EAI Endorsed Transactions
on Sustainable Manufacturing and Renewable Energy Research Article EAI.EU

Review of Azure RTOS ThreadX Real-Time Operating System on STM32 Platforms

P. H. Truong[1,2,*], H. Q. T. Ngo [1,2] and V. D. T Tran [3,4]

[1] Ho Chi Minh City University of Technology (HCMUT), 268 Ly Thuong Kiet street, Dien Hong Ward, HCMC, Vietnam
[2] Vietnam National University-Ho Chi Minh City (VNU-HCM), Linh Xuan Ward, HCMC, Vietnam
[3] Faculty of Biology and Biotechnology, University of Science, Vietnam National University Ho Chi Minh City, Vietnam
[4] Vital-IT, SIB Swiss Institute of Bioinformatics, Lausanne, Switzerland

Abstract

Real-time operating systems (RTOS) play a critical role in modern embedded systems by providing deterministic execution, predictable timing behavior, and efficient use of limited hardware resources. These properties are especially important in industrial automation, energy control, and safety-oriented applications, where reliable timing behavior is required. This article focuses on Azure RTOS ThreadX, a lightweight RTOS, and examines its core real-time mechanisms, including scheduling, inter-thread communication and synchronization, and memory management. Particular attention is given to the integration of ThreadX with STM32 microcontrollers based on ARM Cortex-M architectures, which are widely adopted in industrial and energy-related embedded platforms. Therefore, a structured literature survey is conducted using peer-reviewed publications, official technical documentation, and industrial reports to analyze and compare ThreadX with other embedded RTOS solutions. The analysis indicates that, on typical STM32 platforms using ARM Cortex-M cores, ThreadX achieves context-switch latencies in the low microsecond range while maintaining a small kernel footprint suitable for resource-constrained devices. These characteristics enable reliable and predictable operation in industrial and safety-critical embedded systems, while also highlighting challenges related to scalability, mixed-criticality workloads, and energy-aware scheduling that motivate future research.

Keywords: Real-time operating systems, Azure RTOS ThreadX, STM32 microcontrollers, deterministic scheduling, inter-thread communication, thread synchronization, memory management, ARM Cortex-M.

Received on 27 January 2026, accepted on 14 April 2026, published on 05 May 2026

doi: 10.4108/eetsmre.11710

1. Introduction

RTOS play a critical role in embedded systems that require deterministic timing behavior and predictable execution [1]. RTOS selection significantly influences constrained environments, where missed deadlines can compromise functionality or safety [2].

Among available RTOS solutions, Azure ThreadX stands out for its lightweight, highly deterministic design. Originally developed by Express Logic and now maintained under the Eclipse ThreadX project, ThreadX offers priority-based preemptive scheduling and bounded latency that support robust embedded applications. ThreadX has been recognized as an industrially relevant RTOS and is often mentioned alongside other widely used embedded systems in research surveys of real-time system performance [3].

RTOS research has shown that real-time benchmarks are key metrics for evaluating embedded operating systems. In particular, studies indicate that basic scheduling operations in some open-source RTOSs can differ by factors of two or more in cycle counts, directly affecting task responsiveness and system jitter [4].

*Corresponding author. Email: han.truongphuc@hcmut.edu.vn

Parallel to RTOS advances, STM32 microcontrollers based on ARM Cortex-M cores have become a dominant hardware platform in both academic and industrial contexts due to their balance of performance, low power consumption, and peripheral versatility [5], [6]. Research on STM32-based embedded systems emphasizes real-time processing capabilities and efficient microcontroller usage in IoT and control systems, demonstrating that STM32 devices can achieve reliable real-time performance with appropriate software stacks [7].

The integration of ThreadX with STM32 platforms enables reproducible configuration of scheduling policies, synchronization primitives, and memory management mechanisms across device families. This integration not only simplifies development workflows but also preserves deterministic behavior, which is essential for performance and safety considerations in industrial and sustainable systems [8].

Unlike generic RTOS surveys, this work provides a platform-aware analysis of Azure ThreadX on STM32 microcontrollers, with emphasis on how kernel mechanisms translate into configuration-driven determinism and practical integration considerations. From this perspective, the RTOS comparison is framed by first explaining the underlying mechanisms, rather than being a direct side-by-side comparison. Overall, by focusing on the interaction between ThreadX and ARM Cortex-M-based STM32 platforms, the review addresses real-time requirements commonly encountered in industrial, smart grid, and *energy-related embedded applications* [3], [9].

2. Technical review

2.1. ThreadX kernel architecture

ThreadX is an advanced RTOS featuring a unique picokernel architecture, designed specifically for deeply embedded, real-time, and IoT applications. With over 12 billion deployments worldwide, ThreadX offers deterministic performance, minimal footprint (2KB ROM, 1KB RAM minimum), and extensive safety certifications (IEC 61508 SIL 4, UL 60730, EAL4+) [9], [10].

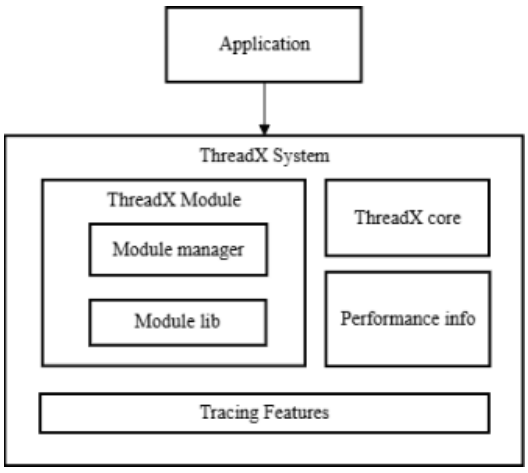

Figure 1. The high-level architecture of ThreadX

Figure 1 illustrates the architectural organization of a ThreadX-based system, structured around a compact kernel core and a set of auxiliary components that support extensibility and observability. At the top of the architecture, the *Application* layer represents user-defined tasks that interact with the underlying real-time system through standardized APIs, ensuring portability and separation between application logic and kernel internals [11].

The *ThreadX core* forms the deterministic execution engine of the system. It implements essential real-time services such as thread scheduling, context switching, synchronization primitives, and basic memory management [9]. Only time-critical mechanisms are embedded within the kernel, while multi-layered service hierarchies are avoided [11], [12].

As shown in Figure 1, ThreadX supports extensibility through a separate *ThreadX Module* subsystem, consisting of a *module manager* and a *module library*. This subsystem enables modular application development without increasing kernel complexity. By isolating module-related operations from the kernel core, ThreadX preserves deterministic scheduling behavior while supporting scalable software architectures [13].

In addition, *performance information* and *tracing features* are implemented as auxiliary components rather than kernel-resident services. These facilities provide runtime observability and diagnostic capabilities while avoiding interference with the kernel's critical execution paths [11].

Overall, the architecture depicted in Figure 1 reflects a balance between minimalism and functionality. This architectural approach has contributed to the widespread adoption of ThreadX, reported to be deployed in over 12 billion devices worldwide, and to its attainment of high-level certifications such as IEC 61508 SIL 4, ISO 26262 ASIL D, and Common Criteria EAL4+, underscoring its suitability for safety-critical embedded systems [3].

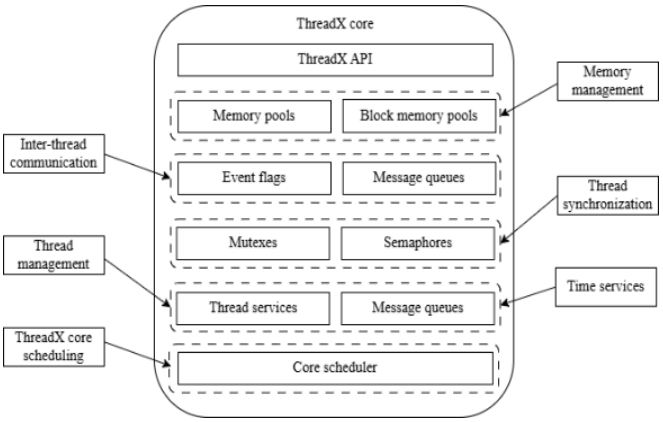

Figure 2. Main architecture diagram of some mechanisms in ThreadX core component

Figure 2 illustrates the main mechanisms implemented within the ThreadX core, it is composed of a tightly integrated set of real-time services exposed through the ThreadX API. At its foundation, the *core scheduler* governs thread execution and preemption, while *thread services* and *application timers* support task lifecycle management and time-driven operations. Synchronization and inter-thread coordination are provided through *mutexes, semaphores, event flags, and message queues*, enabling safe and deterministic communication among concurrent threads. Memory allocation within the core is handled via *byte-oriented memory pools and fixed-size block memory pools*, offering predictable and efficient memory management for embedded systems. Together, these components constitute the functional core of ThreadX and form the basis for its scheduling, synchronization, and memory management mechanisms, which are analyzed in detail in the following sections [13].

2.2. Scheduling mechanisms of ThreadX system

Scheduling is a core mechanism of the ThreadX operating system. In ThreadX, scheduling is based on a fixed-priority preemptive model, where the highest-priority ready thread is always selected for execution. This model ensures that critical tasks preempt less critical ones immediately, a requirement commonly adopted in embedded RTOS designs for predictable response times.

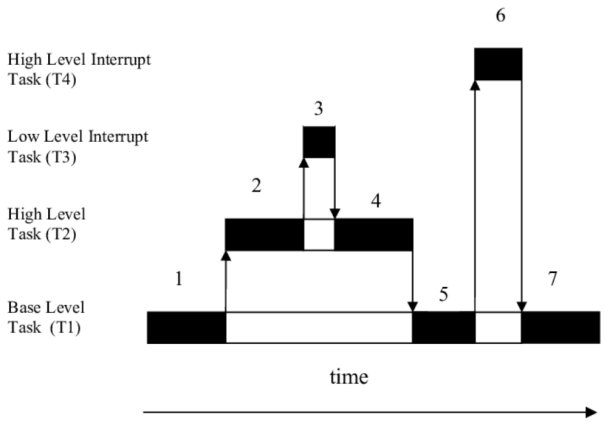

Figure 3. Generic preemptive scheduling scheme in RTOS [14]

ThreadX also features optional time slicing among threads with equal priority, allowing equitable CPU sharing when needed. A notable enhancement is the preemption-threshold mechanism, which extends fixed-priority scheduling by restricting preemption until a higher-priority thread exceeds a configurable threshold [15]. Research has shown that this technique can reduce unnecessary context switches and improve schedulability in priority-driven systems, particularly under moderate to high workloads.
While this mechanism reduces the frequency of preemptive events, each preemption still incurs a context switch, making context-switch efficiency a critical factor in real-time performance. Table 1 therefore summarizes the measured context-switch latency of ThreadX on representative ARM Cortex-M platforms. The data presented in this table are synthesized from reported measurements in prior studies evaluating ThreadX on ARM Cortex-M microcontrollers, including vendor documentation and experimental results in [9].

Table 1. Measured Context Switch Times of ThreadX on ARM Cortex-M microcontrollers

Processor	Clock Speed	Context Switch Time
ARM Cortex-M0+	48 MHz	~2.5 µs
ARM Cortex-M4	120 MHz	~1.5 µs
ARM Cortex-M7	216 MHz	~1.1 µs
ARM Cortex-M7 (with FPU)	216 MHz	~1.8 µs

Formally, if we denote set of threads $T = \{T_1, T_2, \ldots, T_n\}$ with static prorities p_i (higher value indicates higher priority) and corresponding preemption thresholds θ_i, the scheduler selects a ready thread T_j at time t such that

$$T_j = \arg\max_{\substack{T_k \in \mathrm{R}(t) \\ p_k > \theta_{\text{current}}}} p_k$$

where $\mathrm{R}(t)$ is the set of ready threads at time t and θ_{current} is the preemption threshold of the thread currently executing. This formulation illustrates that only threads whose priority exceeds the current thread's threshold can preempt it, thereby enforcing tighter runtime preemption control compared to a purely fixed-priority preemptive scheduler [16].

2.3. Inter-thread communication and synchronization mechanisms of ThreadX system

Other mechanisms of ThreadX system is thread communication and synchronization. Specifcally, synchronization primitives such as *semaphores and mutexes* enforce mutual exclusion and event ordering, while inter-thread communication primitives like *message queues and event flags* enable structured data transfer and event signaling without shared memory hazards [10], [17].

Thread synchronization

In ThreadX, *counting semaphores and mutexes* form the core of thread synchronization. A counting semaphore maintains a non-negative integer count S, which is atomically decremented upon acquisition and incremented upon release; if $S = 0$, a requesting thread blocks until the semaphore is signaled [15]. Otherwise, mutexes extend this model by enforcing ownership and addressing priority inversion: when a high-priority thread T_h is blocked on a

lock held by a lower-priority thread T_l, the RTOS can temporarily boost the effective priority of T_l to that of T_h, bounding the blocking time to the duration of the critical section [15]. A common analytic expression for this mechanism

$$p_{\text{effective}} = \min\left(p(T_l), p(T_h)\right)$$

where lower numeric values indicate higher priorities [18]. Such priority inheritance protocols are essential to meet worst-case latency guarantees infixed-priority scheduling. To clarify how RTOS synchronization primitives coordinate concurrent threads, Figure 4 illustrates the operation of semaphores and mutexes as used in ThreadX. While the terminology follows common RTOS concepts, the diagram reflects how these mechanisms behave in practice in ThreadX, including thread waiting and signaling, resource access control, and the use of priority inheritance in mutexes to avoid priority inversion.

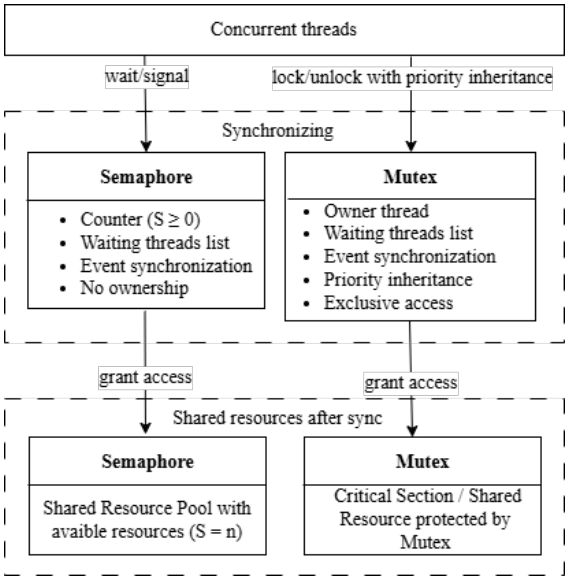

Figure 4. Diagram of RTOS thread synchronization mechanisms as used in Azure RTOS ThreadX

Inter-thread communication

For *thread communication*, ThreadX supports message queues and event flags. Message queues implement FIFO communication with fixed-size slots, allowing producer-consumer patterns where a sending thread can block when the queue is full and a receiving thread can block when it is empty. This structured data transfer avoids the hazards of shared memory and provides timing predictability [17]. Event flags group up to 32 Boolean signals that can be polled or waited on using logical AND/OR combinations, enabling threads to wake only when specified composite conditions are met. Both mechanisms separate control and data flow, reducing contention and enhancing responsiveness in real-time pipelines [18]. To illustrate how RTOS communication primitives coordinate concurrent threads, Figure 5 presents the operation of message queues and event flags as used in Azure RTOS ThreadX. While the terminology follows common RTOS concepts, the diagram reflects how these mechanisms behave in practice in ThreadX, distinguishing between data exchange through message queues and event-based signaling through event flags.

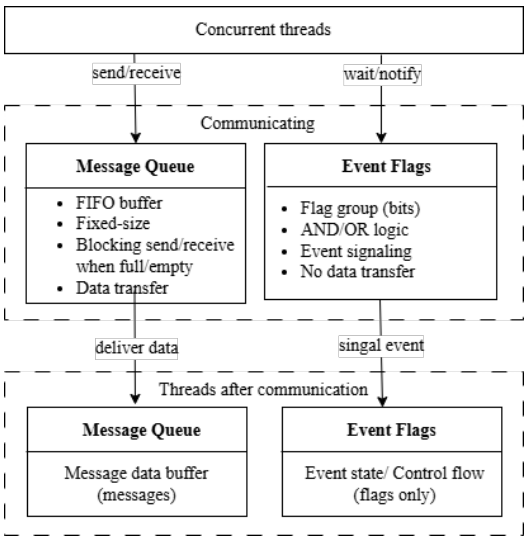

Figure 5. Diagram of RTOS thread communication mechanisms as used in Azure RTOS ThreadX

Together, these inter-thread communication and synchronization primitives in ThreadX support predictable coordination, bounded blocking, and efficient data propagation, consistent with established RTOS design principles for real-time embedded systems.

2.4. Memory manangement mechanisms of ThreadX system

Memory management in ThreadX is designed to provide bounded allocation behavior through pool-based mechanisms, including two primary mechanisms: *byte pools* for variable-size dynamic allocation and *block pools* for fixed-size allocation. Both are implemented as pre-allocated contiguous regions, allowing the application to avoid direct use of *non-deterministic malloc/free-style* allocators in time-critical paths [19], [20].

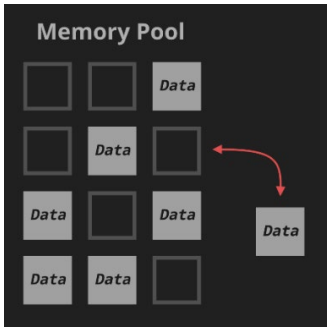

Figure 6. Memory pool structure illustrating pre-allocated fixed-size blocks and pool allocation strategy

Byte pools

A *byte pool* supports variable-size dynamic memory allocation using a first-fit strategy. Memory is allocated from free segments within the pool and may be split as needed to satisfy allocation requests [21]. While this approach offers flexibility and efficient memory utilization, allocation latency depends on the internal fragmentation state of the pool, and external fragmentation may prevent allocation even when sufficient total memory is available [22].

So that, from a temporal analysis perspective, the worst-case allocation time of a byte pool depends on the number of free fragments that must be traversed during the first-fit search [23]. If the pool contains N_f free segments, the worst-case allocation complexity can be expressed as

$$T_{alloc}^{byte} = O\left(N_f\right)$$

where N_f increases as fragmentation accumulates over time. Consequently, allocation latency becomes history-dependent and cannot be tightly bounded, limiting the suitability of byte pools for hard real-time execution paths. Otherwise, the impact of fragmentation in a byte pool can be illustrated by a simple scenario. Consider a 400-byte pool initially free. If an application allocates two blocks of 128 bytes each and later frees only the first block, the pool ends up with a free-used-free pattern (128 B free, 128 B used, 144 B free). Although the total free space is 272 bytes, a subsequent request for a contiguous 200-byte block will fail because no single free segment is large enough [22].

More explanation, let F_i denote the size of the i-th free contiguous segment in a byte pool. An allocation request of size S succeeds if and only if $\max_i\left(F_i\right) \geq S$ even when the total available free memory satisfies [23]

$$\sum_i F_i \geq S$$

In the illustrated example, although the total free memory is 272 bytes, the largest contiguous free segment is only 144 bytes. Therefore, an allocation request of 200 bytes fails despite sufficient total free space.

Block pools

In contrast, a *block pool* divides a memory region into fixed-size blocks at initialization time. Each allocation returns exactly one block, and deallocation simply returns the block to the free list. Because all blocks have identical size, external fragmentation is eliminated, and both allocation and deallocation execute in constant time, O(1) [24]. These characteristics make block pools highly predictable and well suited for real-time workloads. For block pools, allocation and deallocation operations involve only constant-time manipulation of a free-block list. The worst-case execution times can therefore be expressed as [23]

$$T_{alloc}^{block} = T_{free}^{block} = O(1)$$

independent of the number of blocks or the allocation history. This property makes block pools well suited for worst-case execution-time analysis and real-time schedulability verification.

Comparison between byte pools and block pools

Based on the byte pool and block pool mechanisms described above, their fundamental differences in allocation behavior and temporal predictability can be illustrated as follows

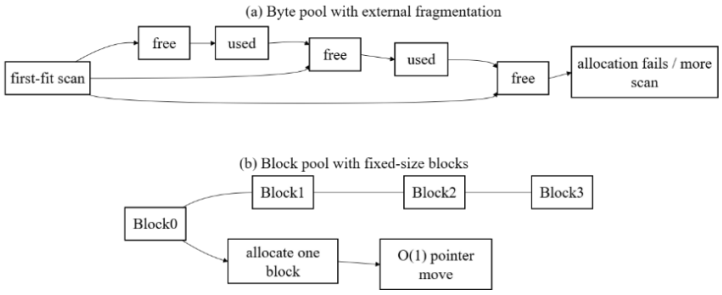

Figure 7. Comparison of memory allocation mechanisms in ThreadX with (a) byte pool allocation using a first-fit strategy with external fragmentation and history-dependent scanning and (b) block pool allocation using fixed-size blocks with constant-time (O(1)) pointer-based allocation

Therefore, due to their deterministic behavior, block pools are typically used for time-critical objects such as communication buffers, control structures, and frequently allocated data with known size. Byte pools, by comparison, are more appropriate for non-critical tasks, initialization routines, or infrequent allocations where flexibility outweighs strict timing guarantees.

From that, the key differences between byte pools and block pools in terms of predictability and real-time suitability can be summarized in Table 2.

Table 2. Comparison between byte pools and block pools in ThreadX

Aspect	Byte pool	Block pool
Allocation granularity	Variable-size blocks	Fixed-size blocks
Allocation strategy	First-fit search	Free-block list
Worst-case allocation time	*O(Nf)*, history-dependent	*O(1)*, deterministic

External fragmentation	Possible	Eliminated
Memory utilization	Flexible, higher packing efficiency	Less flexible, size-constrained
Suitability	Non-critical allocations	Time-critical real-time paths

Overall, the combination of byte pools and block pools in ThreadX provides a balanced memory management strategy that spans flexibility and determinism [9]. When applied with appropriate design discipline, these mechanisms enable efficient dynamic memory usage while maintaining the predictability required in real-time embedded systems [19].

2.5. ThreadX integration with STM32 platforms

STM32 is a family of 32-bit microcontrollers developed by STMicroelectronics based on ARM Cortex-M processor cores. These devices combine high performance, low power consumption, and rich integrated peripherals, making them suitable for a wide range of embedded and real-time applications. STM32 microcontrollers support multiple series with different core variants such as Cortex-M0, M3, M4, and M7, allowing scalability across performance and power budgets. The architectural flexibility and abundant communication/timer peripherals make STM32 platforms widely adopted in IoT, industrial automation, and control systems requiring deterministic and efficient processing [5].

The integration of Azure ThreadX with STM32 microcontroller platforms forms a cohesive hardware-software stack that combines deterministic real-time execution with the peripheral-rich and energy-efficient characteristics of ARM Cortex-M devices [25].

This integration is officially supported within the STM32Cube ecosystem, which provides configuration tools, middleware, and hardware abstraction layers to simplify RTOS adoption while preserving real-time predictability [26]. ThreadX is incorporated into STM32 development workflows through STM32CubeMX [5], where kernel parameters, thread priorities, timers, synchronization objects, and memory pools can be configured at design time. The generated project structure ensures consistent RTOS initialization, clock configuration, and interrupt priority setup across STM32 families [5], [6].

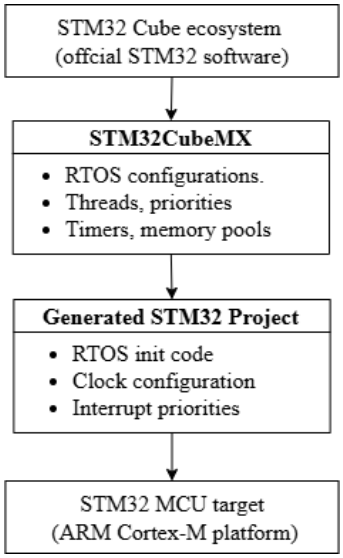

Figure 8. Design-time integration workflow of Azure ThreadX within the STM32Cube ecosystem, illustrating RTOS configuration via STM32CubeMX.

So that, while STM32CubeMX defines the RTOS configuration at design time, the ThreadX kernel is responsible for enforcing scheduling, timing, and synchronization semantics at run time [25].

Beyond the configuration and project-generation stage, the effective operation of ThreadX on STM32 platforms therefore depends on how the RTOS kernel is bound to the underlying hardware mechanisms [5].

At the kernel level, ThreadX interfaces with STM32 hardware via CMSIS-compliant APIs and the STM32 HAL [26]. System timing is typically driven by the SysTick timer or equivalent hardware timers, while the NVIC is configured to maintain compatibility with ThreadX scheduling and critical-section requirements. This tight integration enables predictable interrupt latency and bounded context-switch overhead.

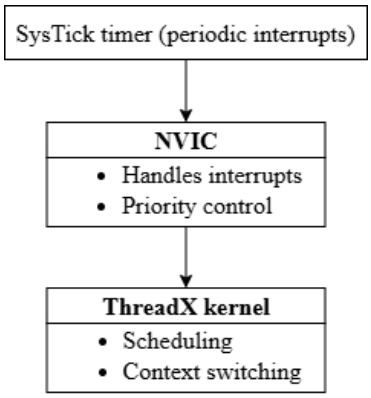

Figure 9. Interaction between SysTick, NVIC, and the ThreadX kernel on ARM Cortex-M (STM32)

From an application perspective, the ThreadX-STM32 combination is widely used in industrial control, smart sensors, communication gateways, and energy-related embedded systems. STM32 peripherals such as timers, and communication interfaces can be managed within ThreadX threads or interrupt service routines, supporting concurrent execution with deterministic behavior.

3. Survey results & Discussion

3.1. Survey methodology

This review is based on a structured survey of peer-reviewed publications, official technical documentation, and publicly available industrial benchmark reports related to embedded RTOSs on ARM Cortex-M platforms. Azure ThreadX, FreeRTOS, and Zephyr were selected due to their widespread adoption and the availability of reported performance data on comparable microcontroller architectures.

The comparison focuses on scheduling behavior, synchronization mechanisms, memory management strategies, and integration complexity rather than on reproducing benchmarks under identical test conditions. Reported performance metrics are synthesized from heterogeneous sources and are used to highlight relative trends and design trade-offs, rather than to claim absolute performance superiority.

Because the benchmark data originate from different platforms and configurations, the comparisons emphasize qualitative and relative insights. Where applicable, the sources of data are explicitly indicated in the accompanying text and figures.

3.2. Comparative analysis of ThreadX with other embedded RTOS platforms

To evaluate the practical real-time implications of different RTOS design choices, this section presents a comparative analysis of representative architectural aspects across Azure ThreadX, FreeRTOS, and Zephyr. The comparison is not a simple side-by-side listing; instead, it is discussed after the key mechanisms have been introduced and explained above, so that the differences reflect practical behavior rather than isolated metrics. Notably, the benchmark results discussed here are taken from different studies and platforms. As a result, they are used to highlight general trends and trade-offs, rather than to provide a strict, like-for-like performance comparison.

Scheduling models with kernel service latency

Based on the scheduling mechanism detailed in Section 2.2, Azure ThreadX reduces unnecessary context switches via preemption thresholds [9]. However, industry performance reports indicate measurable differences among ThreadX, FreeRTOS, and Zephyr in core scheduling operations such as context switching and interrupt handling [3]. On ARM Cortex-M platforms, FreeRTOS demonstrates lower interrupt and context-switch latency than Zephyr, with reported averages of approximately 101 CPU cycles for both interrupt latency and cooperative context switching, compared to about 143 cycles and 468 cycles for Zephyr under comparable conditions [27]. These results suggest that FreeRTOS's lean kernel design yields lower scheduler overhead in low-level timing.

By contrast, Zephyr prioritizes configurability and modularity, including support for additional system services and SMP on multicore architectures. This architectural flexibility introduces higher runtime overhead, leading to increased interrupt and context-switch latency relative to minimal RTOS kernels [28].

From these observations, Figure 10 summarizes scheduling-related kernel execution costs across Zephyr, FreeRTOS, and Azure ThreadX, focusing on context switching and interrupt handling. The results indicate lower scheduling overhead in FreeRTOS, higher latency in Zephyr, and intermediate yet stable performance in Azure ThreadX. The data presented in Figure 10 are taken from the benchmark results reported in [27].

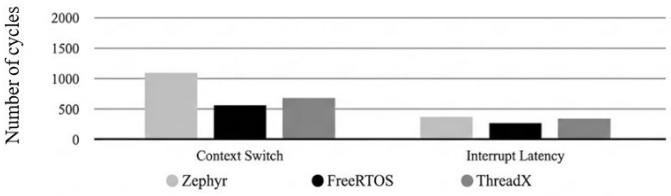

Figure 10. Average execution times of scheduling-related kernel services, including context switching and interrupt latency, measured over 1000 iterations

Synchronization mechanisms

Following the synchronization mechanisms defined in Section 2.3, ThreadX bounds blocking times for high-priority tasks through kernel-level priority inheritance [9], [17]. Zephyr's optimized mutex implementation achieved a mutex block latency of approximately 969 CPU cycles, compared with around 1964 cycles for FreeRTOS under similar conditions, indicating that optimized mutex and scheduling paths in Zephyr can outperform FreeRTOS in specific synchronization operations [27]. Meanwhile, FreeRTOS demonstrated slightly faster unblocking behavior in these tests (about 1212 cycles versus 1309 cycles for Zephyr) [28]. These measured latencies underscore that synchronization performance is not strictly a function of kernel size but depends on interaction with the scheduler.

Although comprehensive benchmarking directly including ThreadX synchronization timing is not widely published in open literature, the design of ThreadX's synchronization primitives seeks to minimize priority inversion and bound worst-case blocking times in contexts where timing assurance is paramount.

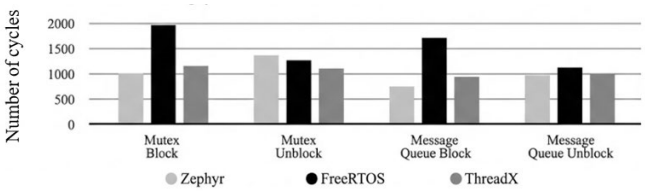

Figure 11. Average execution times of synchronization and inter-thread communication primitives, including mutex and message queue block and unblock operations, measured over 1000 iterations

Overall, Figure 11 illustrates synchronization and inter-thread communication latency, showing faster blocking paths in Zephyr, quicker unblocking in FreeRTOS, and stable intermediate performance in Azure ThreadX. The data presented in Figure 11 are taken from the benchmark results reported in [27].

Memory management strategies

As discussed in Section 2.4, the pool-based memory management in ThreadX enables more predictable allocation behavior, which is particularly beneficial for time-critical execution paths.

FreeRTOS provides multiple dynamic heap management schemes (heap_1 through heap_5), offering varying trade-offs among memory utilization, coalescing behavior, and fragmentation control [26]. However, heap-based allocation introduces variability in execution time that complicates strict worst-case timing analysis, especially in long-running systems with dynamic memory use [23].

Zephyr similarly supports heap and fixed-size slab allocations, where slab mechanisms approximate the constant-time properties of fixed-block pools; in practice, Zephyr's memory management overhead remains higher due to additional subsystems and configuration layers intrinsic to its modular kernel design [19]. When comparing pure allocation costs, independent community measurements suggest that more configurable and feature-rich RTOS kernels tend to incur greater overhead in memory services than minimal RTOS options, although precise cycle-accurate figures are highly dependent on application workload and platform architecture [2].

Integration complexity & Ecosystem support

Integration complexity and tooling support significantly affect time-to-market and maintainability. ThreadX, as part of the Azure RTOS suite, integrates with vendor tooling such as STM32CubeMX to streamline configuration and reduce integration errors that can compromise deterministic behavior [29]. FreeRTOS's simplicity and extensive board and toolchain support make it broadly applicable to resource-constrained MCUs, but developers must manually configure priorities and interrupt contexts to ensure real-time properties. Zephyr provides a rich ecosystem with device trees, networking, and connectivity frameworks, but this breadth introduces steeper integration and configuration overhead compared to more minimal RTOS options [2], [27].

Table 3. Compact qualitative comparison of Azure ThreadX, FreeRTOS, and Zephyr RTOS

Aspect	Azure ThreadX	FreeRTOS	Zephyr
Scheduling	Preemptive & preemption-threshold; stable, bounded behavior	Lean preemptive; lowest latency	Modular scheduler; higher latency
Synchronize	Priority inheritance; bounded blocking	Priority inheritance; faster unblock	Optimized mutex/IPC; faster block
Memory	Fixed-block & byte pools; high determinism	Heap-based; variable timing	Heap & slab; moderate determinism
Integration	Strong vendor tooling (STM32CubeMX, CMSIS)	Broad MCU/toolchain support	Rich ecosystem; higher complexity
Focus	Predictability	Minimal overhead	Feature richness

3.3. Implications for embedded applications

Prior studies in the real-time and embedded systems literature consistently emphasize that RTOS architectural choices have direct implications on system predictability, energy efficiency, and certification feasibility in industrial and IoT deployments beyond raw latency measurements, scheduling models and synchronization design.

Real time predictability and system reliability

The priority-based scheduling mechanism in Thread X described in Section 2.2 contributes to reduced scheduling overhead and more stable timing behavior compared with purely preemptive kernels. By limiting unnecessary preemptions and bounding blocking times, ThreadX reduces timing variability and improves stability under moderate to high system load [30].

This emphasis on bounded execution behavior facilitates more reliable worst-case execution time analysis and simplifies system validation. In contrast to highly extensible RTOS designs that introduce multiple abstraction layers and configuration-dependent execution paths [31], ThreadX favors a controlled execution model that minimizes timing uncertainty. As a result, systems built on ThreadX can achieve higher reliability with less reliance on extensive manual tuning and validation effort.

Energy efficiency considerations

In battery-powered and sustainable embedded systems, energy efficiency depends strongly on predictable execution and efficient processor usage. Azure ThreadX improves energy efficiency by reducing unnecessary processor activity through its scheduling model and deterministic synchronization mechanisms [11]. By

limiting excessive context switching and bounding blocking behavior, ThreadX reduces CPU active time and allows the processor to remain in idle or sleep states for longer periods, leading to lower energy consumption [19]. ThreadX's pool-based memory management further supports long-term energy efficiency. Fixed-size block pools eliminate external fragmentation and guarantee constant-time memory allocation, preventing unpredictable memory-related overhead that could otherwise increase processor utilization over time [9]. By maintaining stable execution behavior throughout the system lifetime, ThreadX demonstrates that predictable execution and controlled resource management are essential for sustaining energy efficiency in long-running IoT and industrial embedded deployments [22].

Suitability for safety-critical and mixed-criticality systems

Safety-critical and mixed-criticality systems demand strong guarantees on temporal isolation, fault containment, and verification feasibility. ThreadX is well suited to these environments due to its deterministic scheduling behavior, kernel-level priority inheritance, and bounded memory management services. These characteristics reduce timing interference between tasks of different criticality levels and simplify the demonstration of timing correctness [9].

In mixed-criticality scenarios, where safety-critical and non-critical workloads coexist, ThreadX's predictable scheduling and controlled synchronization behavior help maintain isolation without excessive configuration complexity. Compared to feature-rich RTOS platforms that offer extensive connectivity and modular services at the cost of increased system complexity, ThreadX provides a more verification-friendly foundation. This balance makes ThreadX particularly appropriate for safety-oriented applications where certification effort, long-term reliability, and predictable system behavior are paramount [30].

3.4. Limitations & Challenges

Although Azure ThreadX offers strong determinism and predictable real-time behavior, its design also introduces limitations related to scheduling scalability, memory allocation trade-offs, and support for emerging embedded system demands.

Scalability and flexibility limitations

One key limitation of ThreadX is its reliance on a static-priority scheduling model. Fixed-priority scheduling with a preemption-threshold mechanism offers strong timing predictability, but it becomes less flexible as system size increases [9]. In systems with many threads, assigning priorities correctly requires careful offline analysis, and small configuration errors can significantly affect schedulability. In large-scale embedded systems, static-priority scheduling has been reported to require extensive offline analysis to avoid schedulability loss [30].

For applications with highly dynamic behavior-such as variable-rate data processing, adaptive control, or sporadic task activation-static-priority scheduling may require frequent redesign or manual tuning. Unlike adaptive scheduling approaches, ThreadX does not dynamically adjust priorities at runtime, which limits its flexibility in systems where workloads change over time. As a result, ThreadX is better suited to systems with well-defined and relatively stable execution patterns.

Scalability challenges also appear in memory management when using fixed-size block pools [20]. Block pools provide deterministic allocation behavior, but they require memory sizes to be defined at design time. In large systems with diverse data structures, this can lead to inefficient memory usage or the need for multiple block pools.

Table 4. Scalability trade-offs of static-priority scheduling and fixed-size block pools in ThreadX

Aspect	Benefit	Limitation
Static-priority scheduling	Strong predictability, bounded latency	Limited adaptability, complex priority assignment in large systems
Preemption-threshold	Reduced unnecessary preemption	Requires careful offline tuning
Fixed-size block pools	Deterministic O(1) allocation	Reduced flexibility, potential memory inefficiency
Design-time configuration	Easier verification	Poor support for highly dynamic workloads

Memory fragmentation and allocation trade-offs

ThreadX balances flexibility and predictability by supporting both byte pools and block pools. Byte pools allow variable-size memory allocation and are useful for non-critical tasks or initialization phases. However, allocation time in byte pools depends on the current fragmentation state of memory. This makes WCET analysis difficult and limits their use in hard real-time execution paths [22].

Block pools, in contrast, provide constant-time allocation and deallocation, eliminating external fragmentation and simplifying timing analysis [23]. These properties make block pools well suited for time-critical tasks and safety-related data structures [9]. The main drawback is reduced flexibility: fixed block sizes may lead to unused memory or inefficient utilization when allocation sizes vary. Adjusting block pool configurations often requires design changes, which can be inconvenient for long-lived or evolving systems.

Table 5. Trade-offs between byte pools and block pools in ThreadX memory management

Aspect	Byte pools	Block pools
Allocation granularity	Variable-size	Fixed-size
Allocation time	Depends on fragmentation state	Constant-time *(O(1))*
WCET analyzability	Difficult	Straightforward
External fragmentation	Possible	Eliminated
Flexibility	High	Limited
Memory utilization	Potentially efficient but unstable	Predictable but may waste memory
Suitability	Non-critical or initialization tasks	Time-critical and safety-related tasks
Scalability impact	Fragmentation increases with system evolution	Pool management complexity increases with system size

Table 5 indicates that ThreadX memory management requires careful design-time decisions to balance flexibility and deterministic behavior, particularly in large and evolving embedded systems [9].

Overall, as embedded systems grow more complex, selecting the appropriate balance between byte pools and block pools becomes increasingly challenging. Developers must carefully decide which parts of the system require strict determinism and which can tolerate more flexible but less predictable memory behavior.

3.5. Open research directions and future trends

Although Azure ThreadX is a mature and industrially deployed RTOS, several open research directions remain to further extend its capabilities. Recent RTOS research indicates that deterministic kernel foundations such as ThreadX can support more adaptive, energy-efficient, and verification-oriented real-time platforms [15].

One key direction concerns adaptive scheduling extensions within the ThreadX model, where controlled, mode-dependent or workload-aware adjustments of priorities or preemption thresholds may be explored to improve flexibility in mixed-criticality scenarios while preserving bounded execution behavior [30]. Another important trend involves energy-aware mechanisms built on the predictable scheduling and synchronization properties of the ThreadX kernel, enabling integration of low-power modes and dynamic frequency scaling without compromising real-time guarantees [19].

In parallel, formal verification of ThreadX kernel mechanisms is gaining relevance, ThreadX are well suited for verifying properties such as bounded priority inversion and timing correctness in safety-critical systems [31]. Collectively, these directions position ThreadX as a foundational RTOS for sustainable cyber-physical systems requiring *predictable execution, energy efficiency, and verification-friendly design.*

These research directions can be summarized in a conceptual roadmap that positions ThreadX as a deterministic kernel foundation supporting adaptive, energy-aware, and verification-oriented real-time systems.

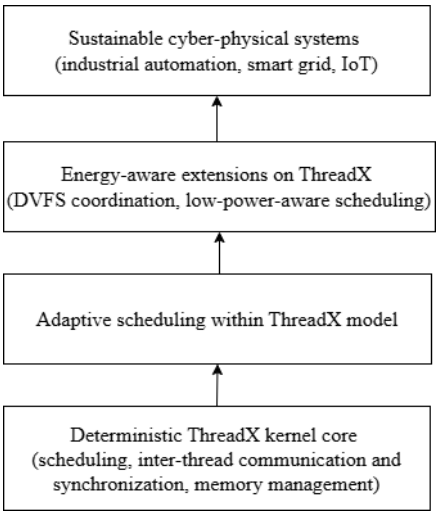

Figure 12. Conceptual research roadmap for Azure ThreadX, illustrating how its deterministic kernel core can be extended

4. Conclusions

This article has examined Azure ThreadX on STM32 platforms through an analysis of its kernel architecture, scheduling behavior, inter-thread communication and synchronization mechanisms, memory management strategies, and hardware integration. The discussion shows that ThreadX provides deterministic and predictable execution by combining priority-based preemptive scheduling with a preemption-threshold mechanism, well-defined inter-thread communication primitives, bounded synchronization behavior, and pool-based memory management. These mechanisms allow concurrent tasks to coordinate safely and efficiently while keeping execution timing under control under compact kernel core.

Compared with other embedded RTOS platforms, ThreadX offers a balanced design that emphasizes predictability and stability rather than extensive feature sets. Its compact kernel footprint and clear separation of real-time services make it suitable for resource-constrained microcontrollers and applications where timing correctness and system reliability are critical, such as industrial control and safety-related systems. At the same time, predictable scheduling

and communication behavior help reduce unnecessary processor activity, supporting energy-efficient operation. Finally, emerging research directions, including adaptive scheduling extensions, energy-aware mechanisms, and formal verification of kernel properties, further extend the role of ThreadX beyond current industrial deployments. These directions highlight the potential of ThreadX as a reliable real-time foundation for future embedded and sustainable cyber-physical systems that require understandable design, predictable behavior, and long-term maintainability.

References

[1] Y. H. Hee, M. K. Ishak, M. S. M. Asaari, and M. T. A. Seman, "Embedded operating system and industrial applications: a review," *Bull. Electr. Eng. Inform.*, vol. 10, no. 3, pp. 1687–1700, Jun. 2021, doi: 10.11591/eei.v10i3.2526.

[2] I. Ungurean, "Timing Comparison of the Real-Time Operating Systems for Small Microcontrollers," *Symmetry*, vol. 12, no. 4, p. 592, Apr. 2020, doi: 10.3390/sym12040592.

[3] Y. Mazzi, A. Gaga, and F. Errahimi, "Benchmarking and Comparison of Two Open-source RTOSs for Embedded Systems Based on ARM Cortex-M4 MCU," *Indian J. Sci. Technol.*, vol. 14, no. 16, pp. 1261–1273, Apr. 2021, doi: 10.17485/IJST/v14i16.387.

[4] P. Hambarde, R. Varma, and S. Jha, "The Survey of Real Time Operating System: RTOS," Jan. 2014, pp. 34–39. doi: 10.1109/ICESC.2014.15.

[5] A. F. H. Dhrubo and M. A. Qayum, "STM32-Based IoT Framework for Real-Time Environmental Monitoring and Wireless Node Synchronization," Oct. 20, 2025, *arXiv*: arXiv:2506.17295. doi: 10.48550/arXiv.2506.17295.

[6] Y. Zhang and J. Li, "Design of Real-Time Embedded System Based on STM32 and Free RTOS with Optimization of Advanced Task Scheduling Algorithm," *2025 3rd Cogn. Models Artif. Intell. Conf. AICCONF*, pp. 1–5, Jun. 2025, doi: 10.1109/AICCONF64766.2025.11063903.

[7] S. Baskiyar and N. Meghanathan, "A Survey of Contemporary Real-time Operating Systems," *Inform. Slov.*, 2005, Accessed: Jan. 21, 2026. [Online]. Available: https://www.semanticscholar.org/paper/A-Survey-of-Contemporary-Real-time-Operating-Baskiyar-Meghanathan/8269eb644fa8519e64f6bc0c1fb1711513571a55

[8] R. A. Walker and S. Chaudhuri, "Introduction to the scheduling problem," *IEEE Des. Test Comput.*, vol. 12, no. 2, pp. 60–69, 1995, doi: 10.1109/54.386007.

[9] M. Borges, S. Paiva, A. Santos, B. Gaspar, and J. Cabral, "Azure RTOS ThreadX Design for Low-End NB-IoT Device," *2020 2nd Int. Conf. Soc. Autom. SA*, pp. 1–8, May 2021, doi: 10.1109/SA51175.2021.9507191.

[10] R. Raymundo Belleza and E. de Freitas Pignaton, "Performance study of real-time operating systems for internet of things devices," *IET Softw.*, vol. 12, no. 3, pp. 176–182, 2018, doi: 10.1049/iet-sen.2017.0048.

[11] X. Shao *et al.*, "The Cost of Performance: Breaking ThreadX with Kernel Object Masquerading Attacks," Apr. 28, 2025, *arXiv*: arXiv:2504.19486. doi: 10.48550/arXiv.2504.19486.

[12] Onell Allan Chakawuya, Forhad Monjur Hasan, and Mostafa Mohammad Razaul, "Real-time Operating Systems (RTOS) For Edge AI," Jul. 2025, doi: 10.5281/ZENODO.16729307.

[13] N. He and H.-W. Huang, "Using Open-source Eclipse ThreadX in a Real-time Embedded Systems Lab," in *Proceedings of the Workshop on Computer Architecture Education*, in WCAE '25. New York, NY, USA: Association for Computing Machinery, Oct. 2025, pp. 1–8. doi: 10.1145/3743646.3750015.

[14] C. Ngolah, Y. Wang, and X. Tan, "The real-time task scheduling algorithm of RTOS+," *Can. J. Electr. Comput. Eng.*, vol. 29, pp. 236–242, Oct. 2004, doi: 10.1109/CJECE.2004.1426051.

[15] W. Cedeño and P. A. Laplante, "An Overview of Real-Time Operating Systems," *SLAS Technol.*, vol. 12, no. 1, pp. 40–45, Feb. 2007, doi: 10.1016/j.jala.2006.10.016.

[16] S. Kim, "Assigning Priorities for Fixed Priority Preemption Threshold Scheduling," *ScientificWorldJournal*, vol. 2015, p. 837472, 2015, doi: 10.1155/2015/837472.

[17] C.-S. Chen and C.-C. Tseng, "Integrated support to improve inter-thread communication and synchronization in a multithreaded processor," in *Proceedings of 1994 International Conference on Parallel and Distributed Systems*, Dec. 1994, pp. 481–486. doi: 10.1109/ICPADS.1994.590359.

[18] D. Voitsechov, O. Port, and Y. Etsion, "Inter-Thread Communication in Multithreaded, Reconfigurable Coarse-Grain Arrays," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct. 2018, pp. 42–54. doi: 10.1109/MICRO.2018.00013.

[19] A. P. Nyrkov, K. A. Ianiushkin, A. A. Nyrkov, Y. N. Romanova, and V. D. Gaskarov, "Dynamic Shared Memory Pool Management Method in Soft Real-Time Systems," in *2020 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*, Jan. 2020, pp. 438–440. doi: 10.1109/EIConRus49466.2020.9039354.

[20] "Chapter 8: Memory Management: Byte Pools and Block Pools | GlobalSpec." Accessed: Jan. 22, 2026. [Online]. Available: https://www.globalspec.com/reference/25415/203279/chapter-8-memory-management-byte-pools-and-block-pools?utm_source=chatgpt.com

[21] B. Pitre and M. Margala, "A Novel Approach to Managing System-on-Chip Sub-Blocks Using a 16-Bit Real-Time Operating System," *Electronics*, vol. 13, no. 10, May 2024, doi: 10.3390/electronics13101978.

[22] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, "Dynamic storage allocation: A survey and critical review," in *Memory Management*, vol. 986, H. G. Baler, Ed., in Lecture Notes in Computer Science, vol. 986. , Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 1–116. doi: 10.1007/3-540-60368-9_19.

[23] I. Puaut, "Real-time performance of dynamic memory allocation algorithms," in *Proceedings 14th Euromicro Conference on Real-Time Systems. Euromicro RTS 2002*, Jun. 2002, pp. 41–49. doi: 10.1109/EMRTS.2002.1019184.

[24] M. Hughes, Y. Zhou, and D. Morgan, *Integrated Memory Control and Thread Scheduling for Real-Time Voice Interaction Systems*. 2025. doi: 10.21203/rs.3.rs-7863332/v1.

[25] "Introduction to Azure RTOS® with STM32 - stm32mcu." Accessed: Jan. 22, 2026. [Online]. Available: https://wiki.st.com/stm32mcu/wiki/Introduction_to_Azure_RTOS_with_STM32?utm_source=chatgpt.com

[26] Y. Zhang and J. Li, "Design of Real-Time Embedded System Based on STM32 and Free RTOS with Optimization of Advanced Task Scheduling Algorithm," in *2025 3rd Cognitive Models and Artificial Intelligence Conference (AICCONF)*, Jun. 2025, pp. 1–5. doi: 10.1109/AICCONF64766.2025.11063903.

[27] "Zephyr RTOS vs FreeRTOS: A Comprehensive Comparison for IoT and Embedded Systems," Ezurio. Accessed: Jan. 21, 2026. [Online]. Available: https://www.ezurio.com/resources/blog/zephyr-rtos-vs-freertos-a-comprehensive-comparison-for-iot-and-embedded-systems

[28] "Measuring Real-Time Operating System Performance – Part II: Comparing FreeRTOS vs. Zephyr," UL Solutions. Accessed: Jan. 23, 2026. [Online]. Available: https://www.ul.com/sis/blog/measuring-real-time-operating-system-performance-part-ii-comparing-freertos-vs-zephyr

[29] I. L. Orășan, C. Seiculescu, and C. D. Căleanu, "A Brief Review of Deep Neural Network Implementations for ARM Cortex-M Processor," *Electronics*, vol. 11, no. 16, Aug. 2022, doi: 10.3390/electronics11162545.

[30] J. Regehr, "Scheduling tasks with mixed preemption relations for robustness to timing faults," in *23rd IEEE Real-Time Systems Symposium, 2002. RTSS 2002.*, Dec. 2002, pp. 315–326. doi: 10.1109/REAL.2002.1181585.

[31] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronization," *IEEE Trans. Comput.*, vol. 39, no. 9, pp. 1175–1185, Sep. 1990, doi: 10.1109/12.57058.