EmberNet

Real-Time Latency, Measured on Real Hardware

Every EmberRTOS latency result with its hardware, kernel, duration, and load, and what none of them show

Fireball Industries · November 16, 2026 · 10 minute read

Abstract. Anyone can say a box is real time. I would rather show you the numbers and tell you exactly what each one is. This paper collects every real-time latency measurement we have taken on real hardware: a four-core Intel machine, a Raspberry Pi 5 for five minutes and then for twenty-four hours, and an industrial PC with two isolated cores. Each number comes with its hardware, duration, and method, and with the load where we recorded it. The headline: on the Raspberry Pi 5, over 24 hours and about 627 million samples under load, the worst wake-up latency on the isolated cores was 23 microseconds. The sections after the results say what these numbers do not show, because that list is longer than I would like and you should read it before you quote anything here.

1. What a latency number means

Every number in this paper is a wake-up latency measured with a standard real-time latency test run under load. It asks the system to wake a high-priority thread at a fixed interval, then records how late the thread actually ran. The lateness of the worst wake-up in the whole run is the figure that matters for a control loop, because a loop does not fail on its average. It fails on its worst case.

Three things decide what that figure is worth. The first is the kernel: a general-purpose kernel can be delayed by almost anything, while a real-time kernel lets a high-priority thread preempt nearly all of the kernel's own work. The second is isolation: the real-time threads run on cores that have been set aside, and everything else is kept somewhere else. The third is the load: a quiet machine proves little. Each result below says what the machine was doing while it was measured.

EmberRTOS ships its own real-time kernel, built and tuned by us, as the default boot of the Headless RT edition.

2. The results

All results are worst-case wake-up latencies in microseconds (µs). Where a run used several threads, each thread is listed. I give every board its own section and I do not compare boards with each other: a number measured on one board says nothing about another, and the first three results were measured on EmberRTOS builds while the fourth was not.

2.1 Four-core Intel machine, 300 seconds

We booted a tuned test image on commodity four-core Intel hardware, not server grade and not hand-picked. The latency test ran at the highest real-time priority, one thread on each of the three isolated cores (1 to 3), at intervals of 200, 700, and 1200 µs. CPU 0 was the housekeeping core. It carried all movable interrupts and was saturated with a scheduler stress load and a CPU-burn loop for the whole run. Deep C-states were limited, the CPU governor was set to performance, and watchdogs were off. The run lasted 300 seconds.

CORE	SAMPLES	MINIMUM	AVERAGE	WORST CASE
1	1,500,000	1 µs	1 µs	36 µs
2	428,572	1 µs	1 µs	42 µs
3	250,000	1 µs	1 µs	31 µs

The worst wake-up was 42 µs. This is a five minute run on one machine, and rare events live in the hours, so read it as a floor. The 24 hour run in section 2.3 shows how the tail grows on a different board; I do not set the two against each other.

2.2 Raspberry Pi 5, 300 seconds

The same method on a Raspberry Pi 5 booted from a real image, with the real-time kernel, cores 1 to 3 isolated, CPU 0 as housekeeping, the load pinned to CPU 0, interrupts moved to CPU 0, and the power settings fixed for steady timing. The run was 300 seconds with one thread per isolated core.

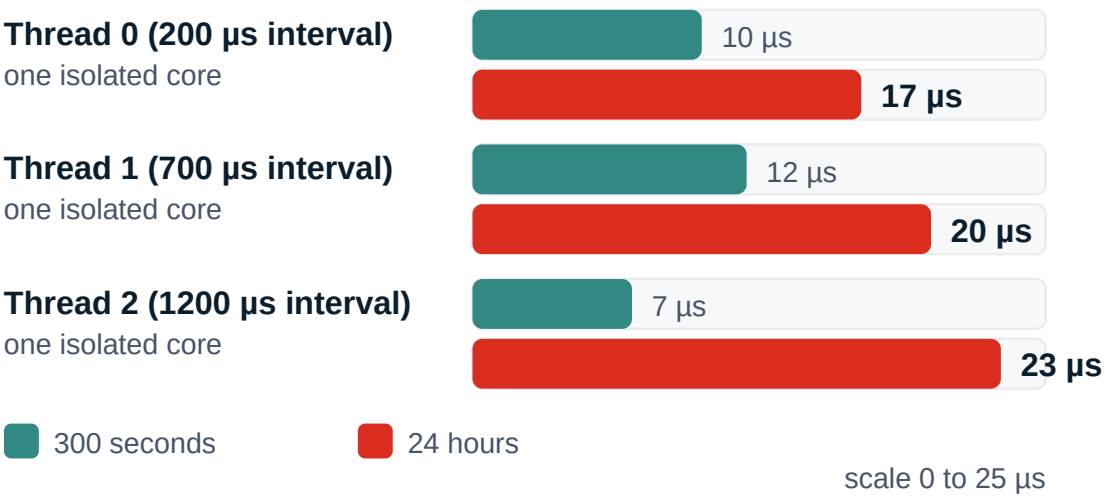
THREAD	INTERVAL	SAMPLES	WORST CASE
0	200 µs	1,500,000	10 µs
1	700 µs	428,572	12 µs
2	1200 µs	250,000	7 µs

The worst wake-up was 12 µs over about 2.18 million samples.

2.3 Raspberry Pi 5, 24 hours

This is the run I trust most, because it is the one that went looking for the rare event, on the same board and image as section 2.2. We ran the same image on the same board for 24 hours (86,423 seconds), with a heavier load: a scheduler stress load, two CPU-burn loops, and a memory streamer in RAM, all pinned to CPU 0 so that the shared memory controller was leaned on the whole time. The run finished clean with exit code zero, and we recorded the temperature.

THREAD	INTERVAL	SAMPLES	WORST CASE
0	200 µs	432,000,000	17 µs
1	700 µs	123,428,572	20 µs
2	1200 µs	72,000,000	23 µs



One Raspberry Pi 5, the same image and method: the worst-case wake-up on each isolated core over 300 seconds and over 24 hours. The tail grew with time, as it should.

About 627 million samples in total, with a worst wake-up of 23 µs. The board started at 30 degrees Celsius, ended at 58, and peaked near 60 under load. We saw no throttling. The tail grew from 12 µs over five minutes to 23 µs over a day, as it should: it is the rare events that take hours to show up, and 23 µs is where the tail settled, not a spike.

2.4 An industrial PC with two isolated cores

On a deployed industrial PC with a hybrid Intel Core i7-14700, we reserved two efficiency cores for a software PLC runtime and its time provider. The kernel was a vendor real-time kernel, which is not an EmberRTOS kernel. I include this result because it shows what core isolation does on a real plant machine, and I label it as what it is. The two cores were isolated at boot, interrupts were kept off them, the clock was held at a fixed frequency, and the time provider ran at a real-time priority on its own core. We measured with the same kind of latency test, one real-time thread on each isolated core, at a 1 ms interval, for 300 seconds. The worst wake-up was 24 µs on one core and 21 µs on the other.

Two cautions come with this row. The raw output of that run was not archived, so the figures come from our engineering record of it. And the load on the machine during the run was not recorded.

Before we isolated those cores, the same kind of machine told us about the problem a different way. In 24 hours, the PLC runtime's own log on one box recorded 138 warnings that its time provider's interval had deviated by 401 to 500 µs (median 448 µs) against a tolerance of 400 µs. That is a log count from the runtime, which is a different instrument from the latency test. The two are not a before and after pair, and I am not presenting them as one.

3. What makes the numbers hold

The pattern behind every result above is the same, and it is not magic.

A real-time kernel. It lets the high-priority thread run in place of nearly everything the kernel is doing.

Cores set aside. One or two cores handle the housekeeping: interrupts, the network, the file system, every other process. The real-time threads get the rest. EmberRTOS ships this as a boot-time layer on the Headless RT edition. On a four-core machine cores 0 and 1 are housekeeping and cores 2 and 3 are kept for real-time work, and the setup app shows which cores are which.

A quiet processor. Deep sleep states are limited so that waking a core does not take a variable amount of time, the governor is set to performance, and watchdogs that wake cores are off.

Nothing else on the isolated cores. This is the one we learned the hard way. On one deployed box we found unrelated work sitting on the cores we had isolated: the kernel workers that encrypt the node's network, the load threads of other runtimes, and network softirq processing. An isolated core only helps if it carries only the workload whose timing needs it, so we now keep everything else off those cores from code at boot, and we check it after every restart.

4. The current kernel

EmberRTOS 0.1.6 ships a real-time kernel, built with our changes on top, for x86-64 and for the Raspberry Pi 4 and 5. Headless RT boots it by default, with the general-purpose kernel beside it in the boot menu.

I have to be plain here: the hardware numbers in this paper come from earlier builds, and I have not yet measured the current kernel on real hardware. I have run the latency test on an earlier build in a virtual machine, and I do not use those figures, because a virtual machine's worst case is set by its host.

5. What this does not show

This section is the reason I trust the rest of the paper, so please read it.

- **No hardware number for the current kernel.** Every hardware result is from an earlier build or, for the industrial PC, a vendor kernel. The current kernel has been booted and checked, and its latency has not been measured on a real board.
- **No virtual machine number is presented as hardware.** Our virtual machine runs are functional checks. Their worst cases are set by the host and say nothing about a real box.

- **The load was kept off the real-time cores.** In the Intel and Raspberry Pi runs the load ran on the housekeeping core. That is what isolation is for, and it shows isolation working, but it is not an adversarial test. A memory-heavy load spread across every core would lean on the memory bus harder, and I have not run it.
- **Only one board has a 24 hour run.** The Intel result is five minutes. Rare tail events live in the hours, so it is a floor, not a ceiling.
- **No Raspberry Pi 4.** We have not had a board on the bench.
- **The industrial PC row has gaps.** The raw output was not kept, the load was not recorded, and the kernel is not ours.
- **The latency test measures wake-up latency.** It does not measure your application's cycle time, the jitter of a fieldbus, the network, or a motion controller. Your loop's budget has to be checked on your own system.
- **This is not a certification and not a safety claim.** The paper reports measurements and nothing more.

6. Check it on your own box

You do not have to take any of this from me. On a Headless RT box, the setup app's Real-time screen shows the kernel and which cores are isolated. The EmberRTOS manual has the commands to check real-time on your own hardware: run the latency test on the isolated cores at a real-time priority, put a load on the housekeeping core, run it for as long as you can afford, and read the worst case, not the average. If your numbers differ from mine, I want to hear about it.

7. Where the numbers came from

These measurements come from EmberNet's engineering records: the EmberRTOS white paper of 14 July 2026 (the four-core Intel run, with its raw test output), the real-hardware results log for the Raspberry Pi 5 (21 and 22 July 2026, transcribed from the board's own console), and our field engineering notes of November 2026 for the industrial PC. The method follows standard core-isolation practice for real-time systems.

Next steps

- Read the manual: EmberRTOS Headless RT
- Download an image: EmberRTOS overview and downloads
- Meet the platform: EmberRTOS on embernet.ai

Patrick Ryan is the CTO of Fireball Industries.