Real-time Programming Guidelines for C and C++

General, environment-agnostic principles (with examples and common pitfalls).

Generated on February 04, 2026.

This document summarizes key principles for writing real-time software. Real-time means correctness depends on both the produced value and meeting a deadline. The focus is determinism: predictable, bounded behavior, not just speed on average.

1) What "real-time" really means (and what it doesn't)

A real-time system is one where correctness depends on time as well as on producing the correct value.

- A "normal" program is correct if it computes the right output eventually.
- A real-time program is correct only if it computes the right output before a deadline.

Hard vs soft real-time

- Hard real-time: missing the deadline is a system failure (for example, flight control, safety interlocks).
- Soft real-time: missing deadlines occasionally reduces quality (for example, video playback stutters) but is not catastrophic.

Latency, jitter, and throughput

- Latency: the time from "event happens" to "your code responds".
- Jitter: the variation in latency or in periodic timing. If one cycle takes 1 ms and another takes 5 ms, jitter is high.
- Throughput: how much work per second. Throughput can be high even when jitter is terrible.

Real-time cares primarily about latency and jitter, especially the worst case, not the average.

Determinism

Determinism means behavior is predictable: "This step always completes within 50 microseconds." Not: "Usually fast."

Common misunderstanding (pitfall)

A system can have excellent average timing and still be non-real-time because it occasionally hiccups (allocator stalls, page faults, lock contention spikes, I/O stalls). Real-time engineering is largely about removing or bounding these hiccups.

2) Deadlines and WCET (Worst-Case Execution Time)

WCET definition

WCET (Worst-Case Execution Time) is the maximum time a piece of code can take under worst conditions (worst inputs, worst cache state, worst contention, etc.).

Real-time developers try to:

- Design code so WCET is bounded and small.
- Measure and validate WCET under stress.

A key difference vs typical optimization: typical optimization aims to improve average runtime, while real-time optimization aims to reduce worst-case spikes and remove unpredictability.

Time budgets (recommendation)

Think in time budgets. If your loop period is 1 ms, you might allocate:

- 100 microseconds for input acquisition
- 300 microseconds for compute
- 100 microseconds for output
- 200 microseconds for communications
- Leave margin (for example, 300 microseconds) for jitter, interrupts, cache misses, and rare paths

Rare-path spikes (pitfall)

Worst case includes rare code paths like error handling, reconnection, buffer wrap-around, and first-time initialization behavior. Those often cause deadline misses in real systems.

3) Scheduling basics (how the system decides what runs next)

In real-time, you typically have multiple tasks (threads or processes) competing for CPU time. Scheduling decides which one runs at each moment.

Common task types

- Periodic tasks: run every fixed interval (for example, every 1 ms, every 20 ms). Example: a control loop that must execute every 5 ms.
- Event-driven tasks: run when something happens (for example, an input arrives, an interrupt triggers).
- Background or best-effort tasks: can run when CPU is free (logging, UI updates).

Priority

Most real-time systems use priority-based scheduling: higher priority tasks preempt (interrupt) lower priority tasks.

The rule:

- Put "must meet deadlines" tasks at higher priority.
- Keep those tasks short and predictable.

Blocking in high-priority code (pitfall)

A high-priority thread that sometimes blocks on a mutex (lock) is often worse than a slightly lower-priority thread that never blocks. Blocking destroys predictability.

Tiering (recommendation)

Separate hard-deadline code from everything else. A simple model is:

- Tier 1: deadline-critical periodic loop (small, bounded)
- Tier 2: event processing (bounded per cycle)
- Tier 3: I/O, logging, UI, file or network output (best-effort)

4) Time-triggered vs event-triggered designs (two major styles)

Time-triggered (cyclic executive) design

A cyclic executive is a fixed schedule like:

- Every 1 ms: read sensors
- Every 1 ms: compute control output
- Every 10 ms: send telemetry
- Every 100 ms: health check

Benefits: very predictable timing, easy to reason about worst case, great for hard real-time.

Downside: less responsive for sporadic events (unless you schedule frequent checks).

Event-triggered design

Code runs in response to external events: a packet arrives, a digital input changes, an interrupt fires.

Benefits: very responsive; efficient when nothing happens.

Downside: harder to guarantee worst case under heavy event bursts unless you add strict bounds.

Most real systems use a hybrid: critical control is time-triggered; events are captured quickly and handled with bounded work.

Bounded event handling (common pattern)

Even in an event-driven system, you usually add a rule like "process at most N events per loop iteration" so event storms cannot starve deadline-critical work.

5) Interrupts and ISRs (Interrupt Service Routines)

Interrupt and ISR definition

An interrupt is a hardware signal that stops the CPU's current work and runs a special handler. The handler code is the ISR (Interrupt Service Routine).

ISR rules in real-time:

- Do as little as possible in the ISR.
- Capture minimal info (timestamp, a few bytes, set a flag).
- Defer heavier work to a normal task or thread.

Why: ISRs preempt normal code. A long ISR increases latency and jitter for everything else.

Typical pattern: ISR pushes data into a ring buffer (a fixed-size circular queue). A worker task drains the buffer and does processing.

ISR work creep (pitfall)

Doing "just a little more work" in the ISR tends to grow over time (someone adds logging, then parsing, then validation). ISRs become accidental "God functions" and timing collapses.

Conceptual example

ISR: copy 16 bytes into a preallocated ring buffer and increment a write index. Worker thread: parse the message, update a state machine, compute outputs.

6) Priority inversion (a classic real-time failure mode)

What is priority inversion?

Scenario:

- High-priority task H needs a lock (mutex) held by low-priority task L.
- But medium-priority task M runs and keeps preempting L.
- L cannot run to release the lock.
- H waits and misses its deadline.

This is priority inversion: the low-priority task effectively blocks the high-priority one.

How to mitigate

- Keep lock-holding times tiny.
- Use mutexes with priority inheritance: if L blocks H, L temporarily inherits H's priority so it can run and release the lock quickly.
- Prefer designs that minimize shared locks altogether.

Rare still happens (pitfall)

Priority inversion can happen even if you rarely lock, because rare is still possible at exactly the wrong time.

Single-writer ownership (pattern)

A common mitigation is single-writer ownership (one thread owns state; others send messages). That reduces locking and thus reduces priority inversion risk.

7) Memory management: the biggest source of unpredictability

Why malloc/new are dangerous in real-time loops

Dynamic allocation (malloc, new) can:

- Take variable time (allocator searches free lists, coalesces blocks).
- Lock internally (contention).
- Fragment memory over long runtimes (slowdowns appear after hours or days).

Real-time rule: no dynamic allocation in real-time paths.

Practical alternatives

- Preallocate everything at startup (buffers, objects, arrays).
- Fixed-size pools: allocate a pool of N objects, then take/return objects in constant time.
- Static storage: global or static arrays for fixed budgets.

Paging and page faults

On systems with virtual memory, if memory is not currently in RAM, accessing it can trigger a page fault, which may take milliseconds.

Mitigation: touch memory during initialization so it is loaded; lock memory if supported; avoid lazy initialization in real-time threads.

C++ hidden allocation pitfalls

Very common C++ pitfalls:

- std::string may allocate (even if it sometimes does not due to small string optimization).
- std::vector reallocates unless you reserve or fix capacity.
- std::function may allocate depending on what you store in it.
- Logging frameworks often allocate and/or take locks.
- First call to something may allocate (lazy init, internal caches, locale setup).

Practical rule: treat anything that might allocate as unsafe in deadline-critical code unless you can prove otherwise and lock it down.

8) Data structures and algorithms: bounded beats clever

Real-time requires predictable upper bounds.

Avoid these in real-time paths

- Containers that sometimes resize or rehash: std::vector without reserved capacity; std::unordered_map (rehash spikes).
- Anything with potentially unbounded runtime: regex parsing; recursion with uncertain depth; "while queue not empty process all" when queue can grow without limit.

Prefer these

- Fixed-size arrays or fixed-capacity vectors.
- Ring buffers.
- Pre-sized std::vector with reserve or built fixed capacity.
- Deterministic state machines.

Bounded per cycle processing (key technique)

- In each cycle, process at most N items.
- If more arrive, defer to next cycle or drop (depending on requirement).

This prevents event storms from exploding execution time.

Examples of boundedness

- Parse at most N packets per cycle.
- Run at most N iterations of a solver per cycle (and carry state forward).
- Coalesce events: if 500 position-update messages arrive, keep only the latest for this cycle.

Big-O is not enough (pitfall)

Even bounded big-O can still be too large if constants are high or memory access is random (cache-unfriendly).

9) Concurrency: avoid shared mutable state when possible

Why shared state is risky

Shared state requires synchronization: locks (mutexes) or atomic operations.

Locks can block unpredictably and cause priority inversion. Atomics avoid blocking but can be tricky and still cause contention.

Safer architecture: ownership and message passing

- One thread owns a piece of data and is the only writer.
- Other threads communicate via messages (queues).
- Readers get snapshots (copies) or use double buffers.

This makes timing more predictable and simplifies reasoning.

Common patterns

- SPSC ring buffer (Single Producer, Single Consumer): one writer thread, one reader thread; efficient and predictable.
- Double-buffering: one buffer is active for readers while the writer fills the other; then swap pointers or indices.
- Snapshot model: periodically copy a small state struct that readers use without locking.

Tiny locks grow (pitfall)

A "tiny lock" becomes expensive if taken frequently, or if contention occurs at the wrong priority boundary.

10) Logging and I/O: keep it out of real-time threads

I/O (disk, network, console) can block unpredictably. Even formatting strings can allocate memory or take variable time.

Best practice: RT thread writes minimal binary records into a ring buffer; a background thread drains it and writes logs/telemetry.

Recommendation

If you must log from real-time code, log only counters, fixed-size numeric fields, and timestamps, and avoid dynamic formatting.

Async still bursts (pitfall)

Non-blocking sockets and async logging can still hide bursts: buffers fill and something eventually blocks or drops. That is acceptable if deliberate and bounded, not accidental.

11) Cache, branch prediction, and rare spikes

Even if your algorithm is bounded, hardware effects can produce jitter.

Cache effects

If data or code is not in CPU cache, access is slower and worst-case latency can spike.

You reduce this by keeping hot code/data small and contiguous, and minimizing pointer-chasing linked structures.

Branch prediction

CPUs try to guess which way a branch goes. Unpredictable branching can cause pipeline stalls.

In real-time, you care less about fastest and more about no surprises.

Additional pitfalls and tips

- Avoid scattered allocations (linked lists, trees) in hot paths: pointer chasing causes cache misses.
- Watch for false sharing: two threads write different variables that happen to sit on the same CPU cache line; performance becomes jittery under load.
- Keep hot data structures aligned and compact.

12) State machines: the real-time developer's best friend

A state machine is a design where your system is in one of a finite number of states, and transitions happen only through defined events.

Benefits: predictable control flow, easier worst-case timing analysis, fewer unexpected behaviors. This is extremely common in embedded and real-time control.

Example

- States: Idle, Armed, Running, Fault
- Events: StartCmd, StopCmd, SensorOk, SensorFail, Timeout
- Each event handler performs bounded work and updates state deterministically.

Pitfall

Ad-hoc if spaghetti grows until you cannot reason about timing or correctness. State machines keep complexity controlled.

13) Testing and measurement: proving your timing

You cannot confidently claim real-time behavior without measurement.

Real-time teams track:

- Maximum observed cycle time
- Jitter
- Deadline misses
- Queue depths and overflow counts

Then they stress the system: maximum input rates, worst-case scenarios, long-duration runs (to catch fragmentation, drift, leaks).

Recommendations

- Instrument start/end timestamps around every major stage in your loop.
- Track max observed per stage (not just average).
- Keep overrun counters and dropped event counters and treat them as first-class health metrics.

Measurement distortion (pitfall)

If your measurement/logging is heavy, it can distort timing. Common pattern: collect lightweight counters in RT code and export them periodically from a lower-priority context.

14) Practical C/C++ rules of thumb for real-time code

Coding rules

- No dynamic allocation in real-time loops.
- No blocking I/O in real-time threads.
- No unbounded loops ("process everything until empty") in RT paths.
- Prefer fixed-size buffers and bounded queues.
- Keep ISR minimal; defer work to threads.
- Keep critical sections tiny; avoid nested locks.
- Prefer explicit error handling over exceptions in RT paths.

Optimization goals

- Reduce worst-case runtime.
- Reduce variance (jitter).
- Keep memory access predictable.
- Avoid hidden work (allocations, resizing, flushing, system calls).

C/C++ pitfalls

- Hidden copies: passing large structs by value; unintended std::string copies.
- Hidden temporaries: operator overloading and expression templates can surprise you.
- Convenient abstractions that allocate or lock under the hood.
- Debug-only checks that are fine until someone compiles with them enabled in production.
- Floating-point corner cases: denormals/subnormals (extremely tiny numbers) can drastically slow some computations on some hardware; many systems flush them to zero to avoid timing spikes.

15) A bigger point: real-time is a system property

Even perfect C++ can fail deadlines if the environment introduces spikes (CPU frequency scaling, background services, scheduler behavior, driver latencies, interrupt storms).

So real-time engineering is always: architecture + code + OS configuration + measurement.

Recommendation

Even if you are environment-agnostic, it helps to design assuming: preemption can occur at any time; interrupts can burst; memory access can be slower than expected; and some subsystem can stall unpredictably. Structure your code so stalls happen only in noncritical layers.

16) A concrete mental model (example)

Imagine a 1 kHz control loop (every 1 millisecond):

- Read inputs (must finish within 100 microseconds).
- Compute control (must finish within 300 microseconds).
- Write outputs (must finish within 100 microseconds).
- Housekeeping (optional, bounded, maybe 200 microseconds).

Total worst-case must remain below 1 millisecond with margin. Anything unbounded (allocations, logging flush, processing "all queued messages") risks blowing the budget.

What margin is for

- Cache misses
- Interrupts
- Occasional slow branches

- Synchronization overhead
- Rare but valid states (fault paths, reconnection logic)

A good real-time design keeps margin intentionally, not accidentally.

Added: Concrete mini-examples and patterns (C/C++ oriented)

A) Ring buffer pattern (fixed capacity, predictable)

A ring buffer is a fixed-size queue where indices wrap around. Typical use: a producer (ISR or I/O thread) pushes fixed-size messages; a consumer (worker thread) pops and processes.

Why it is real-time friendly: no allocation; bounded operations (constant-time); explicit overflow behavior (TryPush returns false).

Pitfall: multi-producer/multi-consumer queues are much harder to implement correctly and predictably. If you need them, keep the design simple (for example, one producer per queue and merge later).

Conceptual C++ sketch (Single Producer + Single Consumer) with abundant inline comments

```
#include <array>
#include <atomic>
#include <cstddef>

/*
    RingBufferSpsc<T, Capacity>

    SPSC = Single Producer, Single Consumer.

    Intent:
    - Provide a fixed-capacity (no allocations) queue.
    - Exactly one producer thread calls TryPush().
    - Exactly one consumer thread calls TryPop().

    Why this is real-time friendly:
    - No malloc/new during operation.
    - Each push/pop is constant-time (bounded).
    - Overflow/underflow is explicit via return value (no blocking).
*/

template<typename T, std::size_t Capacity>
class RingBufferSpsc
{
public:
    // TryPush:
    // - Returns false if the buffer is full (caller decides what to do: drop, count,
    // backpressure, etc.)
    // - Returns true if the item was successfully queued.
    bool TryPush(const T& item)
    {
        // Read the current write index (where the producer will write next).
        // memory_order_relaxed is fine for *this* load because the producer is the only
        // writer of m_write.
```

```
        const std::size_t write = m_write.load(std::memory_order_relaxed);

        // Compute the next write index (wrap around at Capacity).
        const std::size_t next = Increment(write);

        // Full condition:
        // If advancing write would catch up to read, the ring is full.
        //
        // We use memory_order_acquire on the read index to ensure we observe the
        // consumer's
        // latest read position before deciding the buffer is full.
        if (next == m_read.load(std::memory_order_acquire))
        {
            return false; // Buffer full: caller must handle overflow policy.
        }

        // Store the item into the slot pointed to by "write".
        // This is a plain assignment into preallocated storage (no allocation).
        m_data[write] = item;

        // Publish the new write index.
        // memory_order_release ensures that the data write to m_data[] becomes visible to
        // the consumer
        // before the consumer observes the updated m_write.
        m_write.store(next, std::memory_order_release);

        return true;
    }

    // TryPop:
    // - Returns false if the buffer is empty.
    // - Returns true and writes the popped item to "out" if available.
    bool TryPop(T& out)
    {
        // Read the current read index (where the consumer will read next).
        // memory_order_relaxed is fine for *this* load because the consumer is the only
        // writer of m_read.
        const std::size_t read = m_read.load(std::memory_order_relaxed);

        // Empty condition:
        // If read equals write, there is nothing to pop.
        //
        // memory_order_acquire pairs with the producer's memory_order_release store to
        // m_write,
        // ensuring that if we observe an updated m_write, we also observe the
        // corresponding m_data[] write.
        if (read == m_write.load(std::memory_order_acquire))
        {
            return false; // Buffer empty.
        }

        // Copy the item out of the ring slot.
        // This should be a bounded operation; keep T small or trivially copyable if you
        // want tight timing.
        out = m_data[read];

        // Advance the read index (consumes this slot).
        // memory_order_release publishes the updated read index to the producer.
        m_read.store(Increment(read), std::memory_order_release);
```

```
        return true;
    }

private:
    // Increment:
    // - Wraps an index around Capacity.
    // - constexpr makes this foldable by the compiler for fixed Capacity.
    static constexpr std::size_t Increment(std::size_t idx)
    {
        return (idx + 1) % Capacity;
    }

    // Preallocated storage:
    // - Fixed size at compile time.
    // - No heap usage; avoids fragmentation and allocator latency.
    std::array<T, Capacity> m_data{};

    // Producer-owned index:
    // - Only the producer thread writes m_write.
    // - Consumer reads it to test empty / pop.
    std::atomic<std::size_t> m_write{0};

    // Consumer-owned index:
    // - Only the consumer thread writes m_read.
    // - Producer reads it to test full / push.
    std::atomic<std::size_t> m_read{0};
};
```

B) Fixed-size object pool (avoid malloc/new in RT path)

Pattern: allocate N objects at startup; acquire/release from a free list. Benefits: bounded allocation time; no fragmentation.

Pitfall: watch for concurrency. You may need per-thread pools or lock-free free lists; otherwise you reintroduce lock contention.

C) Double-buffering (predictable reads without locking)

Pattern: writer fills back buffer, then swaps a pointer/index so readers see the new front buffer. Readers always read a stable snapshot.

Good for sharing a current state struct (sensor snapshot, control parameters).

Pitfall: if state is large, copying every cycle may be too expensive; then use smaller snapshots or partial updates.

D) Bounded per cycle event processing

Pseudo-logic: in each call, process at most N events; exit early if queue is empty. This guarantees an upper bound on the work performed per cycle.

Conceptual C++ example with abundant inline comments

```
// Bounded per-cycle event processing
//
// Intent:
// - Prevent an "event storm" from consuming unbounded time in one cycle.
```

```
// - Guarantee an upper bound on work per cycle (max_events_per_cycle).
//
// Typical usage:
// - Run this once per periodic tick (e.g., every 1 ms, 10 ms, etc.)
// - Or run in an event loop that must remain responsive/predictable.

template<typename QueueT, typename EventT>
void ProcessEventsBounded(QueueT& queue)
{
    // Upper bound on events processed in this call.
    // Choose this based on your time budget and the worst-case cost of HandleEvent().
    constexpr int max_events_per_cycle = 50;

    for (int i = 0; i < max_events_per_cycle; ++i)
    {
        // Create a local event object. Keep it fixed-size to avoid allocations.
        EventT ev{};

        // Try to pop one event.
        // If the queue is empty, exit early.
        if (!queue.TryPop(ev))
        {
            break;
        }

        // HandleEvent must be designed to be bounded:
        // - no allocations
        // - no blocking I/O
        // - no unbounded loops
        //
        // If processing can be expensive, split it into smaller steps and carry state
        // across cycles.
        HandleEvent(ev);
    }

    // If events arrive faster than you process them, the queue may fill.
    // That is not "mysterious": you must define a policy (drop oldest, drop newest,
    // coalesce, etc.)
    // and track counters (drops, max queue depth, overruns) for observability.
}
```

Pitfall: if events arrive faster than you handle them, the queue fills. That is not a bug. You must define an overflow policy (drop newest, drop oldest, coalesce, compress) and track it with counters/telemetry.

Added: Common failure modes (pitfalls) checklist

These are classic real-time problems that show up in code reviews:

- Unbounded loops: "process all pending messages" without a cap.
- Hidden allocations: string formatting, logging, containers resizing.
- Blocking calls: I/O, sleeps, waiting on condition variables in high priority code.
- Lock contention and priority inversion: high-priority thread waits on a mutex held by low-priority thread.
- Work creeping into ISRs: parsing and logging inside interrupt handlers.
- Rare-path spikes: reconnect logic, error handling, first-use lazy init.

- Poor cache behavior: pointer chasing, huge working set, false sharing.
- No backpressure plan: queues fill, latency explodes, or system thrashes.
- No measurement: "seems fine" until it fails under load.

Added: Real-time-friendly design habits that pay off

- Make overflow explicit (return false, increment a drop counter).
- Structure code as stages with budgets (input, compute, output, optional).
- Use state machines for mode logic.
- Keep data local to a thread and communicate via messages.
- Pre-initialize everything (memory, tables, caches) before entering RT mode.
- Prefer simple, predictable code to fancy abstractions in RT paths.
- Treat "rare" as "will happen at the worst moment."