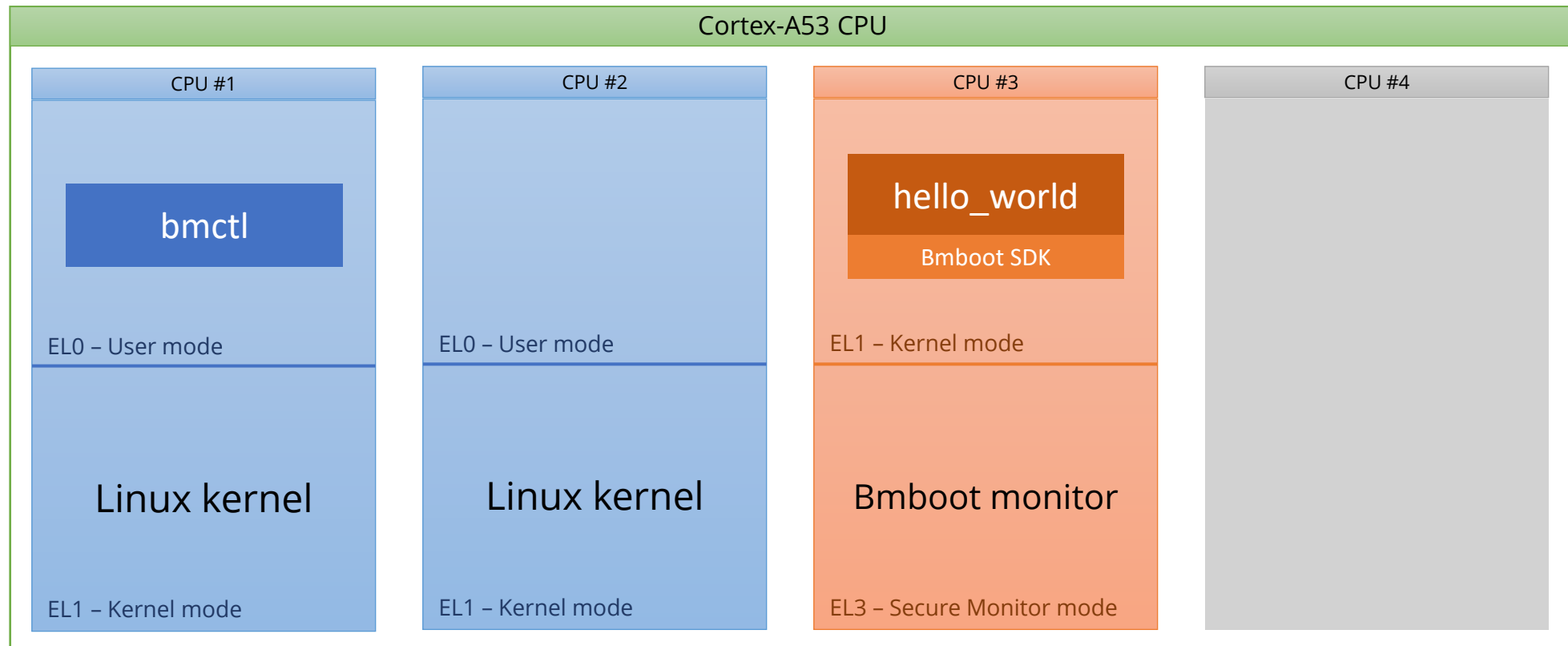
# How does it work?

```
$ bmctl startup cpu3  
$ bmctl exec cpu3 hello_world.bin
```



# How does it work?

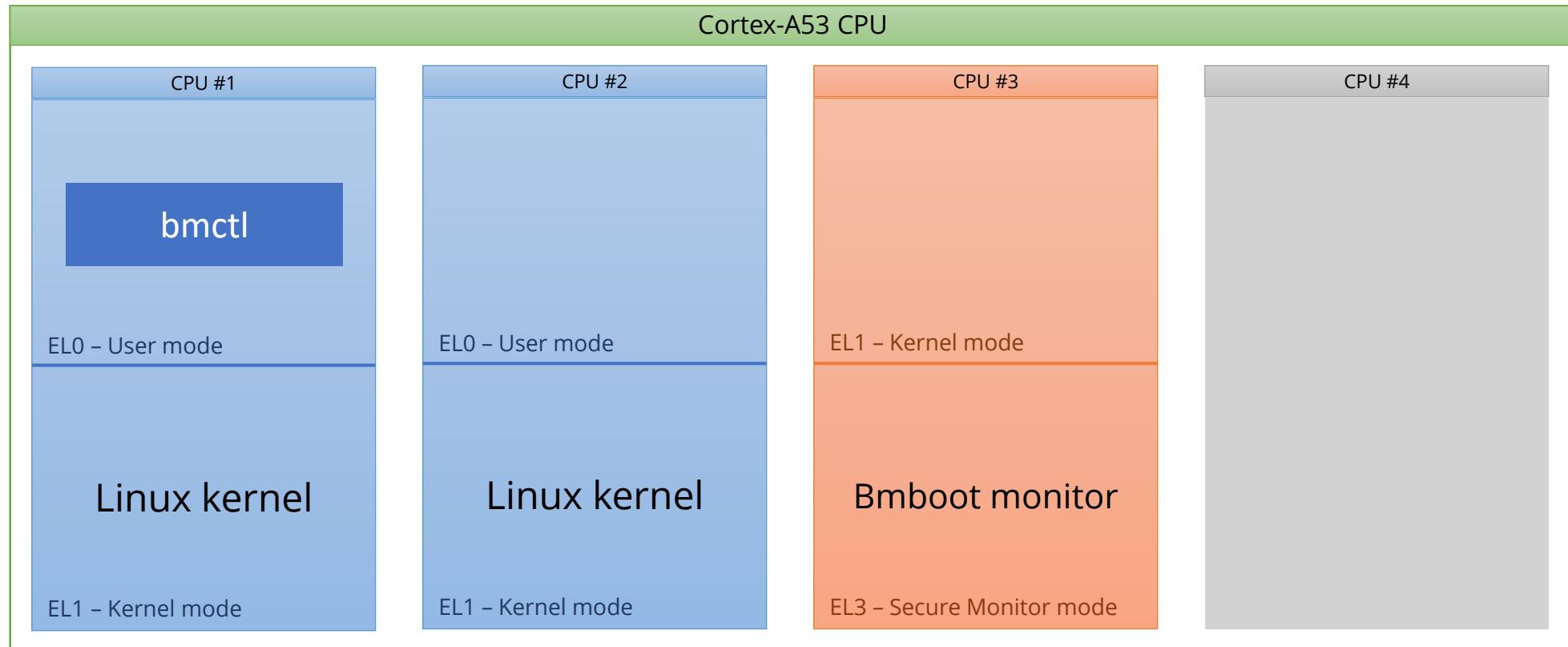
```
$ bmctl startup cpu3  
$ bmctl exec cpu3 hello_world.bin  
$ bmctl terminate cpu3
```



# How does it work?

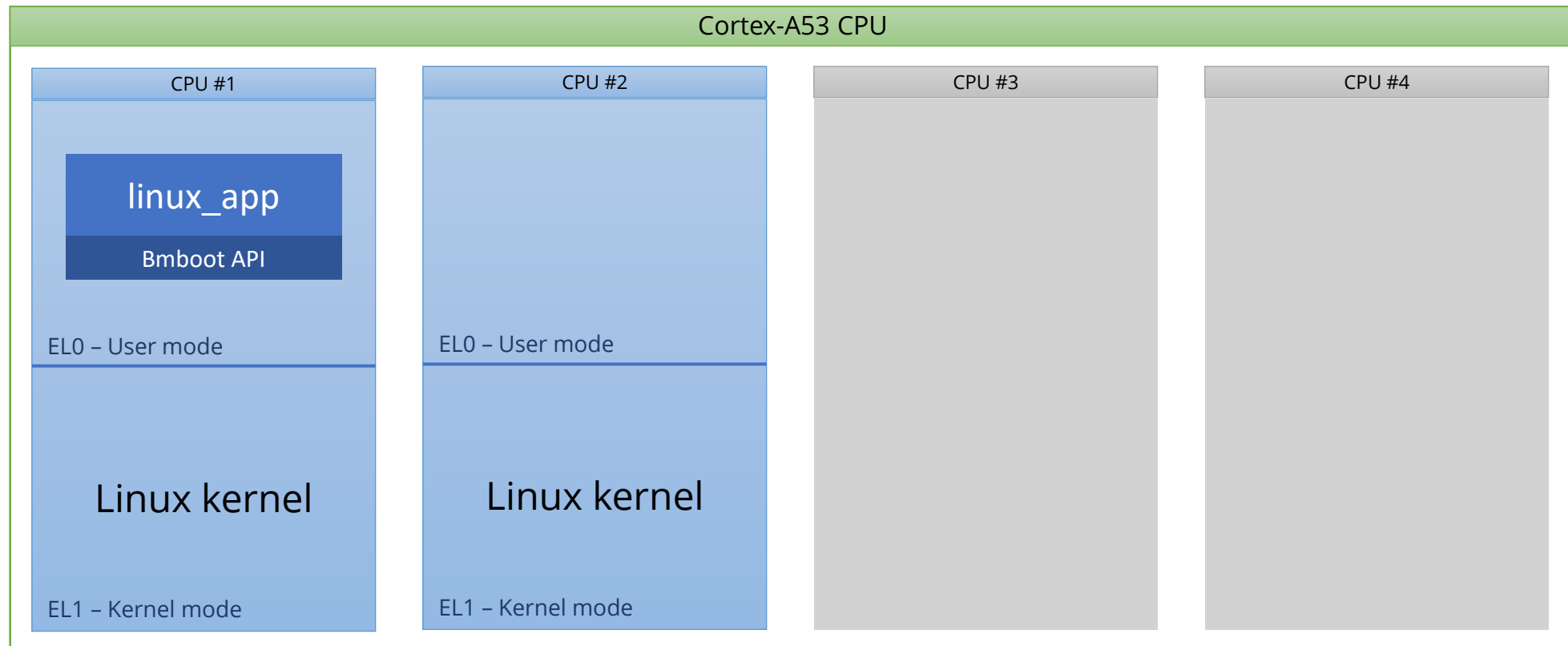


```
$ bmctl startup cpu3  
$ bmctl exec cpu3 hello_world.bin  
$ bmctl terminate cpu3
```



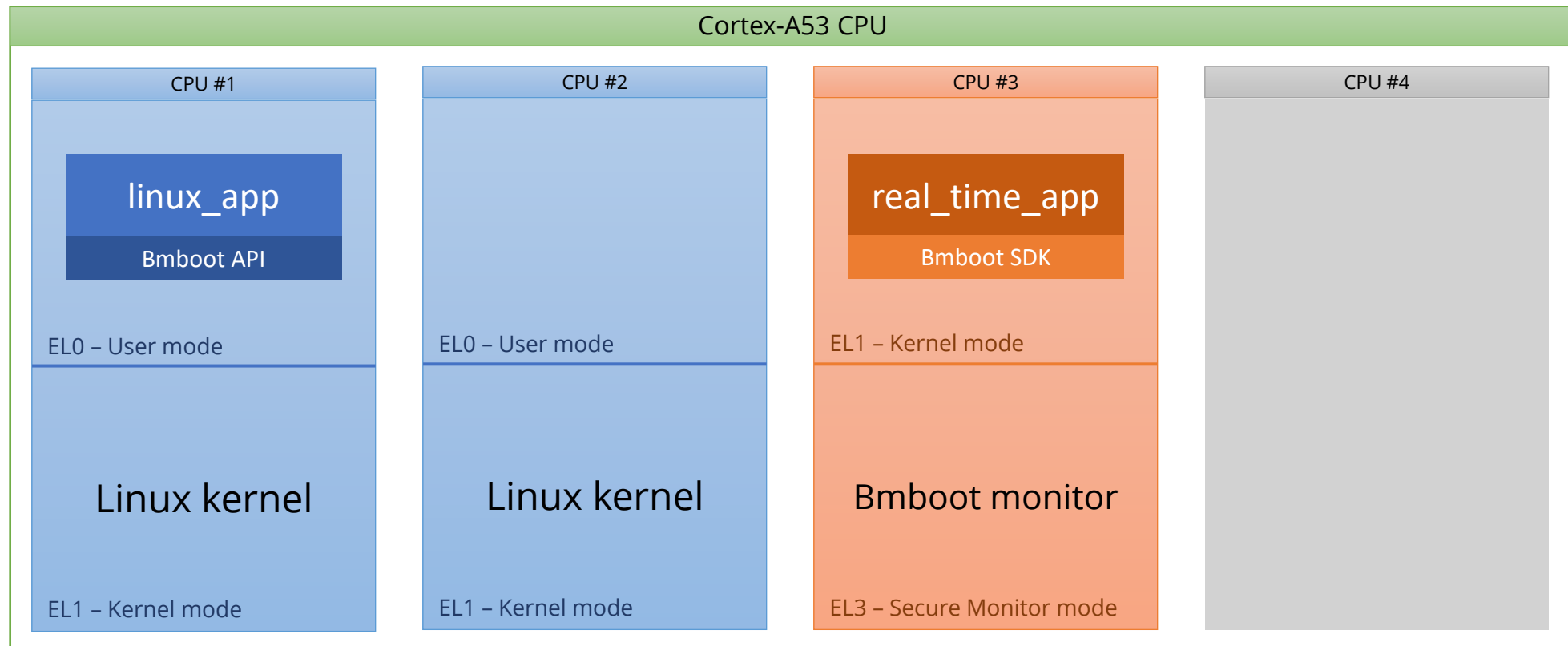
# How does it work?

Lifecycle can also be managed by user application on Linux via API



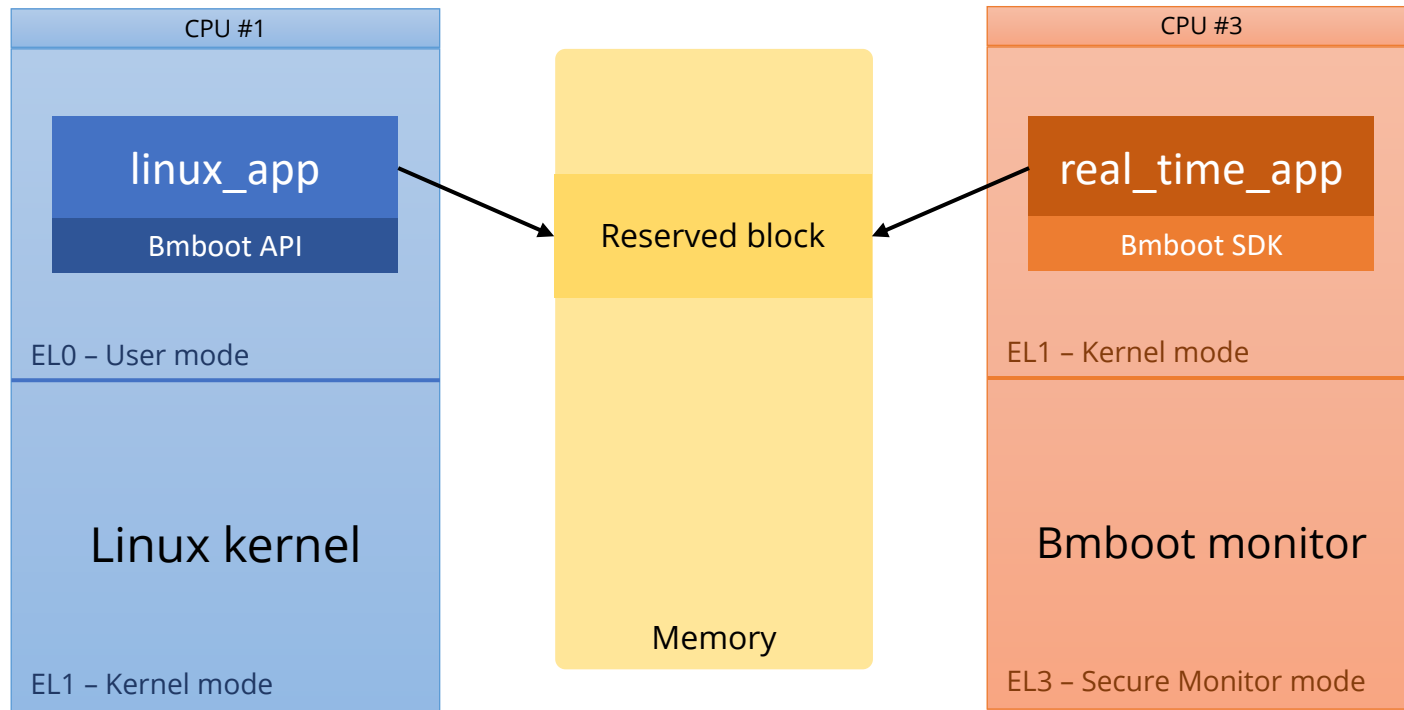
# How does it work?

Lifecycle can also be managed by user application on Linux via API





## Communication via shared memory



- Range of reserved memory determined ahead-of-time
- Cache coherence ensured by hardware Snoop Control Unit (no need to “flush” cache)
- No kernel driver necessary – access via `/dev/mem` special device

## Other features

Crash handling (core dump)

```
root@diot:~# bmctl exec cpu1 real_time_app.bin
```

...

```
root@diot:~# bmctl status cpu1
```

crashed\_payload 🥲

```
root@diot:~# bmctl core cpu1
```

Writing to core.elf

```
root@diot:~# gdb real_time_app.elf core.elf
```

→ Inspect with GDB: registers, stack trace, memory snapshot

Useful also for post-mortems in operation

## Feature summary

- Execution environment with no run-time overhead
- Shared-memory communication
- Interrupt handler registration
  - Periodic *tick* callback (opt-in)
- Crash handling and recovery
- API and CLI for control from Linux

## The bigger picture

- We feel that other groups must be solving a similar problem
- We would be happy to elevate this to a collaborative project
- We would like to hear from you



Thank you!  
Questions?