White Paper

# Deterministic Real-Time Execution on Windows Systems

## Common Misconceptions and Their Consequences


### Executive Summary

The term "hard real-time" is widely used in industrial computing, yet frequently reduced to measurements of latency or system load. In many cases, statistical observations—such as average latency, jitter, or benchmark results—are presented as evidence of real-time capability. These metrics describe performance, but they do not establish deterministic behavior.

For systems in which timing violations constitute failure, such reasoning is insufficient. Hard real-time capability is not a function of speed or optimization, but of architectural control over all time-critical execution paths. This paper examines common misconceptions that arise when statistical reasoning is applied to deterministic requirements and derives the architectural implications necessary to meet strict timing guarantees on Windows-based systems.


### 1. Low Jitter Does Not Establish Determinism

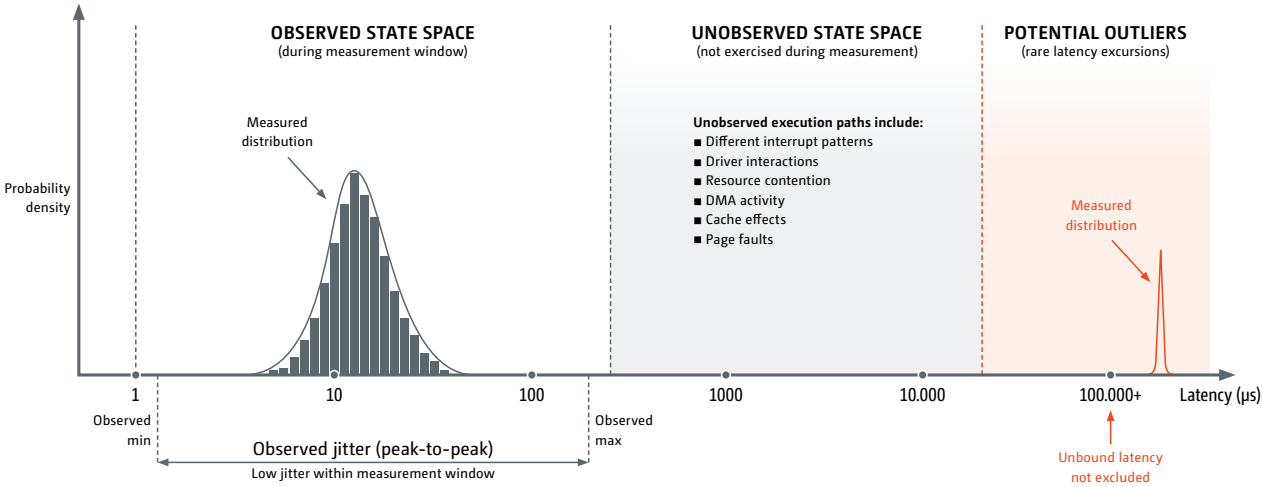
Jitter is commonly interpreted as an indicator of real-time capability. It describes the variation between observed minimum and maximum latency within a given measurement interval. While low jitter may indicate stable performance under specific conditions, it remains a statistical property.

The key limitation is that jitter depends on both observation time and system state coverage. A measurement captures only a subset of possible execution paths. If certain system states—such as specific interrupt patterns, driver interactions, or resource contention scenarios—do not occur during the observation window, their timing effects remain unmeasured.

From a systems perspective, the number of possible states grows combinatorially with the interaction of hardware events, scheduling decisions, and concurrent activities. Exhaustive observation is therefore infeasible. As a result, a low observed jitter does not exclude the existence of significantly higher latency outside the measured range.

Deterministic systems are defined not by narrow statistical distributions, but by the absence of unbounded variation. Jitter can describe variability, but it cannot establish that variability is bounded.

**Fig. 1: Low Jitter Does Not Establish Determinism**
Measured latency reflects only a subset of possible system states.




**Key point:** Low observed jitter indicates stable behavior within tested conditions.
It does not prove that higher latency cannot occur under other (unobserved) system states.

## 2. Measured Maxima Are Not Worst-Case Bounds

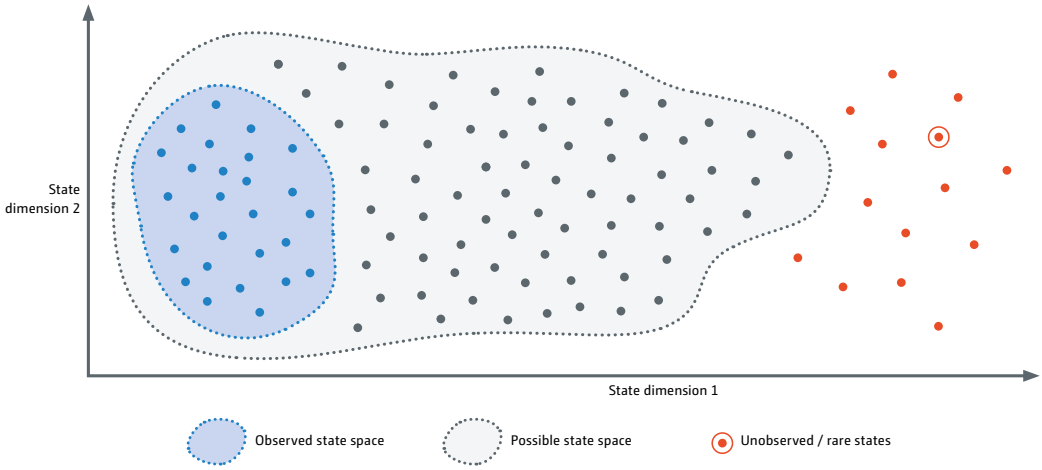
A measured maximum latency represents the worst observed case under specific test conditions. It does not constitute a true worst-case bound unless it can be demonstrated that no higher latency is possible.

Establishing a worst-case bound requires that all timing-relevant mechanisms are both known and controlled. This includes scheduling decisions, interrupt handling, memory access behavior, and interactions with hardware and drivers. If any of these components can introduce delays that are not strictly bounded, the system's timing behavior remains open-ended.

In general-purpose operating systems, many of these mechanisms are influenced by asynchronous events and shared resources. The resulting state space is effectively unbounded, and exhaustive exploration is not possible. Measurement, regardless of duration, samples only a subset of this space.

A worst observed case reflects the absence of worse behavior during testing. A true worst case reflects a proven upper bound derived from architectural constraints.

**Fig. 2: Observed vs possible state space**




## 3. Synthetic Load Does Not Represent Real Systems

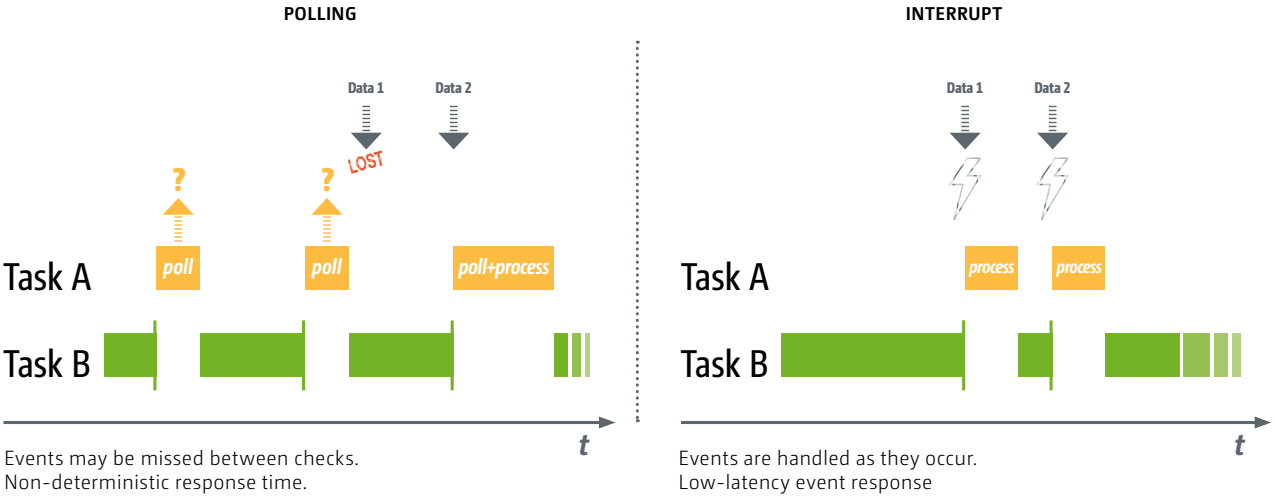
Synthetic load tests are often used to validate real-time behavior by applying sustained computational pressure. These tests typically rely on predictable, CPU-bound workloads such as tight loops or burn-in programs. While they can achieve high utilization, they fail to reproduce the temporal structure of real systems.

In practice, system timing is dominated by asynchronous events. Hardware interrupts occur unpredictably, drivers introduce variable execution paths, and direct memory access enables concurrent data movement outside CPU control. These mechanisms interact with the scheduler and memory subsystem in ways that are not captured by deterministic loop-based workloads.

Interrupt handling further amplifies this effect. Interrupts trigger deferred processing, influence cache behavior, and may preempt execution at non-deterministic points. The resulting latency depends on system state and event ordering, not merely on CPU load.

Synthetic tests therefore exclude key sources of non-determinism and cannot demonstrate real-time behavior under realistic conditions.

**Fig. 3: Loop-based vs interrupt-driven execution comparison**



### 4. Priority Does Not Imply Control

A commonly cited mechanism for achieving real-time behavior under Windows is the use of elevated scheduling priority, particularly REALTIME_PRIORITY_CLASS. This class allows processes to run above all standard user-mode priorities and can reduce interference from other applications. However, it operates entirely within the user-mode scheduling model and does not change the system's fundamental behavior. Threads in REALTIME_PRIORITY_CLASS remain subject to kernel-mode execution. Interrupt Service Routines and Deferred Procedure Calls can preempt them at any time, independent of their priority. These mechanisms are driven by hardware and drivers and are not controllable from user space.

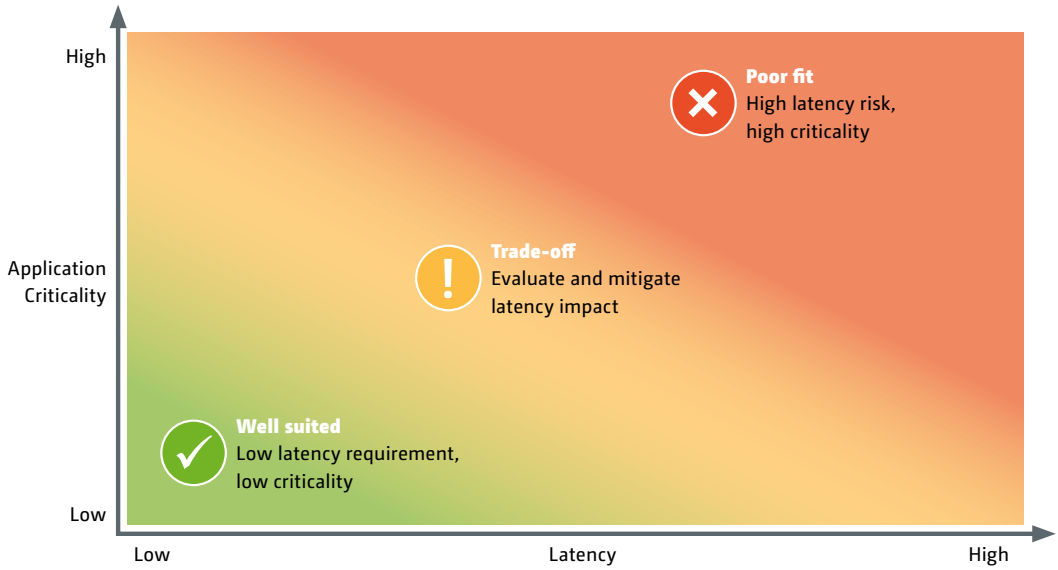
Memory management introduces further uncertainty. Page faults, kernel operations, and cache effects can delay execution in ways that cannot be fully bounded from a user-mode context. The Windows scheduler itself is designed for responsiveness, not determinism. It does not enforce strict upper bounds on latency. As a result, REALTIME_PRIORITY_CLASS improves typical latency but does not eliminate non-deterministic influences. It provides preference, not control, and therefore cannot establish hard real-time guarantees.

### 5. Numerical Latency Values Are Context-Dependent

Latency values are often interpreted in isolation, as though a sufficiently small number were evidence of real-time capability. In practice, their significance depends entirely on system context and failure semantics.

In safety-critical applications, acceptable timing behavior is determined not by average performance, but by the consequences of a missed deadline. A latency that is acceptable for multimedia processing or user interaction may constitute failure in motion control or industrial automation. The same numerical value can therefore represent either negligible delay or unacceptable risk, depending on the application.

**Fig. 4: Latency vs application criticality mapping**



This distinction is especially important in certified environments. Regulatory requirements evaluate whether temporal behavior remains bounded under all admissible conditions, not whether the system performs well under typical load. A system with low average latency but unbounded worst-case behavior may therefore fail deterministic requirements despite excellent benchmark results.

Latency values describe observed behavior under specific conditions. By themselves, they provide no information about guarantees, boundedness, or failure handling.

### 6. Hard Real-Time Is Not Achieved Through Optimization

Optimization improves throughput, reduces average execution time, and can significantly decrease observed latency. These improvements may increase practical usability under load, but they do not remove the underlying sources of non-determinism present in a general-purpose operating system.

Interrupt timing, scheduler behavior, memory management activity, and driver interactions remain asynchronous and only partially controllable. Optimization may reduce the probability of timing violations, but it does not establish that such violations are impossible.

Highly optimized systems may appear stable during testing and produce statistically impressive results. Deterministic behavior, however, is not defined by how rarely a deadline is missed, but by whether the architecture guarantees bounded behavior under all valid operating conditions.

Hard real-time capability therefore cannot be achieved through tuning alone. It requires architectural control over timing-relevant mechanisms, including interrupt handling, scheduling behavior, and resource ownership. Optimization can refine a deterministic system, but it cannot make a non-deterministic architecture deterministic.

### 7. Stability Over Time Does Not Imply Guarantees

Long periods of stable operation are often interpreted as evidence that a system is effectively deterministic. In practice, such observations provide only empirical confidence based on previously encountered operating conditions.

Complex systems contain many possible execution states arising from hardware events, software activity, scheduling decisions, and external inputs. Certain timing violations may therefore occur only under rare state combinations that are unlikely to appear during conventional testing, regardless of test duration. Extended uptime without observable failure does not demonstrate bounded latency behavior. It demonstrates only that critical states have not yet occurred. Deterministic systems rely on architectural bounds rather than statistical observation.

## Architectural Implications

The preceding misconceptions share a common assumption: that deterministic behavior can be inferred from observation rather than enforced through architectural control. This assumption may be sufficient for per-

formance-oriented systems, but not for systems in which timing violations constitute failure.

In general-purpose operating systems such as Windows, execution timing is influenced by kernel-level mechanisms including interrupt dispatching, Deferred Procedure Calls, scheduler decisions, memory management activity, and driver behavior. As long as time-critical execution remains subject to these mechanisms without full control over their timing characteristics, deterministic guarantees cannot be established.

For this reason, hard real-time capability cannot be achieved solely through elevated process priority, benchmarking, or optimization. Deterministic behavior requires architectural separation between time-critical and non-critical execution domains. Time-critical tasks must operate in an environment where interrupt handling, scheduling behavior, and resource access are controlled and bounded, while non-deterministic operating system functionality can execute independently.

This approach preserves the Windows ecosystem while enabling deterministic behavior for time-critical workloads. Determinism is not derived from favorable statistical behavior within Windows itself, but from architectural control over the execution environment.

## What Hard Real-Time Actually Implies

A system is hard real-time capable if exceeding a defined timing constraint constitutes a system failure. To satisfy this requirement, all timing-relevant execution paths must remain bounded under all admissible operating conditions. This includes not only application code, but also scheduling decisions, interrupt handling, memory access behavior, and interactions with hardware and drivers. If any of these mechanisms can introduce unbounded delay, deterministic guarantees cannot be established.

Hard real-time systems therefore require more than low latency or high computational performance. They require deterministic behavior, explicit upper time limits, and defined responses to timing violations. Statistical observations such as average latency, jitter measurements, or long stable runtimes may indicate good performance, but they do not substitute for bounded execution guarantees.

Hard real-time is therefore defined not by how fast a system operates under normal conditions, but by whether its timing behavior remains controlled under all relevant conditions.

## Conclusion

Hard real-time begins where statistical reasoning ends. Systems that rely on measurement can demonstrate performance, but not guarantees. Deterministic behavior requires architectural control over all timing-relevant mechanisms. A system that measures its worst case assumes it has seen it.

Kithara

**Imprint**
Kithara Software GmbH, Alte Jakobstr. 78, 10179 Berlin, Germany, Phone: +49 30 2789673-0, Fax: +49 30 2789673-20
Email: info@kithara.com (general inquiries), support@kithara.com (technical inquiries), Internet: www.kithara.com
Copyright © 2026 Kithara Software GmbH. All rights reserved. No liability for printing errors. Subject to alterations.
Copying or reproducing, in whole or in part, is permitted only with approval of Kithara.
All named trademarks or registered trademarks are the properties of their respective owners. May, 2026