QNX-PulseGuard

Deterministic Microkernel IPC Telemetry & Sub-10ms Deadline Watchdog for **QNX Neutrino 8.0**

Monitors safety-critical automotive and robotics worker processes via non-blocking QNX kernel pulses, profiles microsecond scheduling jitter with zero heap allocations on the hot path, exposes live telemetry over a native `/dev/pulseguard` Resource Manager, and recovers stalled workers in **under 50 microseconds**.

VERIFIED FAULT RECOVERY

29 / 49 us

CPU stall (29 us) & missed pulse (49 us) vs. 10,000 us SLA

HOT-PATH DETERMINISM

0 allocs

Wait-free 4K SPSC ring + 4.1 KB L1 HDR histogram · 0 dropped pulses

C++20 TEST DISCIPLINE

30 / 30

66,253+ assertions verified · -Wall -Wextra -Werror (0 warnings)

DEVELOPER CODELAB

1,776 w

7-module QNX SDP 8.0 tutorial + QEMU x86_64/aarch64 runbook

Dual-Target Build: QNX SDP 8.0 (__QNXNTO__) + POSIX Host Parity · /dev/pulseguard (JSONL + CRC32 devctl)

MIT OPEN SOURCE

Deterministic RTOS IPC & Deadline Watchdog Challenge

1. Tail-Latency Perturbation

Calling `malloc()`, `std::vector::push_back()`, or blocking file I/O on a control thread risks heap lock contention and millisecond-scale scheduling spikes.

`Risk: 10ms actuator loop misses deadline under load`

2. Stale Pulses & False Liveness

When a crashed or hung sensor task restarts, queued heartbeats from the previous execution epoch can trick naive watchdogs into clearing a critical fault.

`Risk: Zombie worker masked by stale IPC queue`

3. Unbounded Priority Inversion

A low-priority background thread holding a shared bus mutex can starve a higher-priority actuator worker (`brake_actuator` prio 22) indefinitely.

`Risk: Delayed safety braking without ceiling boost`

Architectural Shift: Monolithic Polling vs. QNX-PulseGuard Microkernel Design

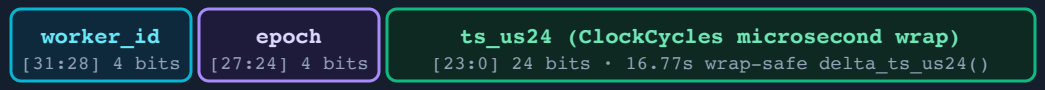
SUBSYSTEM	CONVENTIONAL POLLING / HEAP MONITOR	QNX-PULSEGUARD (QNX NEUTRINO 8.0)
`Heartbeat Transport`	TCP/Unix socket or mutex queue requiring payload copy & heap buffers	**Non-blocking 32-bit kernel pulse** via `MsgSendPulse_r()` on `_NTO_SIDE_CHANNEL`
`Memory Model`	Dynamic allocations + `std::sort()` percentile scans (unbounded WCET)	**0 hot-path allocations**: wait-free 4K SPSC ring + `4.1 KB` O(1) L1 HDR histogram
`Device Interface`	Ad-hoc RPC daemon or log-tailing scripts polling disk	Native user-space `/dev/pulseguard` Resource Manager (`read()` JSONL + `devctl()` CRC32)
`Fault Recovery`	Coarse second-level timers (100–2,000 ms) without epoch fencing	**29 us CPU stall / 49 us missed-pulse** epoch-fenced respawn + PI ceiling 45 escalation

QNX Neutrino 8.0 Native Primitives & `/dev/pulseguard`

Reentrant `_r` Kernel IPC & Hardware Clocking

- `ChannelCreate_r(_NTO_CHF_DISCONNECT | _NTO_CHF_COID_DISCONNECT | _NTO_CHF_UNBLOCK)` creates the watchdog receive channel without mutating thread-local `errno`.
- `ConnectAttach_r(ND_LOCAL_NODE, pid, chid, _NTO_SIDE_CHANNEL, 0)` keeps IPC handles isolated from the POSIX file-descriptor table.
- `MsgSendPulse_r()` queues priority-ordered non-blocking pulses; RX thread blocks safely with `TimerTimeout_r(..., _NTO_TIMEOUT_RECEIVE)` + `MsgReceivePulse_r()`.
- `ClockCycles()` + `SYSPAGE_ENTRY(qtime)->cycles_per_sec` via 128-bit arithmetic (`__uint128_t`) yields sub-microsecond timestamps without 64-bit overflow.
- `SCHED_FIFO` + `PTHREAD_PRIO_INHERIT` / `PTHREAD_PRIO_PROTECT` enforce deterministic thread preemption.

32-BIT `STRUCT _PULSE.VALUE.SIVAL_INT` BIT LAYOUT



Zero payload copies · Priority-ordered kernel delivery · Stale epoch rejection

User-Space `/dev/pulseguard` Resource Manager

Implemented in `src/resmgr_pulseguard.cpp` using standard QNX `dispatch_create()`, `iofunc_func_init()`, and `resmgr_attach()` on a non-RT dispatch thread so device readers never disturb `SCHED_FIFO` workers.

```
resmgr_attach(dpp, &attr, "/dev/pulseguard", _FTYPE_ANY, 0,
              &connect_funcs, &io_funcs, &pg_attr.base);

// 1) Standard POSIX text read: cat /dev/pulseguard
pulseguard_io_read()  -> latches JSONL snapshot via SETIOV()

// 2) Zero-parse C/C++ telemetry: devctl(fd, DCMD_PULSEGUARD_GET_BINARY)
pulseguard_io_devctl() -> 128-byte CRC32 TelemetryBinaryFrame
```

IO_READ / READ64
Double-buffered `dispatch_scratch_` copy with zero allocation under brief lock; supports offset-aware reads.

DCMD_PULSEGUARD_GET_BINARY
Packed 128B struct (`magic 0x504C5347` , percentiles, restarts, inversions, CRC32).

Wait-Free SPSC Ring Buffer & Zero-Alloc Hot Path

DECOUPLED REAL-TIME PIPELINE (ZERO HEAP ALLOCATIONS ACROSS HOT PATH)



Wait-Free SPSC Ring (`ring_buffer.hpp`)

- **Cache-Line Isolation:** Producer `head_` and consumer `tail_` sit on distinct `alignas(64)` boundaries to eliminate cross-core false sharing.
- **Shadow Cursor Caching:** Fast-path checks read local `tail_cache_` / `head_cache_` first, issuing `memory_order_acquire` only when empty/full.
- **Non-Blocking Backpressure:** When full, `try_push()` increments atomic `dropped_` and returns immediately in constant time — never stalls RX.
- **Verified Concurrency:** Stress-tested with multi-wrap concurrent producer/consumer runs (`0` dropped pulses across `1,354` live pulses).

4.1 KB HDR Histogram & Zero-Alloc Proof

- **Fixed Footprint:** 513 `uint64_t` buckets (`4,104 bytes`) reside entirely in L1 data cache.
- **Exact Low-Range + Octave Precision:** `0–63 us` exact `1 us` buckets; `64–1,048,575 us` via 15 log2 octaves × 32 sub-buckets using C++20 `std::bit_width()` (`≤ 3.125%` error).
- **Zero-Alloc Formatting:** Integer `mean_x100` scaling in fixed-buffer `snprintf()` avoids libc `dtoa()` heap allocations.
- **Runtime Guard:** `test_zero_alloc_hot_path` hooks global `operator new` and asserts **0 allocations** on push, histogram, watchdog, and JSON serialization.

Source: include/pulseguard/ring_buffer.hpp · histogram.hpp · telemetry_json.hpp

Sub-50us Fault Injection & Recovery vs. 10ms SLA

EPOCH-FENCED WATCHDOG FINITE-STATE MACHINE (`WATCHDOG.HPP`)



Injected Fault Scenarios (`./build/fault_injector --mode all --stall-ms 14`)

FAULT MODE	WORKER	MECHANISM	RECOVERY / ACTION
`cpu_stall (14 ms)`	#0 lidar_fusion	Detects 6,772 us silence; bumps epoch 1 → 2; discards stale frames	**29 us (0.029 ms)**
`missed_pulse (14 ms)`	#2 imu_ekf	Detects 6,591 us drop; epoch-fenced respawn & handshake	**49 us (0.049 ms)**
`priority_inversion`	#1 brake_actuator	Flags `PRIORITY_INVERSION` on shared bus mutex contention	**Ceiling 22 → 45**

SLA Margin & End-to-End Telemetry Distribution

CPU Stall Respawn: **29 us** — 0.29% of 10,000 us SLA (344× headroom)

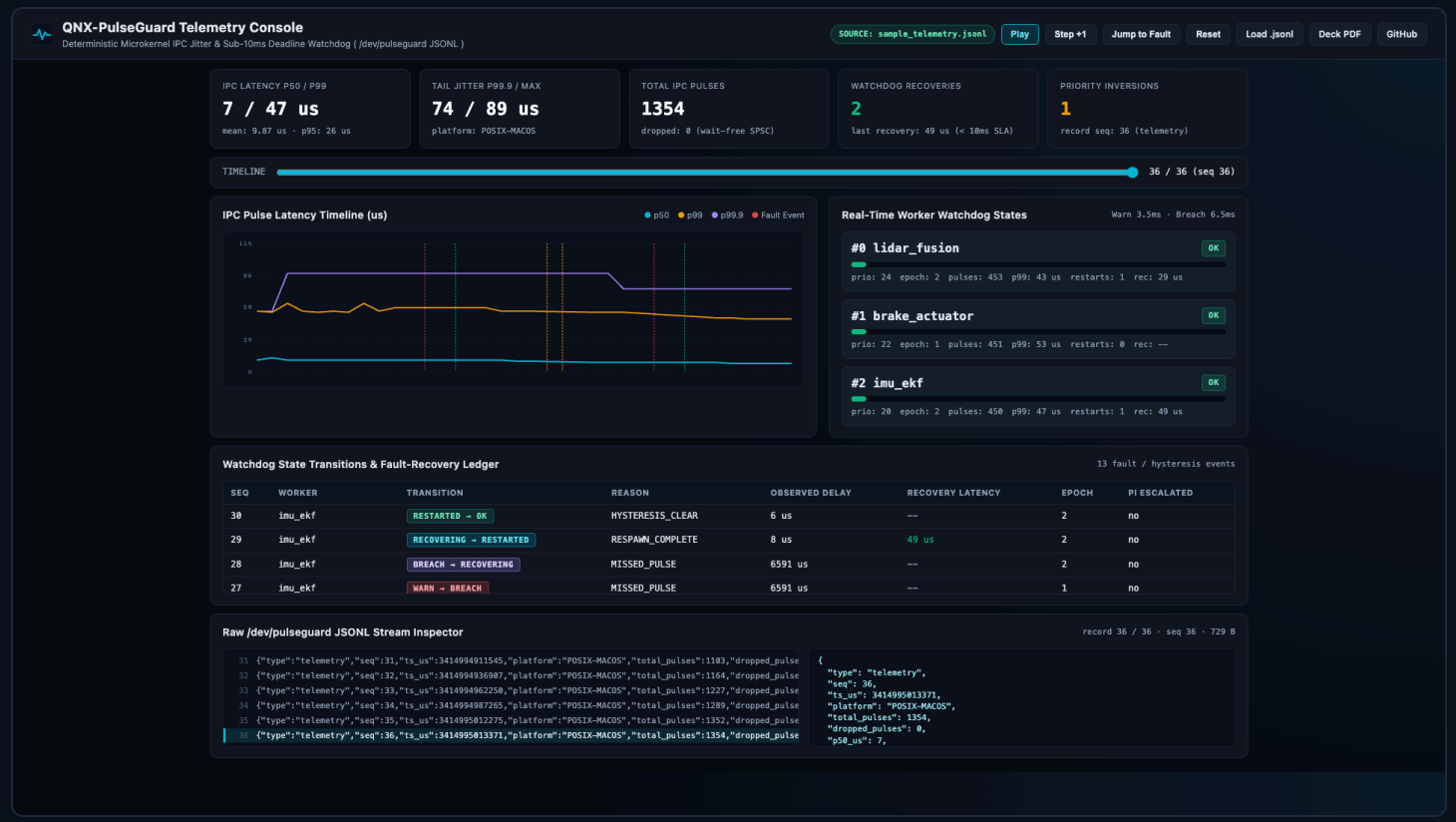
Missed-Pulse Respawn: **49 us** — 0.49% of 10,000 us SLA (204× headroom)

Priority Inversion Guard: Contended `brake_actuator` escalates from prio `22` to ceiling `45` via `PTHREAD_PRIO_INHERIT` ; pre-crash stale epochs are rejected deterministically.

P50 / P95	P99 / P99.9	PULSES / DROPS
7 / 26 us	47 / 74 us	1,354 / 0

Live Web Telemetry Dashboard & JSONL Stream Inspector

alphonshhhomenicmtu.github.io/qnx-pulseguard/ — 0 CONSOLE ERRORS



Interactive Diagnostics

- **Zero Dependencies:** Pure HTML/CSS/SVG in `web/index.html` with inline SVG favicon — loads QEMU or host captures offline or over GitHub Pages.
- **Timeline Replay Controls:** `Play`, `Step +1`, `Jump to Fault`, scrubber slider, and `Load .jsonl` file picker.
- **Percentile Curve Plot:** Live `p50`, `p99`, and `p99.9` curves overlaid with red breach, amber PI, and emerald recovery markers.
- **Worker Gauges & Ledger:** Tracks per-worker epoch, `WARN · PI-CEIL (prio 45)` escalation, and exact recovery microseconds.
- **Raw JSONL Stream Inspector:** Side-by-side `/dev/pulseguard` raw NDJSON lines and formatted packet inspection.

LIVE HOSTED URL
`https://alphonshhhomenicmtu.github.io/qnx-pulseguard/`

Supports live drag-and-drop / upload of QEMU captures (scp root@localhost:/tmp/pg_faults.jsonl)

`CODELAB.md`: 1,776-Word Hands-On Tutorial & QEMU Runbook

MODULE 1

Dual-Target Compilation

Host `make CXX=clang++ test` parity & `q++` cross-builds for `x86_64` / `aarch64`.

MODULE 2

`ClockCycles()` & Bit-Pack

128-bit cycle conversion + packing `worker|epoch|ts_us24` inside 32-bit pulses.

MODULE 3

Reentrant `_r` IPC Calls

`ChannelCreate_r`, `_NTO_SIDE_CHANNEL`, `MsgSendPulse_r`, and `TimerTimeout_r`.

MODULE 4

Wait-Free Ring & HDR

`alignas(64)` SPSC cursors + 513-bucket L1 histogram via `std::bit_width()`.

MODULE 5

`/dev/pulseguard` ResMgr

Mount with `resmgr_attach()`, handle `io_read` JSONL & `devctl()` CRC32 snapshots.

MODULE 6

Watchdog & Fault Lab

Hysteresis, epoch fencing, PI ceiling 45, and running `fault_injector --mode all`.

MODULE 7

Web Console Inspection

Serve `web/index.html` locally or load QNX QEMU JSONL captures directly.

BONUS RUNBOOK

`QNX_QEMU_RUNBOOK.md`

Step-by-step `mkqnximage`, `qemu-system-x86_64`, `scp` deploy & `cat /dev/pulseguard`.

Zero-Hardware QNX 8.0 Reproduction Workflow (45-Minute Developer Path)

```
1. Verify locally:   make CXX=clang++ test && ./build/fault_injector --mode all --stall-ms 14
2. Cross-compile:    source ~/qnx800/qnxsdp-env.sh && make qnx-x86_64   # or make qnx-aarch64
3. Boot QNX QEMU:    mkqnximage --type=qemu --arch=x86_64 --build && qemu-system-x86_64 -smp 2 -m 1024M ...
4. Deploy & verify:  scp -P 2222 build/* root@localhost:/tmp/ && ./pulseguard_daemon & cat /dev/pulseguard
```

Verification Matrix & Open-Source Deliverables

UNIT & E2E TEST SUITE

30 / 30 PASS

66,253+ assertions verified in `tests/test_pulseguard.cpp`: 24-bit clock wrap, SPSC multi-wrap & concurrency, `0` hot-path allocs, HDR percentiles, CRC32 binary frame, simultaneous multi-worker recovery.

COMPILER & BUILD DISCIPLINE

0 Warnings

Strict `-std=c++20 -O2 -Wall -Wextra -Werror` across both `Makefile` and `CMakeLists.txt`, plus QNX SDP 8.0 `q++` targets (`x86_64` / `aarch64`).

SECURITY & HYGIENE AUDIT

0 Leaks

Verified with `gitleaks` (0 secrets detected), 100% English documentation, MIT License, and headless Chrome console/page/network QA (0 errors).

Published Links & Repository Inventory

ARTIFACT	LOCATION / LIVE URL
Public GitHub Repository	https://github.com/alphonshhhomenicmtu/qnx-pulseguard
Live Web Telemetry Console	https://alphonshhhomenicmtu.github.io/qnx-pulseguard/
8-Slide Architecture Deck PDF	https://alphonshhhomenicmtu.github.io/qnx-pulseguard/qnx-pulseguard-deck.pdf
Core C++20 Headers & Sources	include/pulseguard/*.hpp · src/qnx_ipc_bridge.hpp · src/resmgr_pulseguard.cpp · src/pulseguard_daemon.cpp · src/fault_injector.cpp
Tutorial & QNX Runbook	CODELAB.md (1,776 words, 7 modules) · QNX_QEMU_RUNBOOK.md