Real-Time Operating System (RTOS)

2024 Performance Report

By: Jacob Beningo

BENINGO
EMBEDDED GROUP

EMBEDDED
SOFTWARE ACADEMY
BY BENINGO

www.beningo.com
Contact: jacob@beningo.com

CONTENTS

03 Introduction

05 Acknowledgments

07 The Methodology

09 Benchmark Results

10 Basic Processing

11 Cooperative Scheduling

13 Preemptive Scheduling

15 Memory Allocation

16 Message Processing

17 Synchronization Processing

18 Conclusions

22 Going Further

26 Appendices

Introduction

In today's rapidly evolving landscape of embedded systems and IoT, devices are becoming more interconnected and their roles more sophisticated, especially at the edge. This evolution has driven an unprecedented demand for precise and deterministic timing management, where even the smallest delays can lead to significant consequences. The proliferation of advanced technologies-including sensor arrays, human-machine interfaces (HMIs), communication protocols, encryption algorithms, and artificial intelligence (AI)-has elevated the importance of an efficient Real-Time Operating System (RTOS) from a beneficial asset to a critical necessity.

The market offers over a hundred RTOS options, ranging from simple, open-source schedulers to highly specialized, safety-critical, certified commercial solutions. However, despite the wide selection, not all RTOS solutions are created equal, and the choice of an RTOS can significantly impact the performance and success of your application.

Consider, for example, a low-power IoT sensor node that operates on a battery. The device must balance power consumption with performance, ensuring it can remain operational for months or years without battery replacement. An efficient RTOS makes a crucial difference in this scenario: it can manage the device's sleep cycles, wake up the processor only when necessary, and execute tasks with minimal energy overhead. In contrast, an inefficient RTOS might keep the processor awake longer than needed, draining the battery rapidly and reducing the device's operational lifespan. Such differences in RTOS efficiency can determine whether an IoT deployment is viable or doomed to fail.

This report represents an in-depth benchmarking analysis of three popular open-source RTOSes and one commercially available RTOS. We will examine their performance across key areas that are vital to the functioning of embedded systems, including:

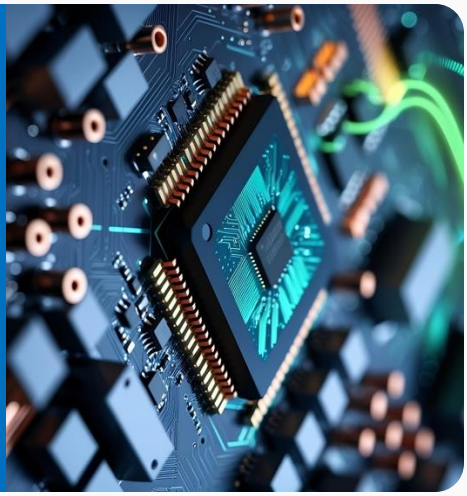
- Cooperative Scheduling
- Preemptive Scheduling
- Memory Allocation
- Synchronization Processing
- Message Processing

While performance is just one of many factors to consider when selecting an RTOS, (see 7 Characteristics to Consider when Selecting an RTOS), it is a critical metric that directly influences your system's reliability and efficiency. As the industry continues to trend toward more complex and interconnected systems, understanding the performance characteristics of an RTOS becomes essential for ensuring that your embedded applications can meet the increasingly demanding requirements of today's technological landscape.

In the future, we plan to expand this study to include additional RTOSes and delve into the impact of POSIX APIs on performance. Until then, enjoy the results we've found so far, and think carefully about how they may or may not impact how you design your real-time applications.

Happy Coding,

"In engineering, accuracy is paramount, but even the most skilled minds can overlook details. That's why reviews are essential—ensuring that fresh eyes scrutinize every design, calculation, and line of code to eliminate bias and achieve the most reliable, objective results."



Acknowledgments

Accurate and unbiased benchmark data is critical in testing like this. I want to thank several people for participating and assisting in this RTOS Performance study.

1) Mohammed Billoo from MAB Labs for assisting in Zephyr RTOS benchmarking and acting as a second set of eyes on the data

2) Jean Labrosse, your review of the draft of this report was instrumental. Your insightful feedback and thought-provoking questions were key in ensuring the accuracy and unbiasedness of the results.

3) Bill Lamie from PX5 RTOS, we appreciate the risk you took in providing us with the PX5 RTOS for testing and comparison. Your courage and trust in our study were commendable.

4) Shawn Prestridge and Rafael Taubinger from IAR for providing me with an extended license for IAR-Embedded Workbench for Arm so I could complete this study.

5) Naymul Hasan, who takes my napkin chicken scratch and half-thought diagrams and brings them to life with color and visual appeal

Trademark Acknowledgments

- **FreeRTOS** is a trademark of Amazon Web Services, Inc.
- **Zephyr RTOS** is a trademark of the Linux Foundation.
- **PX5** is a trademark of PX5 RTOS, Inc.
- **Eclipse ThreadX** is a trademark of Eclipse Foundation.

All other product names, logos, and brands are the property of their respective owners. Use of these names, logos, and brands does not imply endorsement.

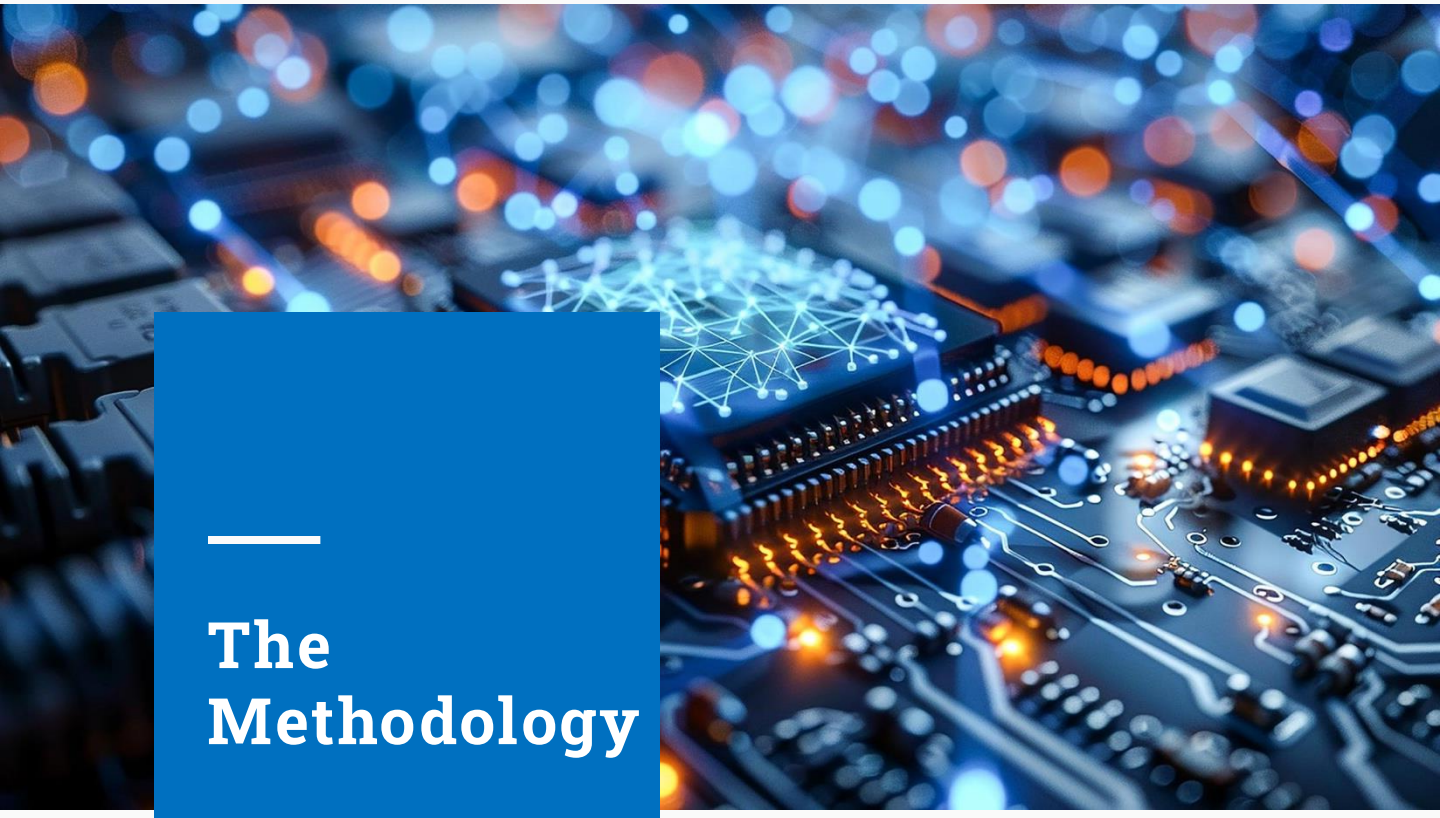
Non-Affiliation Statement

This performance study is conducted independently and is not affiliated with or endorsed by FreeRTOS (Amazon Web Services, Inc.), Zephyr RTOS (Linux Foundation), PX5 (PX5 RTOS, Inc.), Eclipse ThreadX (Eclipse Foundation), or any other RTOS provider mentioned in this paper. All findings are based on objective testing criteria and reflect the results of independent analysis.

Objective Performance Comparison

The performance comparisons presented in this study are based on independent testing and publicly available data for FreeRTOS, Zephyr RTOS, PX5, and Eclipse ThreadX (referred to as ThreadX throughout this paper). All testing criteria are objective and intended for informational purposes only. The results do not reflect the views or opinions of the RTOS providers and should be interpreted as an independent analysis.

It's important to note that each RTOS is designed to meet different requirements and use cases, and the choice of RTOS should be based on the specific needs of a project. Factors such as resource constraints, real-time performance, scalability, and ecosystem support vary between systems and should be carefully evaluated when determining fitness for purpose in any given application.



The Methodology

Our RTOS benchmark analysis was based on a series of tests from the Thread Metric benchmark suite, formerly offered by Microsoft and now offered by the Eclipse Foundation. The Thread Metric benchmark suite is under MIT license and can be found on GitHub here: Thread Metric Benchmark Suite.

The Thread Metric benchmark suite consists of eight tests, one of which is a calibration test to ensure the platform's operation is similar. For this study, we have selected the following five Thread Metric RTOS performance tests:

- Cooperative Scheduling
- Preemptive Scheduling
- Memory Allocation
- Synchronization Processing
- Message Passing

Each test is based on total throughput within a 30-second time interval, with higher throughput indicating a more efficient RTOS.

We conducted the analysis on the ST STM32L4 IoT Discovery Node (B-L475E-IOT01A), equipped with an 80MHz Cortex-M4 processor. Results were consistent across multiple boards, with variations within a 1% margin. The repeatability of the tests easily fell within this margin, and in most cases, it was much better. (Which is what you'd expect from a deterministic RTOS).

The original intent was to compile each RTOS in IAR EWARM and run the tests. We discovered that Zephyr RTOS was tightly coupled to its build system, and the time required to set it up with IAR Embedded Workbench for Arm was prohibitive. That forced us to make some adjustments.

The RTOS and Thread Metric benchmark code were compiled using GCC version 12.3 (or similar). All RTOS C code was optimized for speed with uniform compilation flags across all RTOS versions. The specific versions used in this analysis are:

- FreeRTOS 11.1.0
- PX5 RTOS 5.1.0
- Eclipse ThreadX 6.1.1
- Zephyr 3.7

The RTOSes were left in their default configuration for the STM32L475. The configuration matters because, for your application, you might use different configuration settings that affect how the RTOS performs. The defaults were used because they are the most likely settings most embedded system teams use. Some configurations were adjusted to ensure that the comparisons were as close to "apples to apples" as possible.

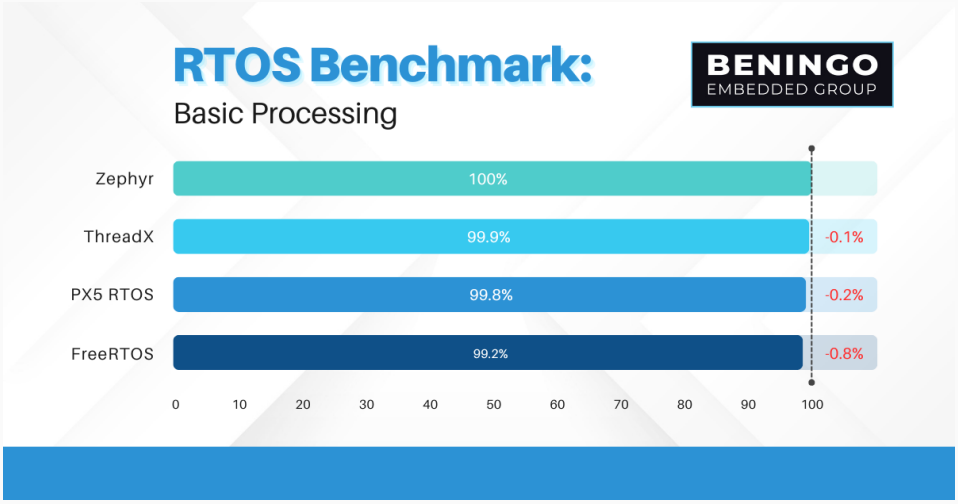
For example, not all RTOSes support argument checking, assertions, or might have additional checks that are performed in debug builds. We disabled these features since they would typically be disabled in production, allowing for the most reliable comparison between RTOSes.

I've provided the configuration settings for each RTOS in the appendices so you can easily see the test conditions. This is important for transparency and for anyone to reproduce the results.

Benchmark Test Results

In this section, we'll review the details of each benchmark test and how the RTOSes performed relative to each. Each section provides the results, with the best-performing RTOS scaled to 100% and the performance of every other RTOS scaled based on the percentage of the best-performing RTOS.

We also add some basic commentary so that you can understand the test and what it means. We point out any interesting information that we discovered along the way as well.



Basic Processing Test

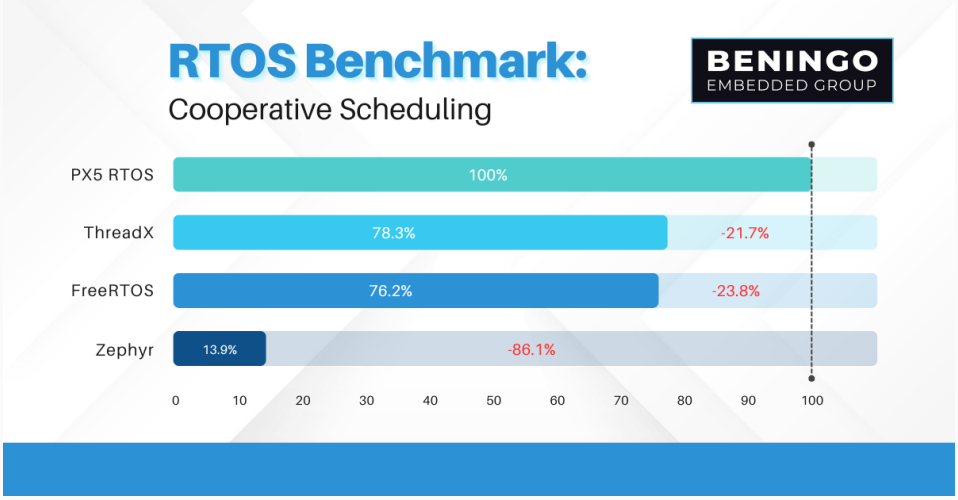
Benchmark Overview

The Basic Processing Test is a calibration test. It is a single thread incrementing a counter. It should be nearly the same on every operating system. The calibration test shows that generic, non-RTOS code ran at the same speed under each RTOS on the same board with the same compiler. It also lets us know that the settings for each RTOS are configured the same. The results are expected to fall within 1%. Any significant variation may indicate a setup issue or the need to scale the results for a more accurate comparison.

Benchmark Results:

The calibration results (Basic Processing Test) were nearly identical for all the RTOSes tested. It averaged 70,782 counts +/- 0.5% over the 30-second interval.

The FreeRTOS basic processing test ran slightly slower than the other RTOSes, but within a 1% spread between the fastest-running calibration tests. The test gives us confidence that the compiler, hardware, and RTOSes are set up correctly and that there are no significant differences in the compiler flags or settings.



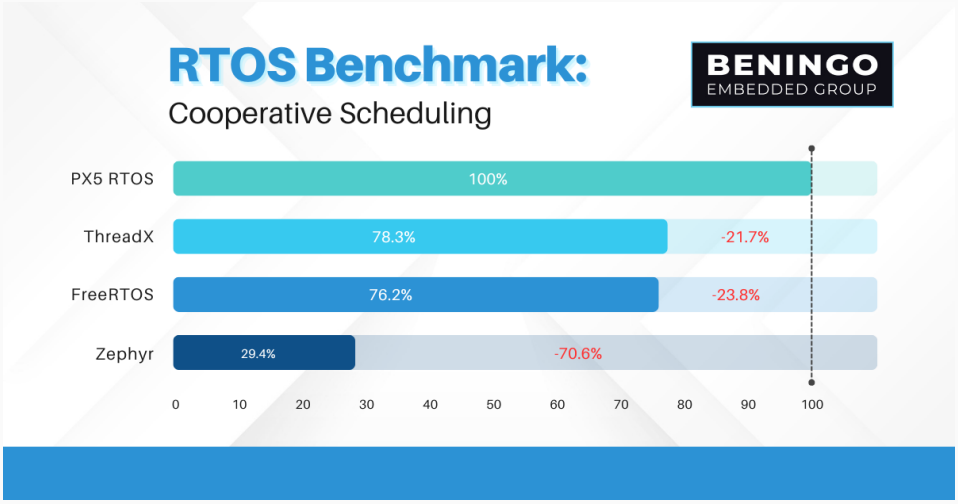
Cooperative Scheduling

Benchmark Overview

Cooperative Scheduling is a scheduling method in which tasks voluntarily yield control of the CPU to allow other tasks to run. Unlike preemptive scheduling, where the RTOS can interrupt and switch between tasks based on priority or time slices, cooperative scheduling relies on each task explicitly giving up the CPU when it has finished its current operation or reaches a suitable point to allow other tasks to execute.

The cooperative scheduling benchmark creates five threads at the same priority level, each voluntarily releasing control in a round-robin fashion. Each thread increments a counter and then relinquishes the CPU to allow other threads to execute.

At the end of the test, the counters from each thread are verified. In a deterministic scheduler, you would expect the counters to all be within a single count of each other. Once the results are verified, the numbers are summed and presented as the result of the cooperative scheduling test.



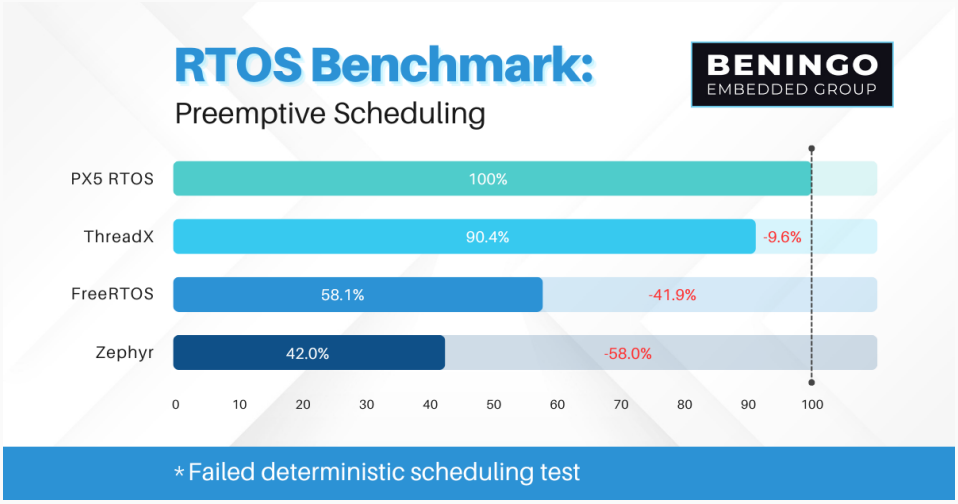
Benchmark Results

The cooperative scheduling benchmark results can be seen in the image and came with a few surprises.

The highest-performing RTOS, PX5 RTOS, performed 3.4 times faster than the slowest, Zephyr. That means for every one time that each thread ran in Zephyr, each PX5 RTOS thread ran 3.4 times! ThreadX and FreeRTOS both performed ~2.6 times faster than Zephyr.

One interesting discovery about FreeRTOS was that the cooperative scheduler becomes non-deterministic if you compile it without optimizations. While this does not impact test results, it's an important consideration given the number of teams that compile their code without optimizations enabled! If you use FreeRTOS, make sure that you are compiling for speed!

Another observation was that using Zephyr's default configuration, PX5 performs 7.2 times faster than Zephyr! Appendix D gives the configuration values to speed up Zephyr. They are not default so if you don't tune your RTOS, it will run slow!



Preemptive Scheduling

Benchmark Overview

Preemptive scheduling is a type of task scheduling in which the operating system can interrupt and suspend a currently running task to allocate CPU time to a higher-priority task. This approach ensures that critical tasks receive timely execution, improving responsiveness and system performance.

The test consists of 5 threads that each have a unique priority. Each thread, except the lowest priority thread, is left in a suspended state. The lowest priority thread will resume the following highest priority thread. That thread will resume the following highest priority thread and so on until the highest priority thread executes.

Each thread will increment its run count and then call thread suspend. Eventually the processing will return to the lowest priority thread, which is still in the middle of the thread resume call. Once processing returns to the lowest priority thread, it will increment its run counter and resume the next highest priority thread again, starting the whole process again.

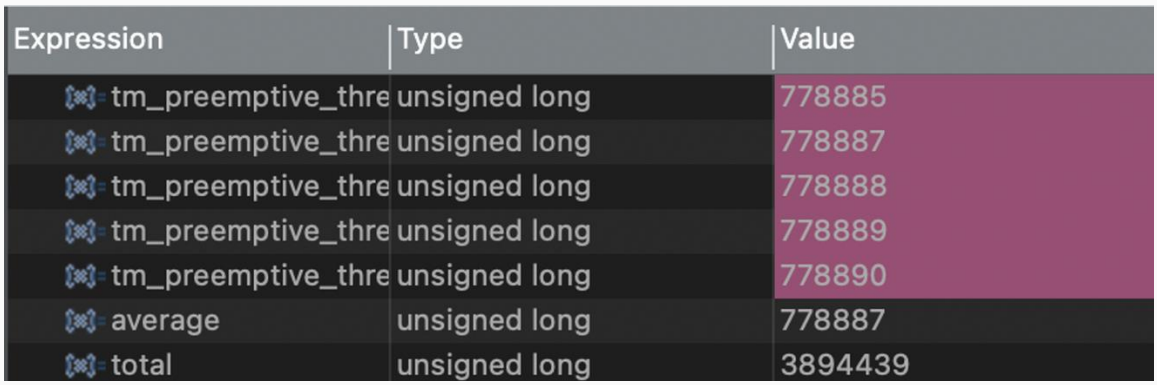
At the end of the test, the counters from each thread are verified. In a deterministic scheduler, you would expect the counters to all be within a single count of each other. Once the results are verified, the numbers are summed and presented as the result of the preemptive scheduling test.

Benchmark Results

The preemptive scheduling benchmark results can be seen in the image and came with a few surprises.

The highest-performing RTOS, PX5 RTOS, performed 2.4 times faster than the slowest-performing RTOS, Zephyr. That means for every one time that each thread ran in Zephyr, each PX5 RTOS thread ran 2.4 times! ThreadX came in a close second but still performed 9.6% slower than PX5 RTOS.

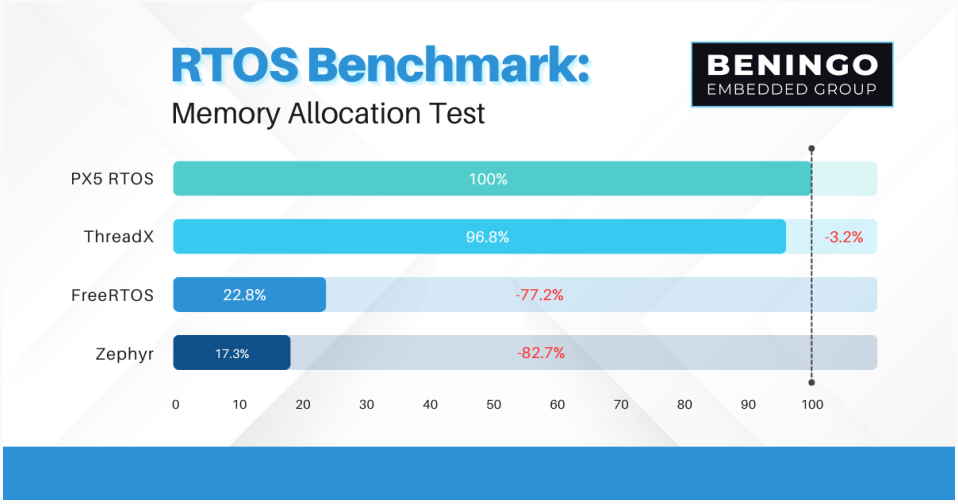
One surprising discovery about FreeRTOS was that the test harness considers the preemptive scheduler non-deterministic. To indicate a deterministic scheduler, the counters should all be within a single count of each other. As you can see in the image below, they are close but not entirely within one.



Expression	Type	Value
tm_preemptive_thre	unsigned long	778885
tm_preemptive_thre	unsigned long	778887
tm_preemptive_thre	unsigned long	778888
tm_preemptive_thre	unsigned long	778889
tm_preemptive_thre	unsigned long	778890
average	unsigned long	778887
total	unsigned long	3894439

Depending on your requirements and needs, the effect may be insignificant to your specific application. It may be more prevalent for systems expected to run non-stop over long periods without a reset.

I don't expect this finding to change anyone's use case for FreeRTOS, but it's still worth mentioning.



Memory Allocation Test

Benchmark Overview

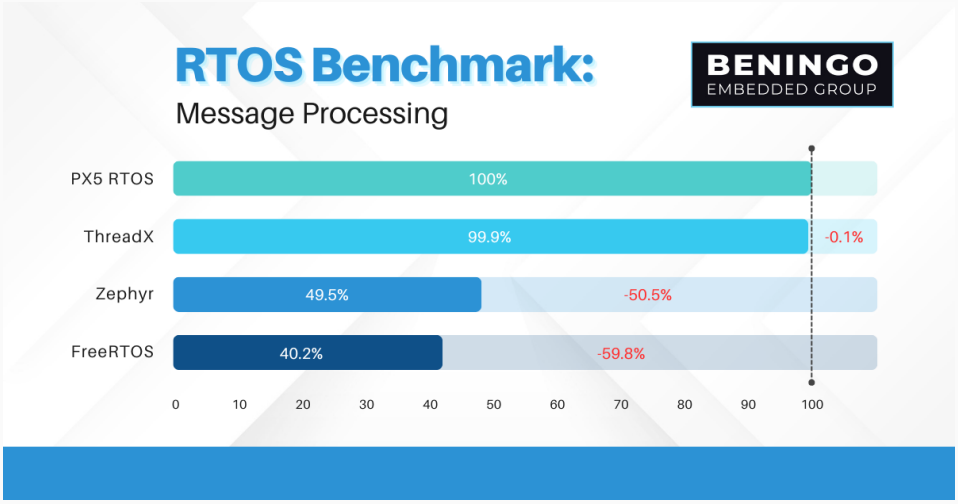
The RTOS Memory Allocation test determines how efficiently the RTOS allocates and releases memory. The test consists of a thread allocating a 128-byte block and releasing it. After the block is released, the thread increments its run counter. The process then repeats for a 30-second time interval.

Benchmark Results:

The memory allocation benchmark results can be seen in the image above. As usual, we scale the results based on the fastest and provide you with the percentage of the quickest that each RTOS scored.

Once again, we find that PX5 RTOS crushes it. However, this time ThreadX is not far behind, trailing by only 3.2%. The spread between PX5 RTOS and Zephyr is quite large, with PX5 RTOS being 5.8 times faster than Zephyr!

These results won't mean much for RTOS applications that use static memory allocation, but quite a few IoT edge devices I see use dynamic memory allocation, making this metric all the more important.



Message Processing

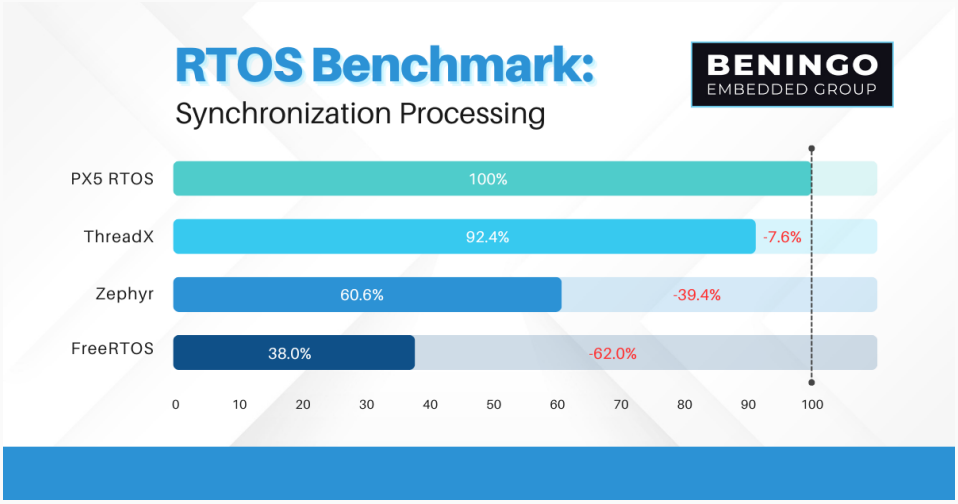
Benchmark Overview

The Message Processing benchmark determines how efficiently the RTOS can send and receive messages. The test consists of a thread sending 16-byte messages to a queue and retrieving the same 16-byte message from the queue. After the send/receive sequence, the thread increments its counter and repeats for the 30-second test.

Benchmark Results:

The spread for message processing was narrower than in other tests. ThreadX and PX5 RTOS were nearly tied for performance, each about two times faster than Zephyr.

The benchmark is the first, where Zephyr beat out FreeRTOS for the slowest. Zephyr was about 9.3% faster at processing messages than FreeRTOS.



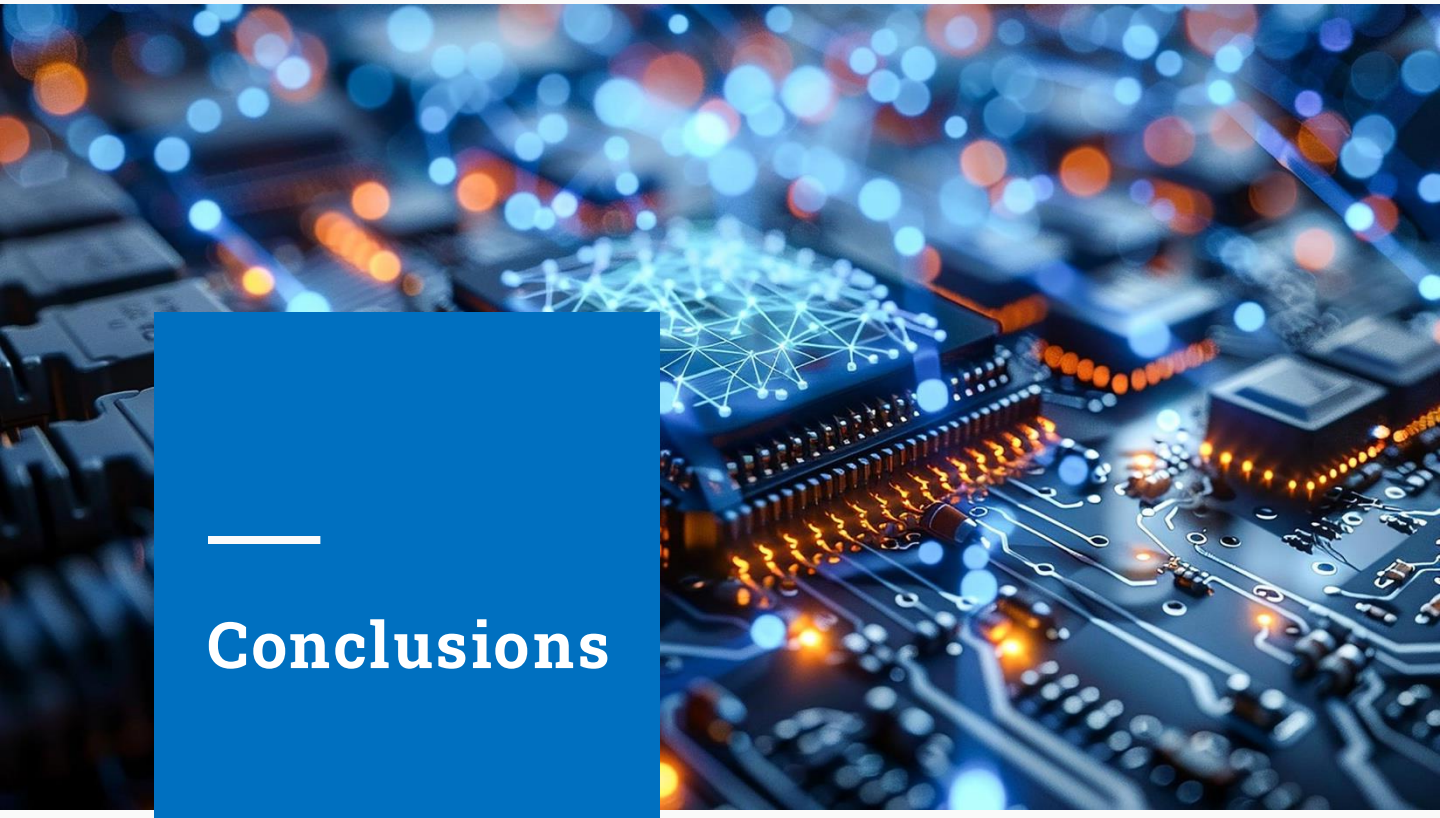
Synchronization Processing

Benchmark Overview

The Synchronization Processing test evaluates the performance of synchronizing threads. It consists of a single thread getting a semaphore and immediately releasing it. After the get/put cycle, the thread increments its run counter and repeats the cycle. The counter value is then reported at the end of the 30-second cycle.

Benchmark Results:

Once again, the PX5 RTOS proved to be the fastest, 2.6 times faster than FreeRTOS. ThreadX was a close second but still 7.6% slower than the PX5 RTOS. Zephyr synchronization was about 1.6 times faster than FreeRTOS.



Conclusions

RTOS performance is essential for applications that require high performance and for applications that need to minimize processor cost and/or power consumption. These considerations should apply to all embedded applications. There is no reason to unnecessarily increase the device BOM cost because of poor RTOS performance.

In this section, we'll draw some conclusions from these results. Be warned: How you configure the RTOS and what settings you enable and disable can affect its performance! We've done our best to create an apples-to-apples comparison, but the richness of RTOSes features can make this challenging.

Different settings may produce different results and conclusions. However, we are confident that the settings and results align with how these RTOSes are typically used and configured in embedded systems.

I would encourage you to use these results with your own tests and requirements to help guide you in selecting the right RTOS for your application.

Interpreting the Results

In this paper, we've looked at the various benchmarks with respect to the best performing RTOS, PX5. Below is another way to interpret the results of this study, normalizing each benchmark for the slowest performer in each category:

RTOS Benchmark Test	ThreadX	PX5 RTOS	FreeRTOS v.10.3.1	Zephyr 3.7.0
Basic Processing	1.0x	1.0x	1.0x	1.0x
Cooperative Scheduling	2.7x	3.4x	2.6x	1.0x
Memory Allocation Test	5.6x	5.8x	1.3x	1.0x
Message Processing	2.5x	2.5x	1.0x	1.2x
Preemptive Scheduling	2.2x	2.4x	1.4x	1.0x
Synchronization Processing	2.4x	2.6x	1.0x	1.6x

We've drawn several observations and conclusions from this table and our work with each of these RTOSes.

1. **There is a clear distinction between Commercial and Open-Source Software.**

   The commercially available PX5 RTOS outperformed the three open-source RTOSes—in some cases, by as much as 5.8 times faster! This highlights the adage that "you get what you pay for."

   ThreadX, while now open-source, offers performance close to PX5 RTOS and was once a commercial RTOS before being acquired and open-sourced by Microsoft. Both PX5 RTOS and ThreadX were developed by Bill Lamie, a veteran in RTOS development, yet PX5 RTOS continues to edge out ThreadX in performance.

   Although open-source software is often valued for its transparency and broad community support, this clearly doesn't translate to superior performance in every case.

2. Open-source software is often Toolchain-restricted

The original plan for this study was to use IAR Embedded Workbench. The idea was to get the best numbers and clearest comparison under the best conditions. However, we quickly discovered that Zephyr and another open-source RTOS we were interested in RT-Thread, would not easily play nice outside of the toolchains they ship with without significant work.

We opted to use GCC to avoid the pain associated with breaking these RTOSes free from their build system chains. If you are working with a commercial product though, what might the costs be if you want to use a commercial compiler?

We should strive to improve Zephyr's inflexibility with other compilers and open-source software without boxing itself into a single-build toolchain. As you'll see later, the compiler itself can be a dramatic source for improving performance.

3. GCC isn't as good as you think

I've often seen, and been guilty myself, of stating that compilers today generate efficient enough code that we don't need to worry about how you write or structure your code. That is an incorrect assumption and statement. It's false!

During this study and throughout other projects I've worked on recently, I've found that GCC, while excellent, doesn't necessarily produce the most efficient and best binaries. The problem is that we don't often have data to show the difference between an open-source compiler like GCC and a commercial compiler.

When I switched from using IAR Embedded Workbench to GCC, I had already collected data for ThreadX, PX5 RTOS, and FreeRTOS using it. So, after repeating the studies in GCC so that Zephyr could be included, it left me with benchmark data to compare IAR Embedded Workbench to GCC.

I took the RTOS with the best results, PX5 RTOS, then took the results for each test and calculated the IAR/GCC and GCC/IAR results. You can see the table results below:

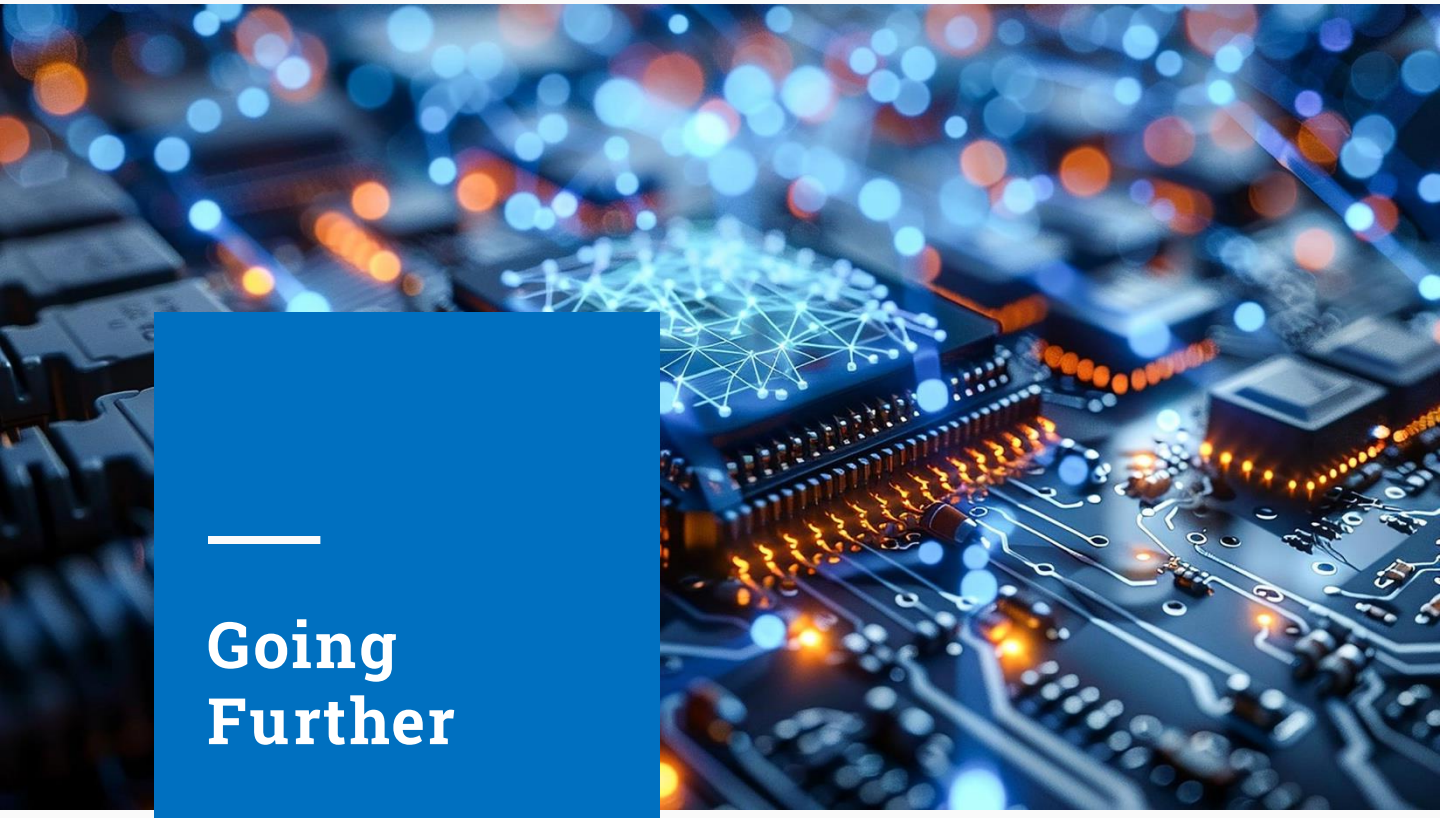
RTOS Benchmark Test	GCC/IAR	IAR/GCC
Basic Processing	57.57%	173.7%
Cooperative Scheduling	100.15%	99.9%
Memory Allocation Test	74.48%	134.3%
Message Processing	72.40%	138.1%
Preemptive Scheduling	81.45%	122.8%
Synchronization Processing	81.94%	122.0%

The compiler flags were set identically in both IAR and GCC. As you can see, the IAR EWARM compiler produced far better results than GCC. It's not a consistent across-the-board amount because it depends on the code being compiled. However, a 20 – 40% performance improvement is a reasonable range.

The Cooperative Scheduling test is likely very similar because that test is beating up the code often written in assembly language. For the Cortex-M, that is the code in the PendSV handler. For PX5 RTOS, there isn't anything for IAR to optimize since that handler is about as efficient as it is going to get. When you look across RTOS implementations though, you see variations in efficiency.

In most industry surveys, real-time performance is one of the most critical issues regarding RTOS selection. However, there are other considerations. Additional considerations include licensing, developer training, middleware support, professional support, and safety certification. That said, RTOS performance is an essential consideration – which, if ignored, can result in a more expensive device that struggles to perform its function.

As our results clearly demonstrate, not all RTOS are created equal. Each has its unique features and solutions to developers' problems. The selection of an RTOS is a decision that should not be taken lightly. Our findings underscore the need for a meticulous and careful approach to RTOS selection and configuration for your applications.



Going Further

RTOS performance is essential for applications that require high performance and for applications that need to minimize processor cost and power consumption. These considerations should apply to all embedded applications. There is no reason to unnecessarily increase the device BOM cost because of poor RTOS performance.

In time, I want to add other RTOSes to this study to provide a fuller picture of RTOS performance and the state of the industry. There are several additional RTOSes that we are looking to add to this study in the future, including:

- RT-Thread
- SEGGERs embOS
- NuttX

With the push to leverage POSIX APIs in embedded applications, we hope also to perform these tests using POSIX APIs in addition to the native RTOS APIs.

Accelerate your RTOS project—consult with an expert to design, optimize, and implement real-time systems.

Are you struggling with project delays, rising costs, or an RTOS system that doesn't scale? We understand the pressure to deliver robust, real-time systems on time and within budget. Many teams face these challenges, but with the right guidance, you can overcome them.

Our expert consulting services help accelerate your RTOS project, ensuring your system is designed for scalability, maintainability, and optimal performance. Whether starting from scratch or refining an existing design, we'll help you create an architecture that meets your immediate needs and supports future growth, saving you money, reducing delays, and avoiding costly redesigns.

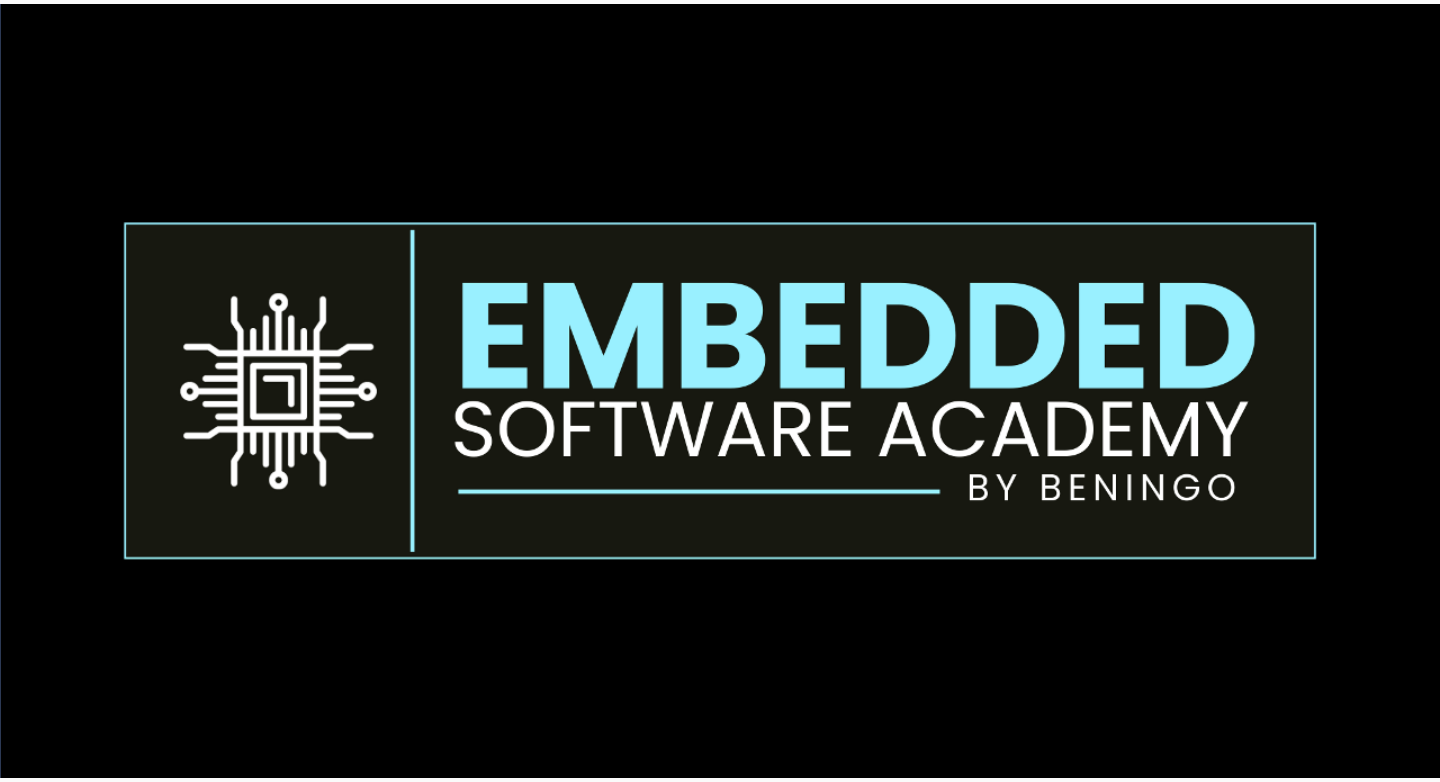
Contact jacob@beningo.com today to see how we can help you bring your RTOS projects to life.

Level up your RTOS skills—design efficient, scalable embedded systems with expert-led training

Working with RTOS applications often leads to frustrating issues like poor performance, scalability issues, and debugging headaches. But it doesn't have to be that way. Our expert-led training helps you overcome these common challenges by teaching you how to design RTOS systems that are efficient, scalable, and ready for production.

Whether you're an individual developer looking to sharpen your skills or a team leader aiming to upskill your engineers, we've got you covered. With flexible training options—on-demand, live online, and customizable team workshops—you can learn how to avoid the pitfalls of RTOS design and build reliable, robust systems.

For more information on how we can help you level up your skills and streamline your RTOS development, contact **jacob@beningo.com** today!



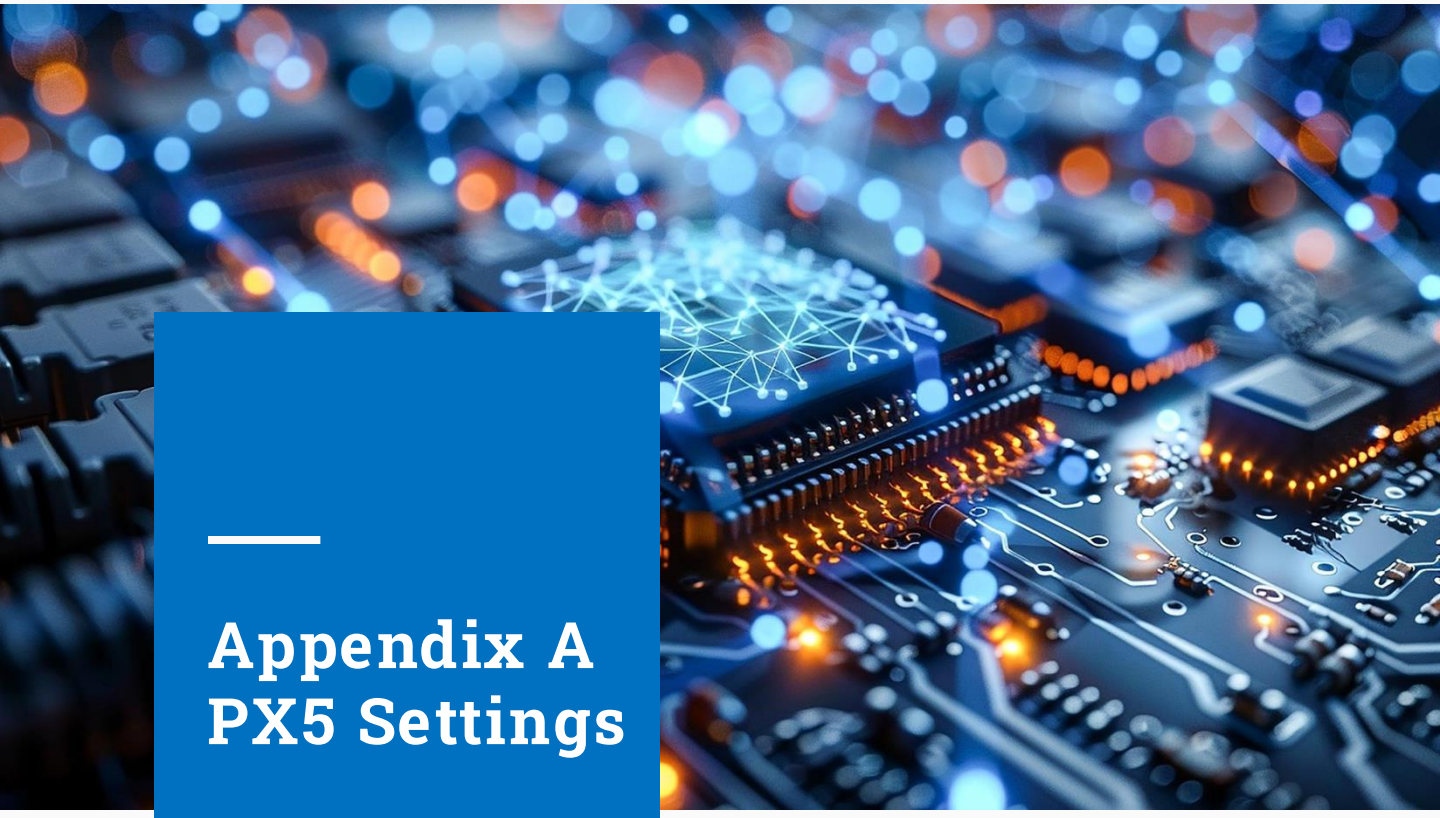
Fast-track your career growth—get the expertise you need to deliver faster, better, and more reliable firmware.

Enhance your skills, streamline your processes, and elevate your architecture. Join my academy for on-demand, hands-on workshops and cutting-edge development resources designed to transform your career and keep you ahead of the curve.

What you'll get:

- Access to over eight hands-on Embedded Software Workshops
- Modernizing Embedded Software Core Courses
- Embedded Software Community Access
- Jacob's Webinar / Presentation Archive
- Embedded Development Q&A's with Jacob Beningo
- Embedded Software Development Resources

Learn more and subscribe by clicking here!



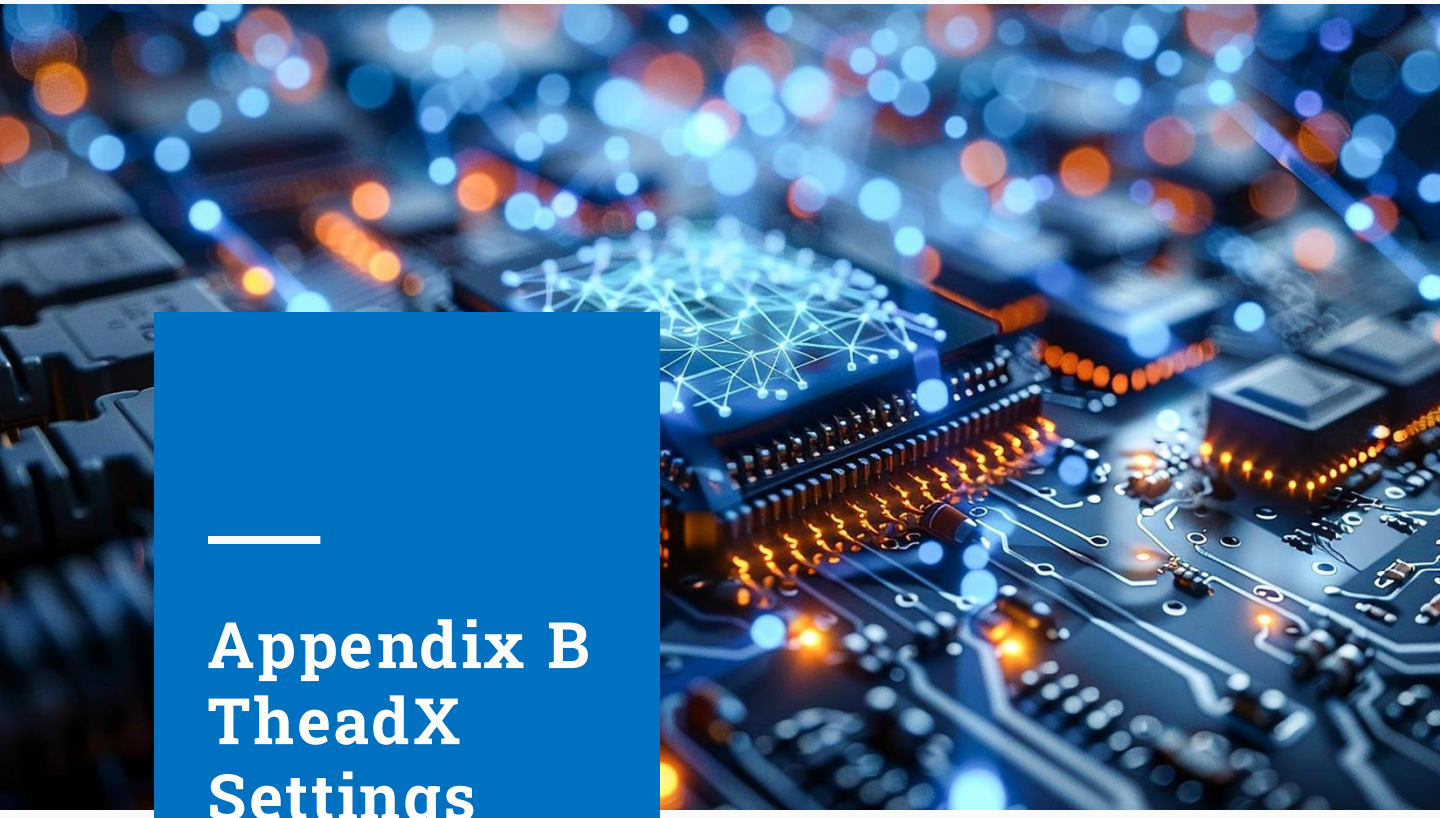
Appendix A PX5 Settings

PX5 was configured in its default configuration. In this configuration, we had the following settings configured in px5_user_config.h that could affect performance:

#define PX5_PARAMETER_CHECKING_DISABLE
#define PX5_CANCELLATION_POINTS_DISABLE

These settings brought PX5 more in line with the other open-source RTOSes we tested. Parameter checking is not standard in most RTOSes. You typically find it in certified commercial RTOSes like PX5, uC OS-III, etc.

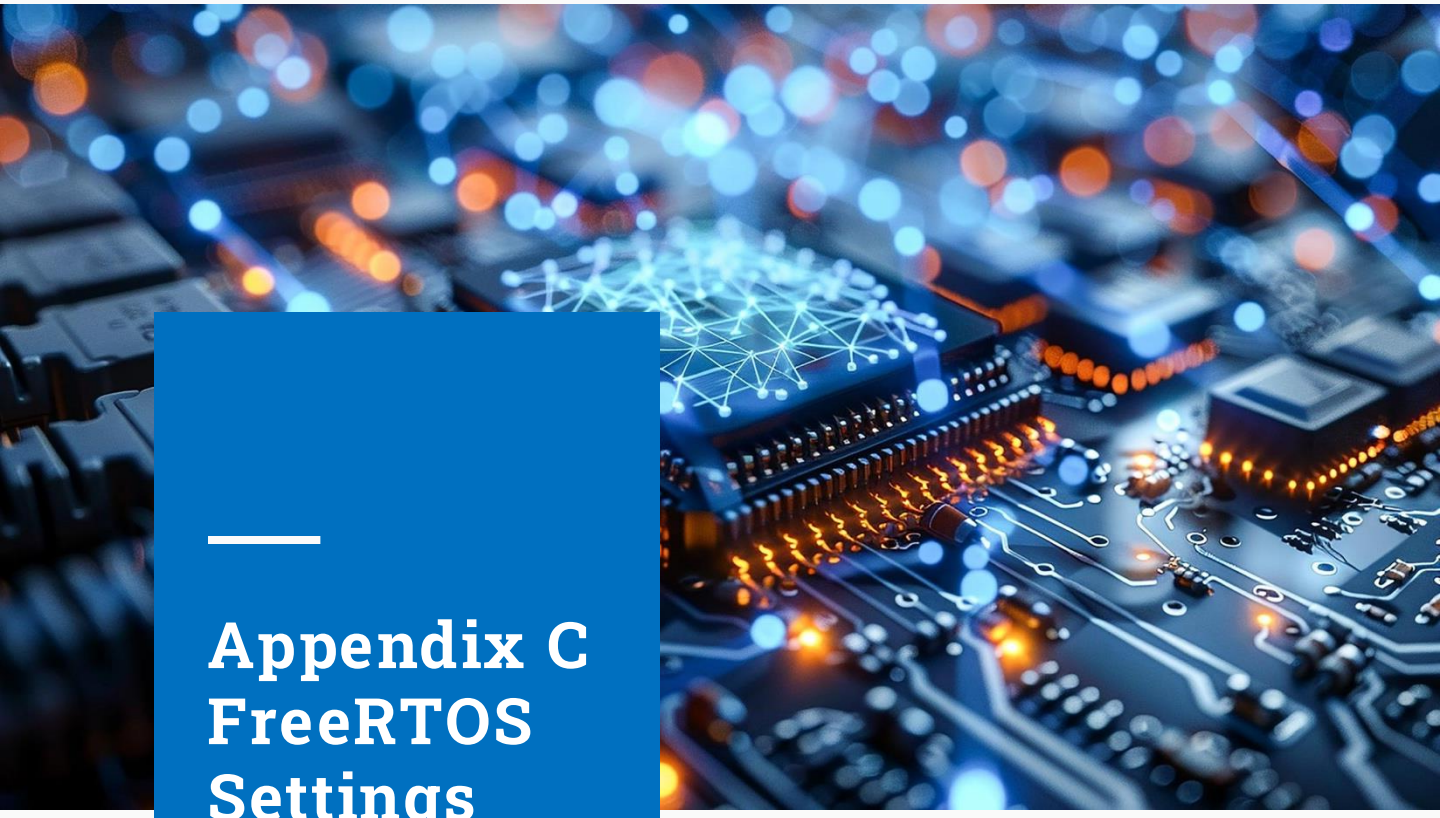
It's typical to enable these features only during development and disable them for production.



Appendix B TheadX Settings

ThreadX was configured in its default configuration. In this configuration, we had the following settings configured tx_port.h. For the tests performed, these should have had minimal, if any impact:

#define TX_DISABLE_ERROR_CHECKING
#define TX_DISABLE_PREEMPTION_THRESHOLD
#define TX_DISABLE_NOTIFY_CALLBACKS
#define TX_DISABLE_REDUNDANT_CLEARING
#define TX_DISABLE_STACK_FILLING
#define TX_NOT_INTERRUPTABLE
#define TX_TIMER_PROCESS_IN_ISR
#define TX_REACTIVATE_INLINE
#define TX_INLINE_THREAD_RESUME_SUSPEND



Appendix C FreeRTOS Settings

FreeRTOS was used in its default configuration based on how ST Microelectronics configures it in their STM32CubeIDE tool. I felt that this configuration was pretty consistent and standard with the usage I've seen in the industry and that would be used by a wide variety of developers and teams.

I did review the FreeRTOS Customization documentation, that can be found at:

https://www.freertos.org/Documentation/02-Kernel/03-Supported-devices/02-Customization

The default settings were used. The following page showcases the most interesting settings for this study.

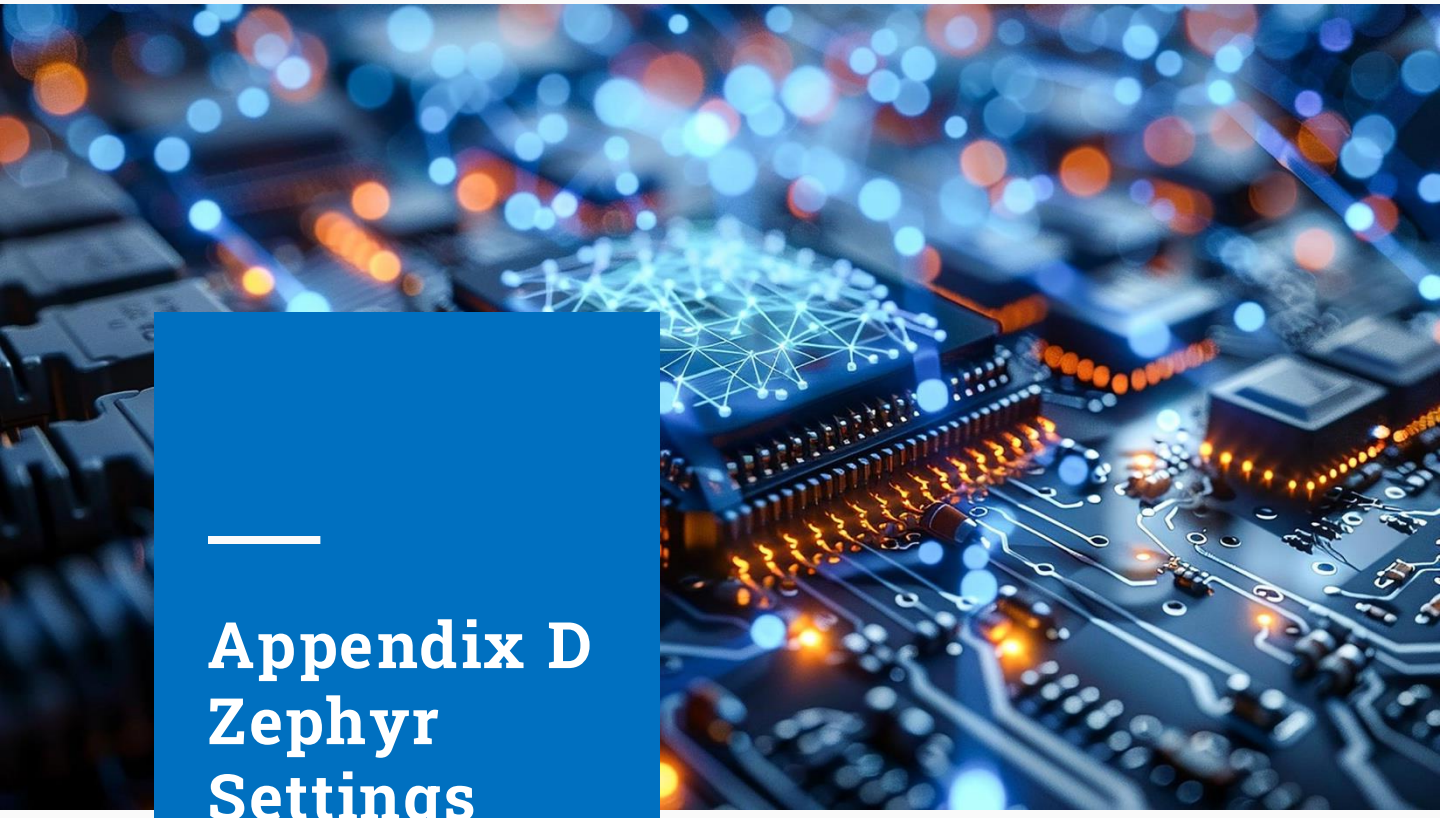
```
#define configUSE_PREEMPTION 1
#define configSUPPORT_STATIC_ALLOCATION 1
#define configSUPPORT_DYNAMIC_ALLOCATION 1
#define configUSE_IDLE_HOOK 0
#define configUSE_TICK_HOOK 0
#define configCPU_CLOCK_HZ ( SystemCoreClock )
#define configTICK_RATE_HZ ((TickType_t)1000)
#define configMAX_PRIORITIES ( 56 )
#define configMINIMAL_STACK_SIZE ((uint16_t)128)
#define configTOTAL_HEAP_SIZE ((size_t)65536)
#define configMAX_TASK_NAME_LEN ( 16 )
#define configUSE_TRACE_FACILITY 1
#define configUSE_16_BIT_TICKS 0
#define configUSE_MUTEXES 1
#define configQUEUE_REGISTRY_SIZE 8
#define configUSE_RECURSIVE_MUTEXES 1
#define configUSE_COUNTING_SEMAPHORES 1
#define configUSE_PORT_OPTIMISED_TASK_SELECTION 0

#define configMESSAGE_BUFFER_LENGTH_TYPE size_t

#define configUSE_NEWLIB_REENTRANT 1

#define INCLUDE_vTaskPrioritySet 1
#define INCLUDE_uxTaskPriorityGet 1
#define INCLUDE_vTaskDelete 1
#define INCLUDE_vTaskCleanUpResources 0
#define INCLUDE_vTaskSuspend 1
#define INCLUDE_vTaskDelayUntil 1
#define INCLUDE_vTaskDelay 1
#define INCLUDE_xTaskGetSchedulerState 1
#define INCLUDE_xTimerPendFunctionCall 1
#define INCLUDE_xQueueGetMutexHolder 1
#define INCLUDE_uxTaskGetStackHighWaterMark 1
#define INCLUDE_xTaskGetCurrentTaskHandle 1
#define INCLUDE_eTaskGetState 1

#define USE_FreeRTOS_HEAP_4
```



Appendix D Zephyr Settings

Zephyr RTOS was configured in its default configuration, but with some adjustments to configuration settings in order to bring it more inline with the default settings for the other RTOSes that were tested. In this configuration, we had the following settings configured in prj.conf:

```
CONFIG_SPEED_OPTIMIZATIONS=y
CONFIG_TIMESLICING=n
CONFIG_SYS_CLOCK_TICKS_PER_SEC=1000
CONFIG_HEAP_MEM_POOL_SIZE=4096
CONFIG_LOG=n
CONFIG_ASSERT=n
CONFIG_DEBUG_OPTIMIZATIONS=n
CONFIG_RUNTIME_ERROR_CHECKS=n
CONFIG_PM=n
```

Note: Without these settings, Zephyr performance is reduced by ~50% !

BENINGO
EMBEDDED GROUP

THANK YOU

Let's Stay Connected

JacobBeningo

Jacob_Beningo

beningoembedded

Website | www.beningo.com
Contact | Jacob@beningo.com

EMBEDDED SOFTWARE ACADEMY
BY BENINGO