MAPUSOFT

WHITE PAPER

Preserve Determinism. Protect Value. Enable the Future.

Deterministic Software Migration for Embedded Systems

Preserve Proven Software.
Migrate with Confidence.

1 **PART 1**
The Embedded Software Modernization Challenge

2 **PART 2**
Deterministic Software Reuse

3 **PART 3**
The Complete MapuSoft Migration Platform

www.mapusoft.com

Table of Contents

Executive Summary

PART 1 – The Embedded Software Modernization Challenge

Every successful embedded product eventually reaches a crossroads: the hardware becomes obsolete long before the software does. Embedded software often represents decades of engineering investment, validation, certification, and field-proven reliability. As organizations migrate to Linux, multicore platforms, and newer RTOSs, preserving that software investment becomes a strategic business objective. Generative AI accelerates development, but operating system migration requires preservation of deterministic runtime behavior rather than regeneration of source code.

PART 2 – Deterministic Software Reuse

Generative AI has become an invaluable tool for software development, accelerating coding, documentation, testing, and developer productivity. However, embedded operating system migration is fundamentally different from software generation. Success depends on preserving deterministic runtime behavior rather than simply producing compilable source code. This section examines where AI provides significant value and why deterministic operating-system abstraction remains the preferred engineering approach for mission-critical embedded systems.

PART 3 – The Complete MapuSoft Migration Platform

Successful embedded software modernization requires more than moving applications from one operating system to another. Organizations need a repeatable engineering methodology that preserves decades of software investment while reducing technical risk, validation effort, and project schedules. The MapuSoft platform addresses this challenge through an integrated suite of technologies centered on deterministic operating-system abstraction. Rather than treating migration as a one-time porting effort, the platform enables continuous software modernization across future processors, RTOSs, and hardware architectures.

Part 1 – The Embedded Software Modernization Challenge

1.1 Why Read This White Paper

Audience: CTOs, architects, engineering directors, program managers, safety engineers, and embedded developers. Readers will learn why embedded migration differs from enterprise modernization, where AI provides value, and why deterministic abstraction reduces migration risk.

1.2 The Growing Value of Embedded Software

Modern embedded software embodies architecture, algorithms, testing, certification, field experience, and maintenance knowledge accumulated over years. These engineering assets frequently exceed the value of the hardware itself.

1.3 Why Modernization Is No Longer Optional

Hardware obsolescence, Linux adoption, multicore processors, cybersecurity, vendor lock-in, supply-chain changes, and long-term support requirements make modernization inevitable while applications continue to deliver business value.

1.4 A Realistic Modernization Scenario

A mature industrial controller with over one million lines of validated software must migrate from an obsolete processor and RTOS to a modern platform. The strategic decision is whether to rewrite and revalidate the application or preserve it through deterministic abstraction.

1.5 The Hidden Cost of Rewriting

Rewriting software recreates engineering knowledge, debugging history, certification evidence, timing behavior, regression testing, field experience, and customer confidence—assets not visible in source code.

1.6 Embedded Systems Are Fundamentally Different

Embedded systems require deterministic scheduling, interrupt responsiveness, synchronization, memory management, and predictable execution in addition to functional correctness.

1.7 Why Determinism Matters

Deterministic execution depends on scheduler semantics, interrupt latency, priority inheritance, jitter, worst-case execution time, and resource management. Preserving these characteristics is essential during migration.

1.8 Characteristics of a Successful Modernization Strategy

Preserve validated software, minimize source changes, reduce regression testing, maintain deterministic behavior, protect certification investments, enable future portability, shorten schedules, and lower lifecycle cost.

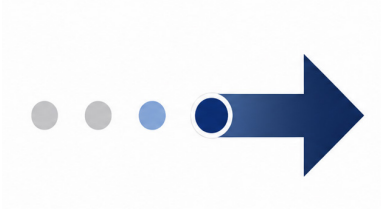
Part 1 Summary

Key takeaways

- **software is the primary long-term asset**
- **modernization is inevitable**
- **rewriting introduces risk**
- **migration is a runtime engineering challenge**
- **deterministic behavior must be preserved during migration.**

Transition to Part 2

Part 2 examines where Generative AI contributes to embedded development and why deterministic operating-system abstraction remains the preferred approach for preserving runtime behavior.



Part 2 – Deterministic Software Reuse

Why Embedded OS Migration Requires More Than Source-Code Translation

2.1 Why This Matters

Operating system migration affects far more than API calls. It influences scheduling, interrupt handling, synchronization, memory management, timing, inter-process communication, and overall system behavior. Organizations must preserve these runtime characteristics while modernizing platforms, making migration an engineering verification problem rather than a code conversion exercise.

2.2 Source Code Is Only One Layer of the System

An embedded application operates within a complete execution environment consisting of the scheduler, interrupt controller, device drivers, timers, synchronization primitives, memory manager, BSP, and processor architecture. Two applications with identical source code can behave differently if these runtime services change.

2.3 Translation vs. Behavioral Preservation

Source-code translation focuses on replacing APIs and generating compilable software. Behavioral preservation focuses on maintaining scheduling semantics, synchronization behavior, timing characteristics, interrupt responsiveness, and deterministic execution. The latter is essential for preserving validated software.

2.4 Where Generative AI Delivers Value

AI excels at documentation, software understanding, legacy code analysis, unit-test generation, build scripts, developer assistance, and repetitive programming tasks. These capabilities improve engineering productivity and should be embraced as complementary technologies.

2.5 Where Deterministic Engineering Remains Essential

Migration requires preserving scheduler semantics, interrupt latency, priority inheritance, synchronization behavior, DMA interactions, cache coherency, memory allocation strategies, and certification evidence. These characteristics are verified through execution rather than inferred from source code.

2.6 Deterministic Software Reuse

Instead of regenerating proven software, deterministic abstraction preserves validated application logic while adapting operating-system services beneath the application. This minimizes application modifications, regression testing, debugging effort, and schedule uncertainty.

2.7 OS Changer® and OS Abstractor®

OS Changer virtualizes operating-system interfaces rather than rewriting applications. At its core, OS Abstractor provides OS-specific implementations engineered to preserve native kernel semantics, deterministic scheduling, synchronization behavior, memory characteristics, robustness, and runtime performance. This architectural approach protects decades of software investment while simplifying future platform migrations.

2.8 Engineering Benefits

Organizations adopting deterministic abstraction benefit from reduced migration risk, lower validation costs, predictable project schedules, preserved application software, improved maintainability, and greater long-term portability.

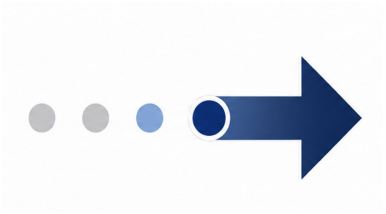
Part 2 Summary

Key takeaways

- **AI improves developer productivity but does not replace runtime verification.**
- **Embedded migration is fundamentally about preserving execution semantics.**
- **Deterministic abstraction minimizes technical risk.**
- **OS-specific abstraction preserves robustness and performance.**
- **Software preservation is more valuable than software regeneration for long-lived embedded products.**

Transition to Part 3

The next section introduces the complete MapuSoft migration platform, demonstrating how OS Changer®, OS Abstractor®, the RTOS Simulator, the Optimized Target Code Generator, and the Application/Platform Profiler work together to provide a comprehensive, repeatable modernization workflow.



Part 3 – The MapuSoft Modernization Platform

From OS Porting to a Repeatable Software Modernization Strategy

3.1 A Platform, Not Just a Porting Tool

Many migration solutions focus exclusively on API conversion. MapuSoft takes a fundamentally different approach.

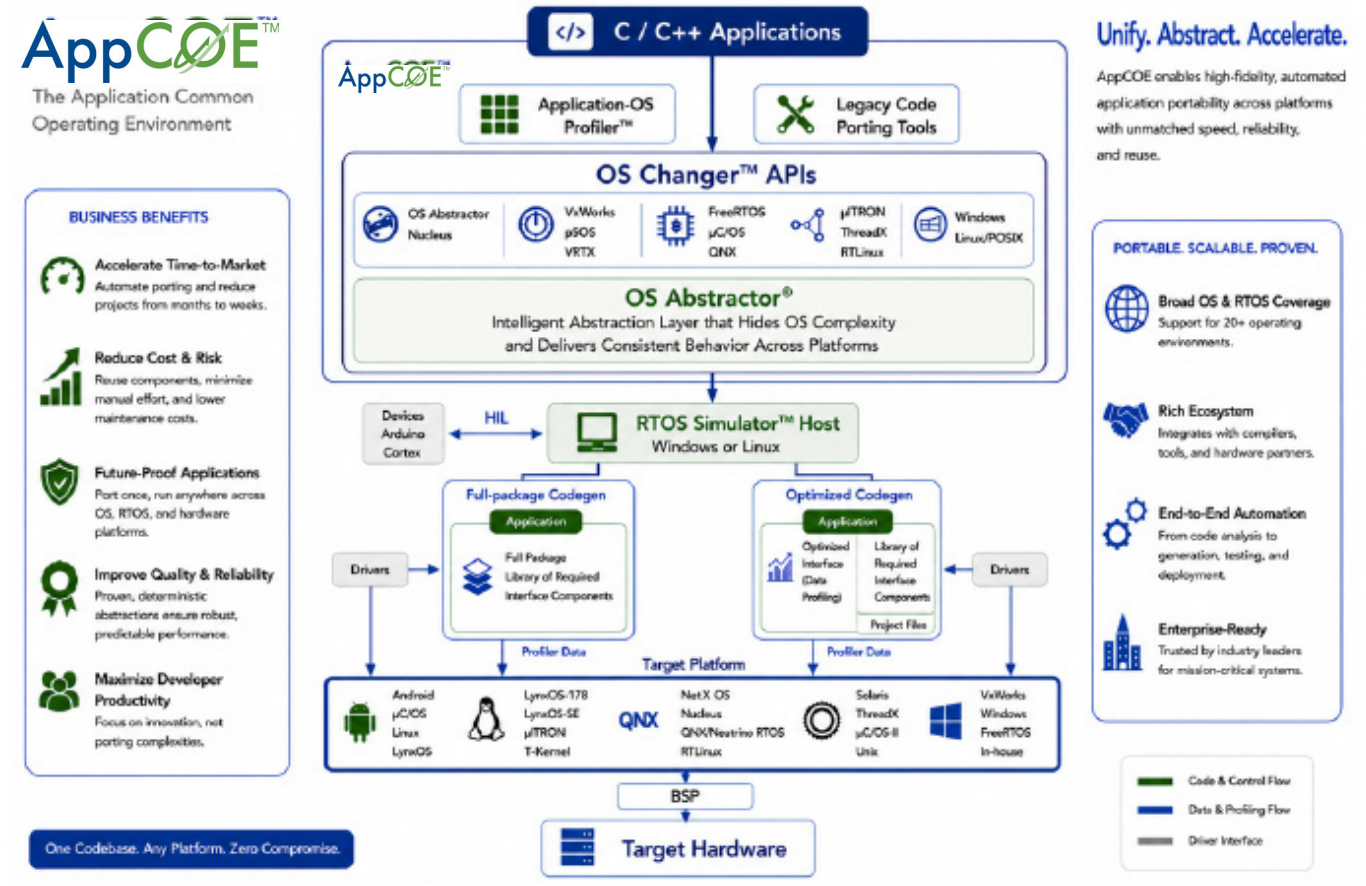


Figure 1 – Complete Modernization Platform Architecture (AppCOE)

Porting toolkit integrates six core components into a unified engineering workflow:

- AppCOE IDE
- OS Changer
- OS Abstractor
- Optimized Target Code Generator
- Application-OS Profiler
- RTOS Simulator

This structured workflow allows organizations to **measure progress** continuously and **identify issues early**. Crucially, it enables engineering teams to **parallelize software and hardware development**, drastically shortening time-to-market. Together, they eliminate schedule uncertainty and minimize engineering risk.

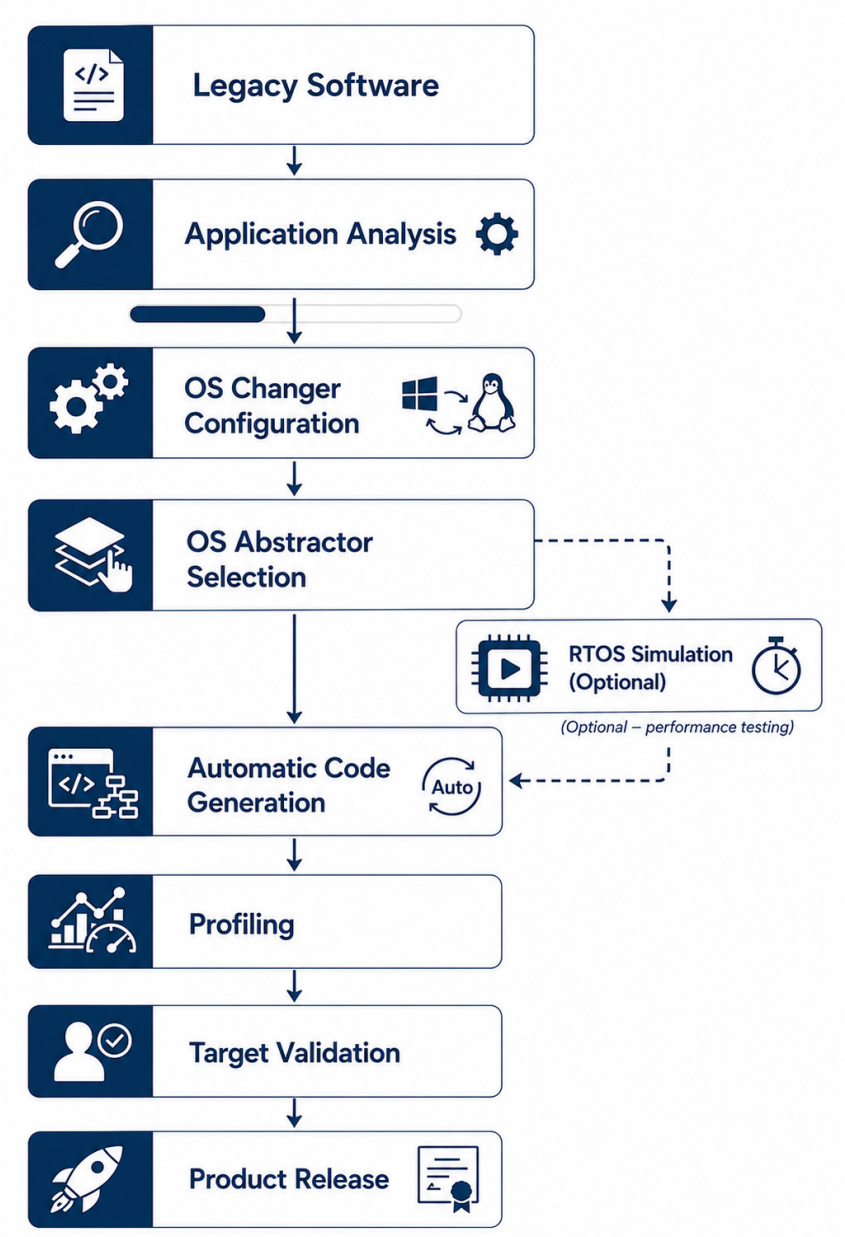


Figure 2 – Modernization Workflow

3.2 OS Changer®: Preserving Proven Applications

OS Changer preserves validated application software by virtualizing operating-system interfaces rather than rewriting source code. Existing applications continue using their original APIs while OS-specific abstraction layers map services onto the destination operating system. This approach protects software architecture, regression tests, certification evidence, debugging history, and field-proven behavior.

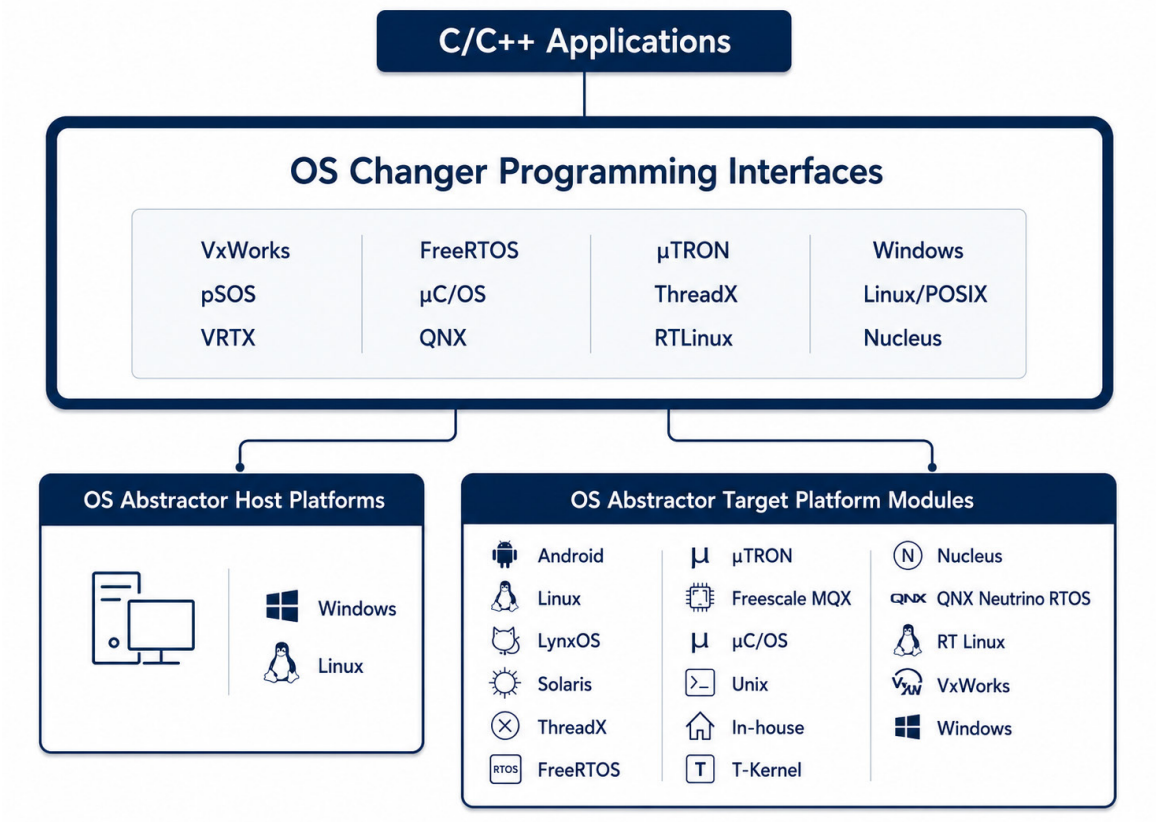


Figure 3 – OS Changer Architecture Block Diagram

3.3 OS Abstractor®: The Core Differentiator

OS Abstractor is the foundation of deterministic portability. Each OS Abstractor module is engineered specifically for an individual target operating system to preserve native scheduling semantics, synchronization behavior, memory management characteristics, interrupt handling, robustness, and runtime performance.

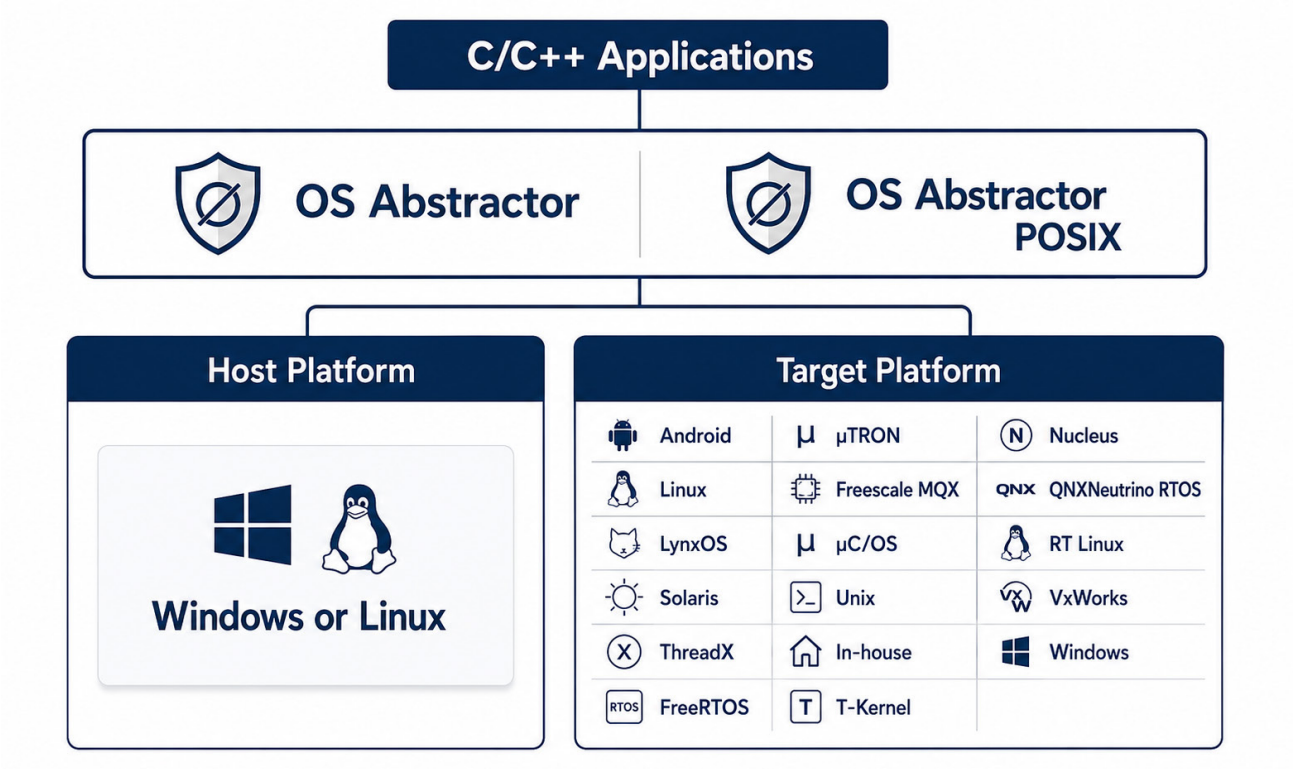


Figure 4 – OS Abstractor Architecture Block Diagram

Key Differentiators

1. Comprehensive OS Abstraction – Beyond a Traditional Wrapper

Unlike conventional wrapper libraries that simply translate one API into another, OS Abstractor Development Platform implements most operating system services internally. As a result, applications remain largely independent of the underlying RTOS.

Only a limited number of services—including priority scheduling, thread priority management, semaphores, messaging, and thread suspend/resume operations—require interaction with the native operating system.

This architecture provides:

- Consistent behavior across supported operating systems
- Improved application portability
- Reduced dependency on OS-specific implementations
- Deterministic runtime behavior

2. Rapid Support for New Operating Systems

OS Abstractor is designed for rapid extensibility. Support for new commercial, proprietary, or in-house operating systems can typically be developed and integrated within approximately two weeks. This enables organizations to adopt new processors, RTOSs, or hardware platforms without lengthy porting projects.

Benefits include:

- Fast adoption of new operating systems
- Reduced engineering effort
- Shorter project schedules
- Lower migration costs

3. Process Support on Any Operating System

Many embedded operating systems do not provide native process isolation or shared memory services. The OS Abstractor Development Platform supplements these capabilities by implementing software-based process management and shared memory support. This enables applications to utilize advanced software architectures without requiring modifications to the underlying RTOS.

Capabilities include:

- Software-based process implementation
- Shared memory support
- Process isolation
- Improved application modularity

4. Device Driver Framework Support

Porting an embedded application to a new operating system often requires significant redevelopment of device drivers due to differences in driver models, initialization sequences, interrupt handling, I/O interfaces, and hardware abstraction mechanisms. These operating system-specific dependencies frequently become one of the largest obstacles to software portability and long-term maintainability.

OS Abstractor® addresses this challenge by providing a flexible Device Driver Framework that establishes a consistent interface between application software and hardware devices while isolating operating system-specific driver implementations. The framework abstracts common driver services and standardizes interactions with peripheral devices, enabling application software to access hardware resources through a stable, operating system-independent interface.

By encapsulating device-specific and operating system-specific functionality within the OS Abstractor, existing device drivers can be adapted to new target platforms with minimal impact on the application. When migrating to a different RTOS or hardware platform, only the underlying driver adaptation layer typically requires modification, while the application and higher-level software components remain unchanged.

By providing a standardized driver architecture, the framework simplifies hardware migration, promotes software reuse, and enables consistent driver behavior across multiple operating systems and processor architectures.

5. Advanced Memory Management

Reliable memory management is critical for mission-critical embedded applications. The platform incorporates deterministic memory allocation techniques that improve system reliability while preventing memory exhaustion.

Key features include:

- Reservation of required system heap during process creation
- Guaranteed availability of required application memory
- Configurable memory usage limits
- Protection against applications consuming excessive system resources
- Deterministic allocation strategies for long-running embedded systems

6. Thread Pooling for Improved Performance

Dynamic thread creation can introduce unnecessary runtime overhead and memory fragmentation. OS Abstractor provides built-in thread pooling to improve performance and system determinism.

Advantages include:

- Elimination of repeated thread creation and deletion
- Reduced CPU overhead
- Improved deterministic execution
- Increased throughput
- Enhanced platform robustness

7. Mission-Critical Fault Recovery

High-availability embedded systems require rapid recovery from unexpected software failures. OS Abstractor Development Platform includes built-in recovery mechanisms that allow applications to recover from software fatal errors without requiring a complete system reboot.

The platform performs a controlled soft reset by restoring the application stack to a known safe execution point, enabling faster recovery and improved system availability.

Benefits include:

- Faster recovery from software failures
- Reduced system downtime
- Increased application availability
- Improved mission-critical reliability

8. Flexible OS API Strategy

OS Abstractor provides exceptional flexibility for developing and migrating applications across multiple operating systems.

Applications may:

- Use one or multiple OS Development Interfaces simultaneously

- Combine applications developed using different operating system APIs
- Execute multiple applications on a single operating system
- Distribute applications across multiple operating systems without significant code changes. This flexibility greatly simplifies software integration during platform migration.

9. Zero-Copy Message Queues

The platform's messaging subsystem supports zero-copy communication, eliminating unnecessary memory copying during message transfers.

This provides:

- Lower CPU utilization
- Reduced communication latency
- Higher messaging throughput
- Improved real-time performance

3.4 Optimized Target Code Generator (OTCG)

Intelligent Generation of Application-Specific Runtime Environments

Traditional portability frameworks typically deliver a generic runtime library that contains implementations for every supported operating system service, regardless of whether an application uses those services. While this approach simplifies portability, it often introduces unnecessary code, increased memory usage, additional initialization overhead, and larger executable images.

The Optimized Target Code Generator (OTCG) addresses these limitations by generating an application-specific implementation of the Cross-OS Development Platform. Instead of deploying a generic abstraction layer, the OTCG analyzes the application's operating system requirements and automatically generates only the components required by the application and the selected target operating system.

Application-Driven Optimization

The optimization process begins with an analysis of the application's operating system interactions. The OTCG identifies every operating system service utilized by the application, including task management, synchronization mechanisms, timers, inter-process communication, memory management, scheduling services, interrupt handling, and other kernel resources.

Based on this analysis, the generator automatically eliminates unused services and produces a customized implementation containing only the required APIs and supporting infrastructure. This application-driven approach significantly reduces the amount of portability code introduced into the final executable.

Target-Specific Code Generation

After determining the application's operating system requirements, the OTCG generates source code specifically tailored for the selected target environment. The generated implementation incorporates operating system-specific API mappings, processor architecture considerations, compiler requirements, and platform configuration parameters.

Rather than maintaining a universal runtime that supports multiple operating systems simultaneously, the generated implementation is specialized for a single deployment target. This specialization minimizes abstraction overhead while preserving complete source-level portability across supported operating systems.

Profile-Guided Runtime Optimization

The OTCG extends beyond static code generation by utilizing execution statistics collected through the Application-OS Profiler. The profiler records how the application interacts with operating system services during execution, including API invocation frequency, execution paths, synchronization behavior, and resource utilization.

Using this profile data, the generator can further optimize each operating system service according to its actual usage characteristics. Frequently executed services can be streamlined to reduce execution overhead, while infrequently used or unnecessary functionality can be minimized or removed entirely. This profile-guided optimization enables the generated runtime to closely match the operational characteristics of the application.

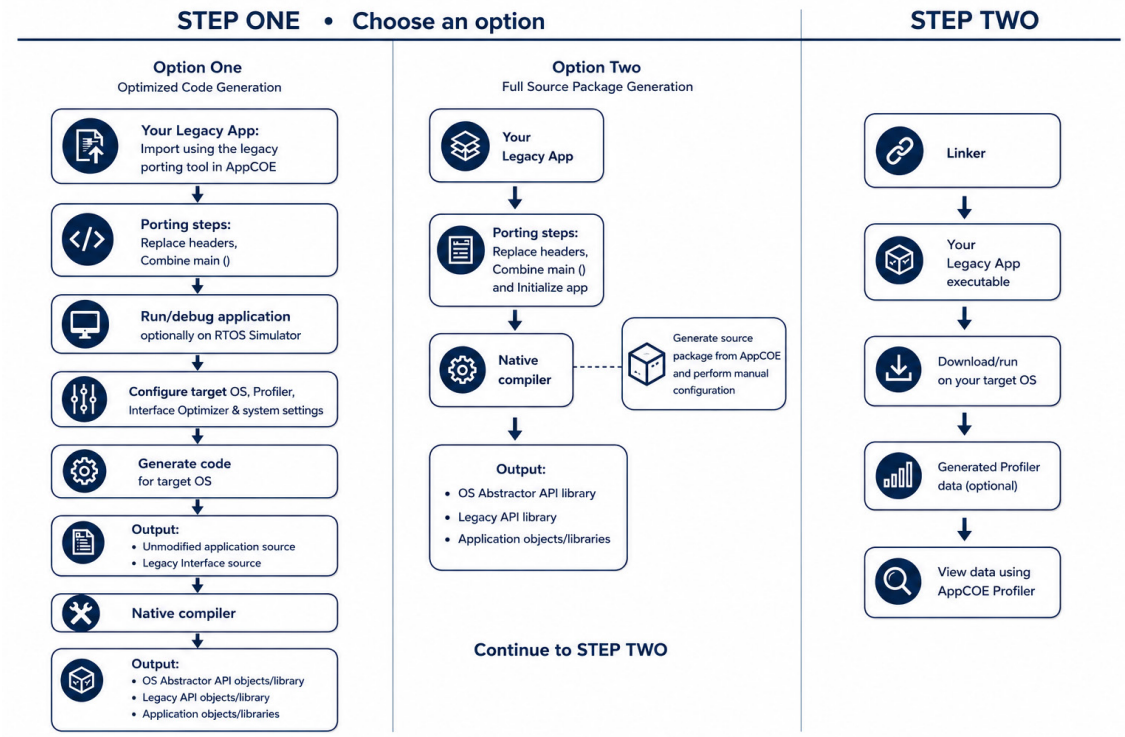


Figure 5 – Optimized Target Code Generation Options

Automatic Resource Configuration

In addition to API optimization, the OTCG automatically generates application-specific operating system configuration parameters.

Instead of relying on conservative default values, the generated configuration reflects the application's actual resource requirements, including:

- Number of tasks and threads
- Semaphore and mutex requirements
- Message queues
- Timer objects
- Memory pools
- Event objects
- Scheduling parameters
- Stack allocations
- System initialization requirements

This automated configuration reduces memory consumption while simplifying system integration and maintenance.

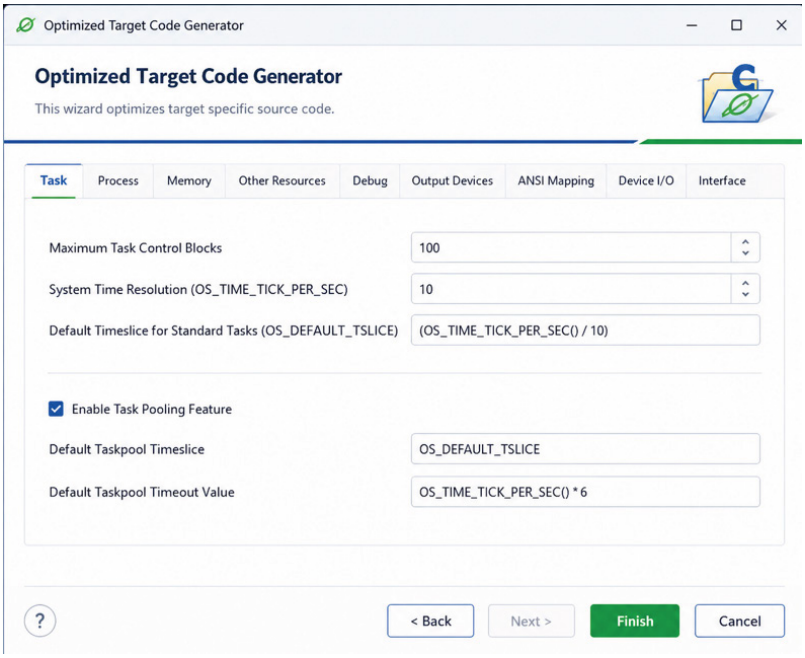


Figure 6 –Optimized Target Code Generation Config Options

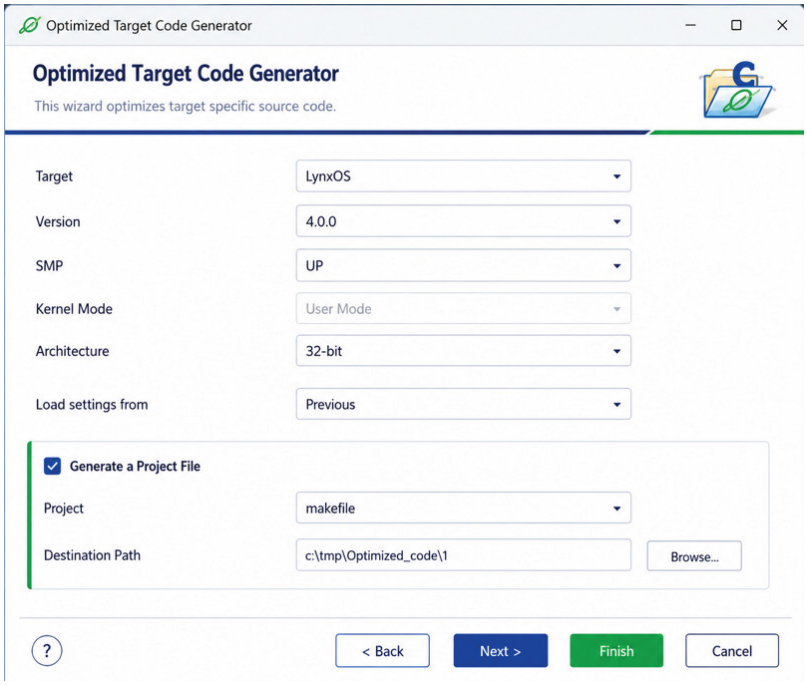


Figure 7 – Optimized Target OS Config Option

Deterministic Code Generation

The OTCG employs a deterministic generation process based on configurable transformation and optimization rules. Given identical application input, target operating system selection, configuration parameters, and optimization settings, the generated implementation remains consistent and repeatable.

Deterministic generation provides significant advantages for regulated industries by simplifying regression testing, certification, verification, configuration management, and long-term maintenance.

Performance and Footprint Benefits

By generating only the functionality required by the application, the OTCG provides measurable improvements compared to traditional portability frameworks. Typical benefits include:

- Reduced executable image size
- Lower RAM utilization
- Faster application initialization
- Reduced portability layer overhead
- Improved execution efficiency
- Shorter build and integration times
- Simplified validation and certification
- Reduced maintenance complexity

The magnitude of these improvements depends on the application's operating system usage patterns, the target operating system, and the selected optimization configuration.

3.5 Integrated Engineering Components

The Optimized Target Code Generator creates only the abstraction components required by the application, minimizing footprint and improving performance. The RTOS Simulator enables development and regression testing before target hardware is available. The Application/Platform Profiler provides visibility into CPU utilization, scheduling latency, interrupt response, memory usage, and overall runtime behavior to verify deterministic execution after migration.

Key Differentiators

The AppCOE Optimized Target Code Generator differs from conventional portability frameworks in several important ways:

Conventional Portability Layer	Optimized Target Code Generator
One-size-fits-all runtime for every application	Runtime tailored specifically for each application
Includes all supported OS services, regardless of use	Generates only the OS services required
Uses a static runtime library	Performs intelligent application analysis and generates optimized target code
Generic resource configuration	Automatically generates application-specific resource configuration
Limited optimization capabilities	Supports static and profile-guided optimization
Relies primarily on the compiler for optimization	Performs optimization before compilation for greater efficiency
Larger memory footprint	Smaller, optimized runtime footprint

3.6 Application-OS Profiler

MapuSoft's Application-OS Profiler is a runtime analysis tool that executes independently on the target platform to collect application performance and operating system usage data without requiring any modifications or instrumentation of the application source code, and without requiring a live connection between the target and the host machine. During execution, it transparently captures detailed runtime information, including OS API utilization, API execution timing, thread activity, and application function performance. The collected profiling data is stored in a file and later imported into AppCOE for comprehensive analysis.

Fully integrated with AppCOE, the Application-OS Profiler identifies application-specific optimization opportunities for the OS Changer framework and OS Abstractor APIs. The profiling results are used by the Optimized Target Code Generator (OTCG) to determine which OS services are required, identify frequently executed APIs, and optimize each OS API according to the application's actual runtime behavior. This profile-guided optimization enables AppCOE to generate highly efficient, application-specific target code that reduces memory footprint, improves execution performance, and delivers more deterministic real-time behavior.

Profiler Differential Report enables developers to compare two application profiling sessions and quantify the impact of software modifications, OS configuration changes, or target code optimizations. By reporting differences in API execution counts, average execution times, minimum and maximum latencies, and cumulative execution times, the report provides objective evidence of improvements or regressions. This capability allows optimization efforts to be validated using measured runtime data, ensuring that generated application-specific code delivers the expected performance gains.

What it does

The profiler instruments the application automatically during execution and records how the application performs and uses the operating system services.

Specifically, it collects:

- OS API usage statistics
  - Which OS APIs are called
  - How often each API is invoked
  - Which threads/tasks are using them
- Execution timing
  - Execution time of every OS API
  - Average execution time
  - Maximum execution time
  - Total cumulative execution time
- Application function profiling
  - User-defined functions can also be profiled, not just MapuSoft APIs. If you need to profile application code that is not organized as functions (e.g. code sections), then the developer can manually insert the enter/exit profiler hook functions around the code
- Thread-level analysis
  - Performance data can be broken down by individual tasks or threads.
- Performance comparison
  - Compare two profiling runs to determine the impact of software changes or optimizations.

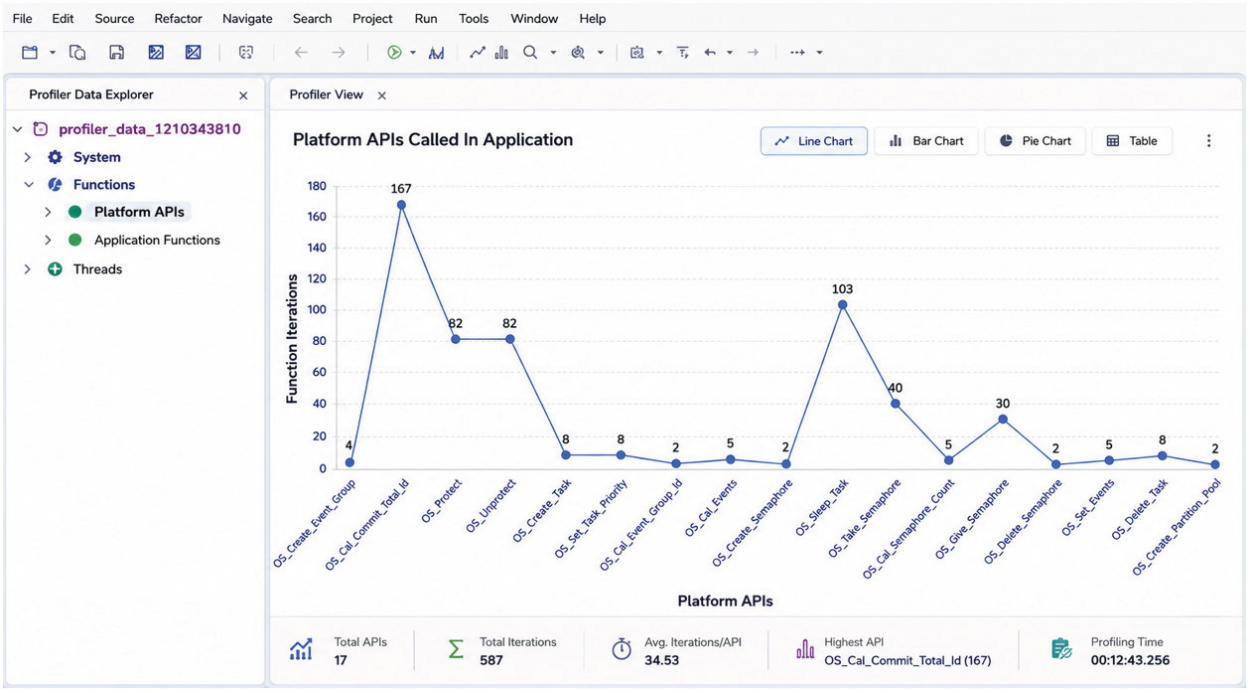


Figure 8 – Execution Count of MapuSoft APIs and Application Functions

Application-OS Profiler Differential Report

Comparison of Two Profiling Sessions

Profile Information	Baseline	Optimized
Project	Flight_Control	Flight_Control
Target OS	VxWorks	VxWorks
CPU	ARM Cortex-R52	ARM Cortex-R52
Profiling Duration	300 sec	300 sec
Generated By	AppCOE Application-OS Profiler	AppCOE Application-OS Profiler

API Timing Comparison

OS API	Calls	Avg Time Before (µs)	Avg Time After (µs)	Improvement
OS_Create_Task	25	18.4	12.7	**31%**
OS_Delete_Task	25	14.8	10.2	**31%**
OS_Create_Semaphore	112	7.3	5.1	**30%**

Figure 9 – Differential Timing Report

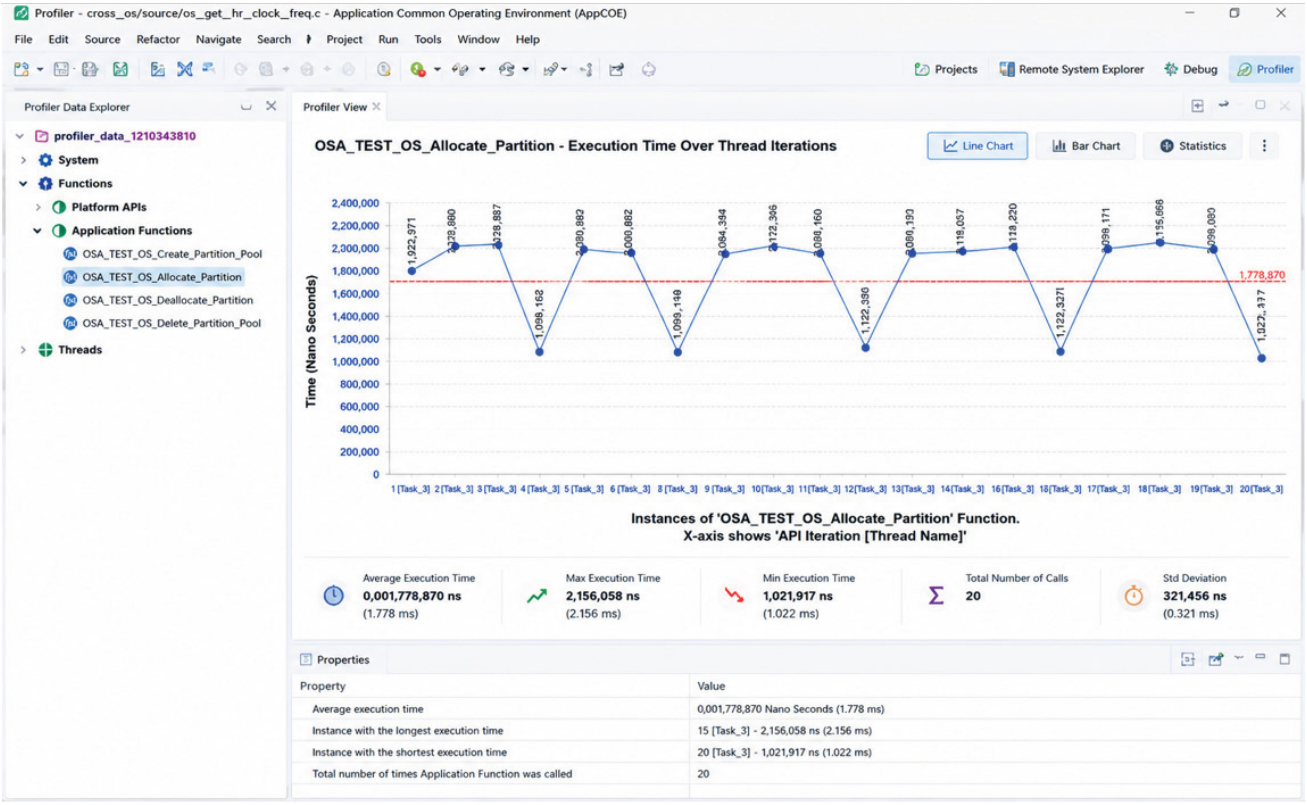


Figure 10 – Execution Time of Application and OS Functions

How the Profiler Works

The Application/Platform Profiler integrates seamlessly into the software modernization workflow with minimal impact on the application development process. Once enabled during the build process, the profiler operates transparently on the target platform, automatically collecting detailed runtime information without requiring modifications to the application source code or a continuous connection to the host development system.

The profiling workflow consists of the following six steps:

1. **Enable the Profiler**

   The Application/Platform Profiler is enabled during the application build process through the AppCOE configuration.

2. **Execute on the Target Platform**

   The application runs normally on the target hardware while the profiler transparently monitors application execution and operating system activity.

3. **Collecting Runtime Measurements**

   Using high-resolution timers, the profiler records every OS Abstractor® API call, operating system service invocation, and selected application function, capturing detailed execution statistics with minimal runtime overhead.

4.Generate the Profiling Data File

During execution, runtime performance information, including execution counts, timing statistics, thread activity, and API utilization—is continuously written to a profiling data file stored on the target system.

5.Import Results into AppCOE

After execution, the profiling data file is loaded into AppCOE®, where the collected runtime information is processed and organized for detailed analysis.

6.Analyze and Optimize

Developers analyze the profiling results using comprehensive graphical reports, including:

- Bar charts
- Line charts
- Pie charts
- Scatter plots
- Execution timelines
- Timing reports
- Function call relationships
- CPU utilization reports
- Differential comparison reports

The collected profiling data can also be supplied directly to the Optimized Target Code Generator (OTCG), allowing AppCOE to optimize OS Abstractor® APIs based on actual runtime usage and generate application-specific target code with improved performance, reduced memory footprint, and more deterministic real-time behavior.

Why it is valuable

For embedded software, optimization should be driven by actual runtime behavior, not assumptions. For example:

Without the Profiler	With the Profiler
Assume semaphore APIs are heavily used	Know exactly how many semaphore calls occur
Guess which APIs are slow	Measure precise execution times
Optimize every OS service equally	Focus only on the APIs that matter
Difficult to justify optimization work	Performance data identifies the highest-impact areas

Figure 11 – Profiler Optimization Comparison

How Application-OS Profiler complements the Optimized Target Code Generator

The profiler becomes especially powerful when used with MapuSoft's Optimized Target Code Generator (OTCG).
A typical workflow is:

1. Analyze the application.
2. Execute the application with the profiler enabled.
3. Collect runtime usage statistics.
4. Identify:
    - frequently used APIs,
    - rarely used APIs,
    - unused services,
    - timing bottlenecks.
5. Feed this information into the Target Code Generator.
6. Generate an application-specific runtime that includes only the required services and optimizes heavily used APIs.

This allows optimization decisions to be based on real execution data rather than static analysis alone.

Key Takeaways

- **Provides visibility into actual runtime behavior.** The Application-OS Profiler captures how an embedded application uses operating system services under real operating conditions, eliminating the need to rely on assumptions or estimates.
- **Enables profile-guided optimization.** Runtime profiling data allows the Optimized Target Code Generator (OTCG) to tailor the generated runtime to the application's actual usage patterns, resulting in more efficient target code.
- **Optimizes OS Abstractor APIs.** By identifying frequently executed APIs and unused services, AppCOE can optimize individual OS Abstractor APIs and eliminate unnecessary runtime components.
- **Improves performance and determinism.** Profile-guided optimization reduces execution overhead, lowers memory consumption, and enhances deterministic real-time behavior—critical requirements for embedded and safety-critical systems.
- **Accelerates performance tuning.** Detailed API timing, thread analysis, and function profiling help developers quickly identify bottlenecks and focus optimization efforts where they have the greatest impact.
- **Validates optimization decisions with measurable data.** Performance comparisons between profiling runs provide objective evidence of improvements, reducing development risk and shortening optimization cycles.
- **Seamlessly integrates with AppCOE.** Profiling data can be loaded into AppCOE to support analysis, visualization, and generation of highly optimized, application-specific runtime code.
- **Complements static analysis.** While static analysis identifies the services an application requires, runtime profiling reveals how those services are used, enabling a higher level of optimization than static analysis alone.

3.7 RTOS Simulator

Virtualizing the Edge: Streamlining Embedded Software with MapuSoft's RTOS Simulator

The modern embedded systems industry faces a challenging paradox: software complexity is skyrocketing due to demanding architectures like pervasive artificial intelligence, while development schedules continue to shrink. Historically, engineers had to wait for expensive, specialized target hardware to be built, configured, and delivered before execution testing could begin. MapuSoft's RTOS Simulator removes this bottleneck by allowing commercial Real-Time Operating System (RTOS) applications to be completely developed, run, and verified on standard host environments.

Architectural Foundations: OS Abstractor and AppCOE

The core technology driving the RTOS Simulator is MapuSoft's OS Abstractor®, a commercial-grade OS Abstraction Layer (OSAL). Unlike layered middleware wrappers that introduce heavy performance penalties, OS Abstractor provides direct API interfaces for numerous operating systems without multi-layered overhead.

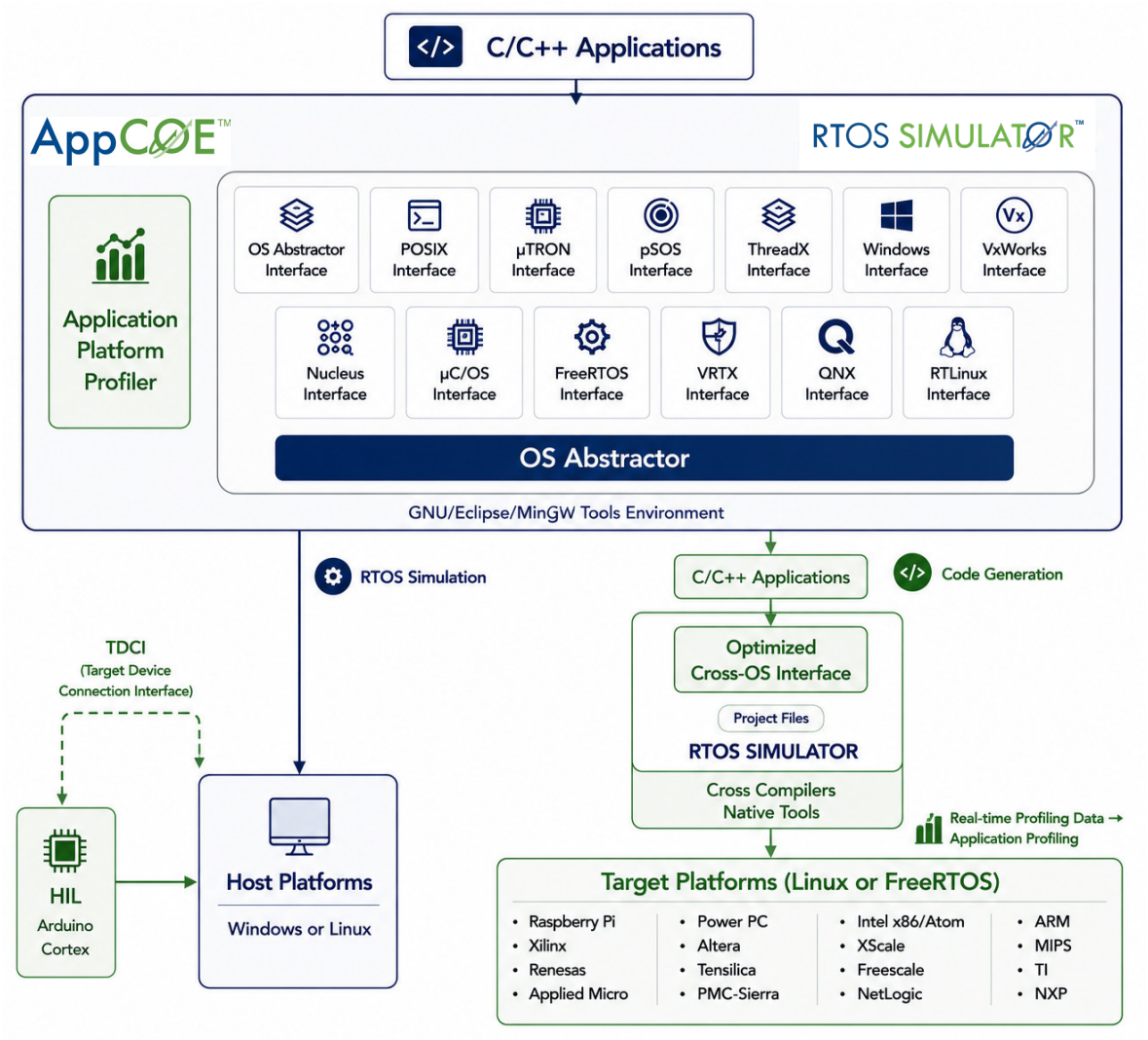


Figure 12 – RTOS Simulator Architecture

The simulator maps these cross-OS APIs directly into a simulated runtime ecosystem. It seamlessly integrates within MapuSoft's **AppCOE™** (Application Common Operating Environment), an Eclipse-based development framework packaged with CDT, BIRT, and GNU x86 tools to deliver a comprehensive integrated development environment (IDE).

Key Operational Features

1. Host-Based Development Platform

RTOS Simulator enables engineers to develop, simulate, and validate real-time applications directly on Windows or Linux host systems. A hardened and optimized host platform, combined with a proprietary OS Abstractor Interface provided in standard object format, accelerates early software development, simulation, testing, and system integration on x86 host architectures without requiring access to RTOS source code. For projects requiring target-native toolchain integration, complete source code libraries are optionally available. The platform also supports Hardware-in-the-Loop (HIL) testing through the Target Device Communication Interface (TDCI), allowing host-based applications to communicate directly with physical target hardware for functional validation, peripheral testing, and early hardware/software integration.

2. Virtualized Testing Bench

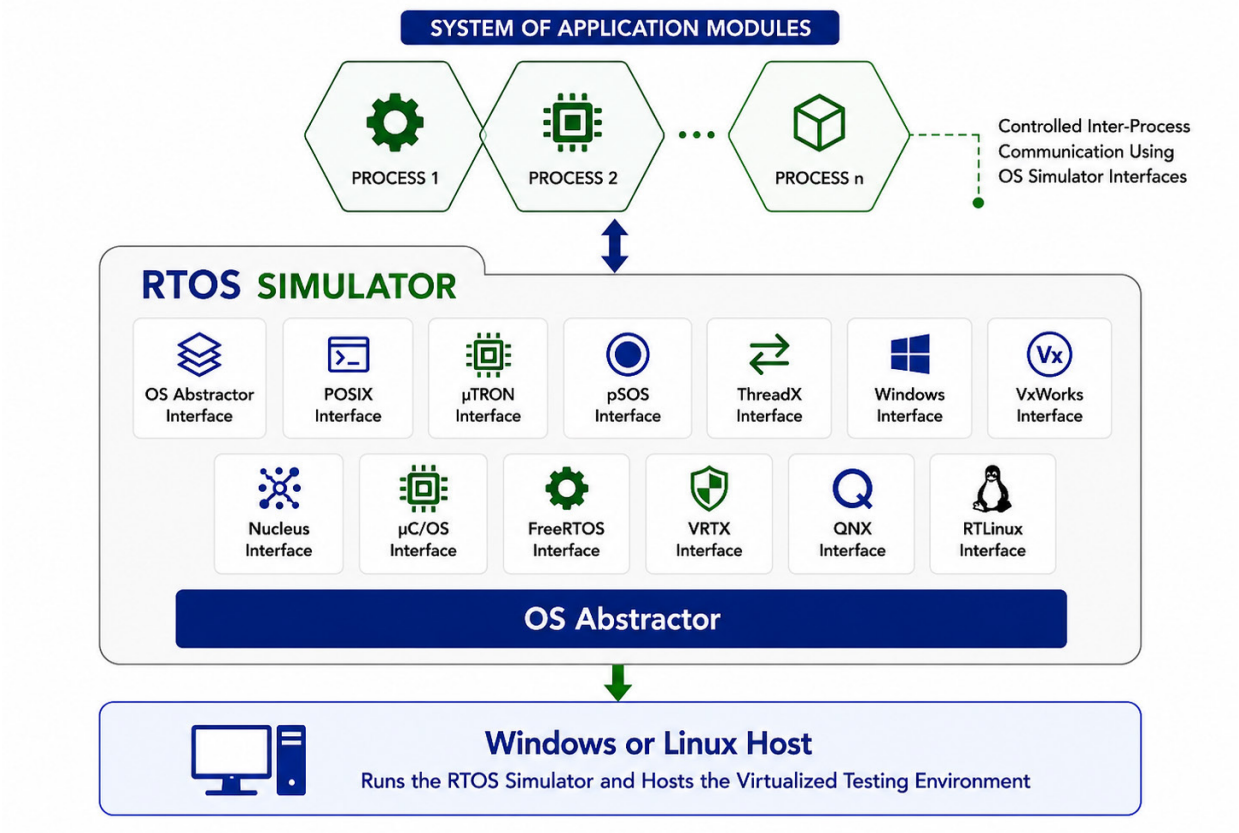


Figure 13 – RTOS Simulator as a Virtualized Testing Platform

The environment supports **modular testing** by breaking applications down into isolated processes. Each process runs with dedicated heap memory and protected kernel resources, preventing a failure in one module from affecting the execution of other modules.

3. Expanding the Simulation: Hardware-in-the-Loop (HIL) Testing

While standard host simulation provides a powerful sandbox for validating application logic and cross-OS API compliance, it operates independently of physical realities. To truly bridge the gap between virtual code and physical environments without sacrificing development speed, MapuSoft incorporates **Hardware-in-the-Loop (HIL) RTOS Simulation.**

HIL simulation inserts real physical components, sensors, and microcontrollers directly into the virtual testing loop. This ensures that real-world timing constraints, raw electrical behaviors, and peripheral data streams are factored into the software testing cycle long before a final commercial system is assembled.

MapuSoft's HIL capabilities are driven by the **Target Device Communication Interface (TDCI)**. This framework sets up a real-time, bi-directional communication pipeline that splits responsibilities cleanly between the high-performance host computer and inexpensive target prototyping hardware:

- **TDCI Host Library:** Resides on the Windows or Linux host machine. It contains Hardware Abstraction Layer (HAL) APIs that support multiple target hardware architectures.
- **Communication Buffer Formats:** Commands sent from the host AppCOE application to the target are strictly structured into efficient packets. These packets contain specific function codes, parameter counts, data types, sizes, and values. To accommodate boards with limited memory buffers, TDCI can automatically split large commands into multiple chunks.
- **TDCI Target Module:** A lightweight target monitor program that resides directly on the physical target hardware (such as an **ARM Cortex-M4** or **Arduino board**). The monitor continuously listens on the communication interface (typically UART) for Hardware Abstraction Layer (HAL) command requests issued by applications running within AppCOE. Upon receiving a request, it executes the corresponding hardware operation on the target device and returns the execution status, output data, or console messages back to the AppCOE application running on the host.

4. Practical Application: Pervasive AI and Smart Systems

MapuSoft's HIL setup is particularly useful for building edge devices that utilize **Pervasive AI**—embedded systems that constantly sense, predict, and adapt with minimal human intervention. By keeping a physical microcontroller in the loop, developers can feed real-world analog noise into lightweight Python-based Machine Learning models (like scikit-learn Decision Trees or Random Forests) running concurrently on the host.

5. Educational Utility: RTOS Simulator for Academics

Because commercial RTOS licenses and physical target suites often strain university budgets, MapuSoft offers an **Academic Edition**. This package allows students to learn real-time programming architectures on their personal laptops without specialized hardware.

The academic bundle includes custom embedded lab exercises, complete lab manuals, and a dedicated **Raspberry Pi support package**. Students can generate optimized hypervisor source code hosted on free environments like Linux or FreeRTOS, building production-grade embedded expertise at a minimal price point.

RTOS Simulator - Quantifiable Benefits

By decoupling software maturation from physical hardware cycles, the RTOS Simulator delivers clear engineering advantages:

- **Shorter Time to Market:** Software development and integration testing begin much earlier, completely bypassing hardware procurement lead times.
- **Reduced Licensing Costs:** Teams eliminate the need to purchase individual commercial RTOS seats for host-side application testing.
- **Deep Performance Profiling:** Built-in **Application-OS Profiler** graph toolsets allow developers to visually map task execution times, monitor thread iteration counts, and resolve bottlenecks before writing code to a physical chip.

Business Outcomes

Organizations adopting the MapuSoft platform can protect software intellectual property, reduce migration risk, shorten development schedules, lower verification costs, accelerate time-to-market, reduce vendor lock-in, extend product lifecycles, and lower total cost of ownership. Software modernization becomes a repeatable engineering capability rather than a disruptive redevelopment effort. These outcomes allow organizations to transform software modernization from a high-risk redevelopment effort into a predictable, repeatable engineering process.

Future-Proofing Embedded Software

Once operating-system dependencies have been isolated through deterministic abstraction, future hardware refreshes, RTOS upgrades, multicore migration, virtualization, and platform diversification become significantly easier. Applications become portable engineering assets capable of evolving with changing technologies instead of being tied to a single operating system.

Executive Perspective

The value of embedded software continues to grow while hardware lifecycles become shorter. Organizations that preserve software rather than regenerate it gain a strategic advantage through predictable modernization, lower engineering costs, and improved product agility. Deterministic software abstraction transforms operating-system migration into a sustainable long-term modernization strategy.