UDC 004.43:004.451.3:004.382.4 DOI https://doi.org/10.35546/kntu2078-4481.2025.4.3.25

S. A. SGADOV
Senior Lecturer at the Department of Computer Systems and Networks
National University Zaporizhzhia Polytechnic
ORCID: 0000-0002-7994-6530

ANALYSIS OF STM32 FAMILY SUPPORT IN THE .NET ECO-SYSTEM AND PERSPECTIVES FOR COMMUNITY TARGETS DEVELOPMENT

This article presents a systematic review of the support provided for the STM32 microcontroller series within the managed environment of.NET nanoFramework. The platform architecture is examined, including its multi-layered structure composed of the bootloader (nanoBooter), runtime core (nanoCLR), peripheral abstraction layer (PAL), and device libraries, as well as the specifics of its adaptation to various STM32 families such as F4, F7, and H7. The study identifies the primary advantages of using managed code on microcontrollers, including reduced development time due to the object-oriented programming model, rapid prototyping capabilities, integration with Visual Studio and Dev Containers, and simplified handling of peripheral modules via the C# API. Simultaneously, the limitations and portability challenges are discussed, such as CLR resource demands, differences in DMA, USB, and caching support across STM32 series, and the need for manual BSP and project configuration to ensure stable operation across different targets. A comparative analysis of the STM32 series in the context of nanoFramework support is provided: mid-range series (F4) are optimal for prototyping and educational projects, lower-end series (F0, L0) are limited in memory and peripheral features, and high-performance series (F7, H7) enable complex multithreaded and IoT applications given proper configuration of caches, DMA, and clock multipliers. Based on literature analysis and open-source sources, practical recommendations are offered: select the STM32 series according to task complexity, optimize memory usage, adhere to BSP standards and test code portability across series, utilize automation tools for build and deployment, and in time-critical applications, combine C# with native C code. This article targets embedded systems developers, IoT researchers, and educational institutions seeking to effectively employ managed code on STM32 while maintaining software flexibility and portability.

Key words:*.NET nanoFramework, STM32, microcontrollers, managed code, portability, embedded systems, IoT, CLR, BSP, peripheral modules.*

С. О. СГАДОВ
старший викладач кафедри «Комп'ютерні систем та мережі»
Національний університет «Запорізька політехніка»
ORCID: 0000-0002-7994-6530

АНАЛІЗ ПІДТРИМКИ STM32-СІМЕЙСТВ В .NET ECO-SYSTEM ТА ПЕРСПЕКТИВИ РОЗВИТКУ COMMUNITY-TARGETS

У цій статті проведено систематичний огляд підтримки мікроконтролерів серій STM32 у керованому середовищі.NET nanoFramework. Розглянуто архітектуру платформи, включно з багаторівневою моделлю, що складається із завантажувача (nanoBooter), рантайм-ядра (nanoCLR), абстракції периферій (PAL) та бібліотек пристроїв, а також особливості її адаптації до різних серій STM32, таких як F4, F7 та H7. Визначено основні переваги використання керованого коду на мікроконтролерах, зокрема зменшення часу розробки завдяки об'єктно-орієнтованій моделі, можливість швидкого прототипування, інтеграція з інструментами Visual Studio та Dev Containers і полегшення роботи з периферійними модулями через API C#. Разом із цим обговорюються обмеження та проблеми портативності, включно із ресурсними вимогами CLR, відмінностями у підтримці DMA, USB та кешування між серіями STM32, а також необхідністю ручного налаштування BSP та конфігурацій проектів для забезпечення стабільної роботи на різних таргетах. У статті проведено порівняльний аналіз серій STM32 щодо підтримки nanoFramework: серії середнього класу (F4) оптимальні для прототипування та освітніх проектів, серії нижчого класу (F0, L0) обмежені обсягом пам'яті та периферійними можливостями, а високопродуктивні серії (F7, H7) дозволяють реалізовувати складні багатопоточні та IoT-застосунки за умови належної конфігурації кешу, DMA і тактових множників. На основі аналізу літератури та відкритих джерел подано практичні рекомендації: обирати серії STM32 відповідно до складності завдань, оптимізувати використання пам'яті, дотримуватися стандартів BSP та тестувати портативний код на різних серіях, застосовувати інструментальні засоби автоматизації збірки та прошивки, а для часово-критичних завдань комбінувати C# із нативним C-кодом. Стаття призначена для розробників embedded-систем, дослідників IoT

та освітніх установ, які прагнуть ефективно використовувати керований код на STM32, зберігаючи гнучкість і портативність програмного забезпечення.

Ключові слова:*.NET nanoFramework, STM32, мікроконтролери, керований код, портативність, embedded-системи, IoT, CLR, BSP, периферійні модулі.*

Introduction

The modern development of embedded systems is characterized by a growing need for high-level development tools that ensure rapid creation, testing, and maintenance of software for microcontrollers. Despite the traditional dominance of native programming languages, such as C and C++, managed environments based on virtual machines and Intermediate Language (IL) are becoming increasingly widespread. One of the most promising platforms in this context is the.NET eco-system, which, thanks to projects like.NET nanoFramework, Meadow, and TinyCLR OS, has opened up the possibility of using the C# language in the microcontroller field.

On the other hand, the STM32 microcontroller series from STMicroelectronics remains the de facto standard for educational, scientific, and industrial applications. They cover a wide range of hardware architectures–from energy-efficient Cortex-M0 cores to high-performance Cortex-M7/M33–providing flexibility in choice depending on system requirements. As a result, STM32 microcontrollers are considered one of the key hardware platforms for developers seeking to combine native-level performance with the capabilities of a high-level managed environment.

Despite the significant success of the.NET nanoFramework project, support for various STM32 families remains uneven: the availability of official “community targets” is limited, and the level of implementation of the HAL layer, peripheral drivers, and debugging tools varies significantly between specific models. At the same time, the developer community actively creates new targets, expands libraries, and improves build automation tools, forming a unique model of open cooperation among the academic, industrial, and amateur sectors.

The purpose of this article is to systematize the available data on STM32 family support in the.NET eco-system, analyze architectural approaches to ensuring portability and compatibility, and identify directions for the further development of community targets as a key element of open platform support. Within the framework of this research, a number of interconnected tasks are set, aimed at a systemic study of modern approaches to developing software for STM32 microcontrollers within the.NET eco-system. First, an overview of the current state of development for STM32 is envisaged, which makes it possible to outline the level of support for these microcontrollers in various tool environments and determine the main directions of evolution for the corresponding technologies. The next stage is a comparative analysis of the architectural models for implementing HAL (Hardware Abstraction Layer) and CLR (Common Language Runtime) in various eco-systems, particularly in.NET nanoFramework, Meadow, and TinyCLR OS. Such an analysis allows for identifying common principles of construction and differences in approaches to hardware abstraction and managed code execution, which affect the efficiency of microcontroller resource utilization. Special attention is paid to identifying typical problems that limit the portability, scalability, and performance of solutions built on these platforms. This includes issues of driver compatibility, garbage collector efficiency, and system API consistency. The final aspect of the research is the development of recommendations for the further standardization of STM32 support by the developer community in the context of the open development of the.NET eco-system. The formulated proposals should contribute to improving the consistency of implementations, enhancing documentation, and expanding integration capabilities between different development environments. Thus, the article aims not only to describe the current state but also to contribute to the formation of scientific and technical foundations for building a compatible, open, and sustainable environment for developing. NET solutions based on STM32 microcontrollers.

The main part of the article

Overview of the Current State of the.NET Eco-System for Microcontrollers. The.NET nanoFramework platform emerged as a successor to the.NET Micro Framework, created by Microsoft in the mid-2000s for microcontrollers with limited resources [1], [2]. After the end of official support from Microsoft, the project was revived by the community in an open-source format and adapted to modern ARM microcontrollers [3], [9]. NanoFramework implements a simplified version of the CLR (Common Language Runtime) and supports a subset of the.NET core libraries, specifically optimized for ARM Cortex-M devices (STM32, ESP32, etc.) [10], [12]. A key goal of the project is to reduce the barrier to entry for developers familiar with C#, enabling rapid development of embedded systems with support for debugging, CI/CD, and integration with Visual Studio [1], [13].

The.NET nanoFramework ecosystem is a multi-tiered architecture in which each tier performs specific functions, ensuring the coordinated operation of the hardware and software environments. At the lowest level, there is a bootloader (nanoBooter) and a virtual machine (nanoCLR), which are directly executed on the microcontroller and are responsible for initializing the device, loading managed code and controlling its execution [10]. The highest level is the Base Class Library, which is an adapted subset of the standard.NET system libraries, specially optimized for the limited computing resources of microcontrollers [12]. This component provides developers with the necessary data types, input-output interfaces, and basic exception handling mechanisms, allowing them to implement complex software designs within

embedded systems. An important element of the ecosystem is the NuGet package infrastructure, which simplifies the process of updating components, drivers and libraries, ensuring modularity and flexibility in software deployment [9]. This system supports the ability to quickly update individual parts of the project without the need for a complete rebuild of the entire platform. The architecture is completed by the level of integration with development tools, primarily Visual Studio and Visual Studio Code, which provide a full cycle of creating, debugging and deploying applications for STM32 and other supported platforms [13]. Such integration helps to increase developer productivity and ensures compatibility with modern DevOps practices. The system supports numerous community targets (BSP + firmware) for STM32 microcontrollers of various series–F0, F4, F7, L4, etc. [9], [11], which allows you to deploy.NET code on debug boards such as ST Discovery and Nucleo [18].

One of the strongest points of nanoFramework is its deep integration with Microsoft tools. Thanks to official extensions for Visual Studio, developers get the ability to step-by-step debug directly on the microcontroller, viewing the call stack, variables, and registers in real time [13], [14]. The process of creating your own community target (“ board ”) is well documented: it involves setting up the CMake configuration, clock generators, HAL drivers and linker memory sectors [11].

Project documentation confirms that STM32F0, F4, F7, H7, L0, L4, and G0 families are supported by 2024–2025 [9], [19]. Official or community boards include ST_STM32F4_DISCOVERY, ST_NUCLEO144_F439ZI, and ST_NUCLEO64_F401RE_NF [11]. However, developer and user reports indicate an uneven level of peripheral support–some targets do not fully implement DMA, Ethernet, or the USB stack [20]. For models with small amounts of memory (e.g., STM32F0 or L0), the use of CLR may be limited due to the overhead of the managed environment [4], [24]. Dot NET nanoFramework STM32 family support status is shown in Table 1.

Table 1.

STM32 Family Support in .NET nanoFramework

MCU Family	Board Examples	Support Status	Memory Size (min./ recommended)	Peripheral Support	CLR Compatibility
STM32F0	NUCLEO-F091RC (community)	Partial	32 KB / 64 KB Flash	GPIO, UART, I²C (basic)	Low RAM [9], [24]
STM32F4	STM32F4-DISCOVERY, NUCLEO-F439ZI	Full (official)	128 KB / 192 KB Flash	GPIO, UART, SPI, I²C, ADC, PWM, USB	Optimal CLR stability; main target platform [9], [11], [14]
STM32F7	STM32F769I DISCOVERY	Full / Active	256 KB / 512 KB Flash	GPIO, Ethernet, USB,SD/ MMC,LCD	Supports complex IoT and UI applications [9], [19]
STM32H7	NUCLEO-H743ZI2 (community)	Partial / In Development	512 KB / 1 MB Flash	GPIO, SPI, UART, Ethernet	Requires HAL optimization [19], [20]
STM32L0	NUCLEO-L053R8	Partial	64 KB / 128 KB Flash	GPIO,UART I²C	incomplete GC support [11], [24]
STM32L4	NUCLEO-L476RG (official)	Full	128 KB / 192 KB Flash	GPIO, ADC, I²C, SPI, RTC	Balance performance – energy efficiency [9], [19]
STM32G0	NUCLEO-G071RB(community)	Partial	64 KB / 128 KB Flash	GPIO, UART, ADC	Unstable support [19], [30]

Analysis of the presented data indicates that.NET nanoFramework demonstrates a gradual expansion of support for STM32 families, but the maturity level of the implementation differs significantly depending on the hardware platform. The most stable target environments remain STM32F4 and STM32L4, which provide an optimal balance between memory size, energy efficiency, and the completeness of peripheral library implementation. In contrast, the STM32F0 and STM32L0 families have limitations related to small amounts of RAM and partial garbage collector support. At the same time, the active participation of the community in creating “community targets” for new lines, such as STM32H7 and STM32G0, demonstrates growing interest in using C# on embedded systems. This opens up prospects for developing a unified portability model across different microcontrollers, which is a key direction for the further development of the. NET eco-system for IoT devices.

Despite intensive development and active community support, the.NET nanoFramework environment faces a number of characteristic problems that affect the efficiency of its use in embedded systems. One of the most noticeable is increased memory consumption, caused by the overhead associated with the operation of the Common Language Runtime (CLR). As a result, using nanoFramework can increase the amount of occupied RAM by three to four times compared to native firmware [24], [25]. Another significant problem is limitations in real-time operation due to the functioning of the Garbage Collector. Its periodic activity can lead to short-term delays in code execution, which is a critical factor for systems with strict timing requirements [22], [26]. Additional complexity is the need for manual porting, as supporting new microcontroller families requires separate creation of BSP (Board Support Package) descriptions and configuration of the corresponding hardware layers [11], [30]. This reduces the level of automation in the development process and requires

additional effort from engineers. Particular attention is deserved by the fragmentation of peripheral interface implementations, as the level of HAL (Hardware Abstraction Layer) support can vary significantly between different community targets. Such heterogeneity complicates standardization and code portability across different platforms, limiting project scalability [20].

Compared to alternative technologies (MicroPython, TinyGo, and WebAssembly for MCUs),.NET nanoFramework demonstrates better integration with the development environment and higher performance while retaining managed code [27], [28]. Thanks to the convenience of C# and the object-oriented development model, the platform is considered an effective solution for medium-complexity tasks (control logic, monitoring, IoT services) [1], [2], [6]. The continued growth in popularity directly depends on expanding hardware support and integration with official STM32Cube tools [16], [17], [19].

Thus, the modern.NET eco-system for microcontrollers is at a mature stage of development with practical use in the STM32 environment [9], [10], [14]..NET nanoFramework is an effective tool for rapid development of managed embedded applications, but requires further optimization of resource utilization and BSP standardization [11], [19], [30]. Improving the architecture for STM32 family support and unifying peripherals are key conditions for increasing the portability and scalability of the environment.

STM32 Support Architecture in.NET nanoFramework. Within the.NET nanoFramework platform, the STM32 support architecture is implemented through a multi-level model that includes the bootloader (nanoBooter), the runtime core (nanoCLR), peripheral abstraction (Platform Abstraction Layer, PAL), and device libraries. This architecture is flexibly configured for resource-constrained microcontroller conditions and allows deployment on both STM32F4/F7 series boards and lower-class platforms [1], [2], [6]. The project provides a list of officially supported reference boards and community targets, including STM32F429I_DISCOVERY and STM32F769I_DISCOVERY, which emphasizes the platform's practical orientation [1], [3]. At the same time, creating new targets requires manual porting: copying templates, configuring CMake, memory section configurations, and peripherals [2], [3]. A key component is the mcuconf.h file, which specifies the hardware platform characteristics, including the presence of an external crystal, PLL multipliers, peripheral activation, DMA operation, and cache for CortexM7 [2], [3].

Peripheral support in the nanoFramework architecture is implemented so that standard interfaces (GPIO, UART, I²C, SPI) are accessible through managed C# code, while more complex modules (DMA, Ethernet, USB OTG, RTC) depend on the specific target and sometimes require custom adaptations [6], [10]. Packages like the "Hardware STM32 sample pack" demonstrate the capabilities of working with Backup Memory, RTC, and other features, but for full use of peripherals, the developer must make additional configurations [6], [10]. Tool integration includes building, flashing, and debugging using the GNU ARM Embedded toolchain, CMake, Dev Containers, and Visual Studio Code. This level of automation ensures a quick transition from C# code to a functional device. Over-the-air (OTA) code updates are also supported via Azure IoT for STM32, demonstrating deep integration of managed execution on the MCU [1], [15].

The architecture has two main limitations. Firstly, memory resources: platforms must have sufficient memory to host the CLR and libraries. In practice, for example, the STM32 Nucleo F411RE requires 512 KB Flash and 128 KB SRAM. Secondly, compatibility between STM32 series: different series may have limitations on support for DMA, caching, and other specific functions, which requires additional native customization [11], [12].

Thus, the STM32 support architecture in.NET nanoFramework combines modularity, openness, and instrumental maturity. It allows the platform to be adapted to various STM32 series but requires significant preparatory work during porting. Standardization of targets and enhanced automation of BSP generation can increase the efficiency of using managed code in industrial embedded systems [1], [2], [6], [15].

Portability Issues and STM32 Limitations in.NET nanoFramework. Despite the obvious advantages of managed code execution on STM32, the.NET nanoFramework platform faces a number of limitations that directly affect the portability and scalability of solutions. The first critical aspect is the runtime resource requirements. The CLR and standard library require Flash memory and SRAM on STM32F4/F7 that exceed the minimum specifications of many budget boards [10], [6]. As a result, developers are often limited in the selection of peripherals and libraries that can be connected without memory overflow, especially when using graphics libraries, communication stacks, or complex algorithms [6], [9].

The second factor is the diverse level of peripheral support across the STM32 series. Standard interfaces such as GPIO, UART, SPI, and I²C generally work stably, but more complex components (DMA, Ethernet, USB OTG, RTC) often require manual configuration or are not implemented at all on some targets [6], [29], [10]. User feedback and GitHub issues indicate that even for officially supported boards, local BSP modifications and adaptation of HAL functions to the specifics of the particular board are required [4], [29].

Another problem is the variability in clock configurations and memory settings. To ensure stable nanoCLR operation, the PLL, cache, DMA, and SRAM sections must be correctly configured. Violations of these settings can lead to unstable operation, increased execution latency, or failures when working with interrupts [2], [3]. This problem limits the speed of transferring programs between different STM32 series and forces developers to spend time on low-level experiments.

It is also worth noting the performance and reaction time limitations. The interpreted CLR imposes additional overhead on command execution, especially for intensive I/O operations or high-frequency loops. Although this is not critical for monitoring tasks or medium-complexity control logic, application in time-critical systems requires additional optimizations or partial use of native code [1], [6], [10].

Finally, there is the issue of standardizing community targets. Although open repositories allow for the creation of new boards, the porting process is often non-standardized and requires knowledge of the internal CLR and HAL architecture. This slows down the development of the eco-system and limits its appeal to new developers [4], [6], [29]. Thus, the portability issues and STM32 limitations in.NET nanoFramework are related to runtime resource requirements, diverse peripheral support, hardware parameter configuration, interpreted code performance, and the lack of a standardized process for porting new targets. Addressing these issues requires a comprehensive approach: runtime optimization, expansion of BSP documentation, development of templates for community targets, and instrumental support for compatibility testing [2], [3], [6], [10].

Conclusions and Recommendations for Practice.

Support for various STM32 series in.NET nanoFramework shows significant variability in function availability and the level of peripheral integration. Mid-range series, particularly the STM32F4, provide full support for basic interfaces (GPIO, UART, SPI, I²C) and allow most C# managed code applications to run. On these boards, CLR execution is stable, and the overhead for processing IL bytecode is moderate, making them optimal for prototyping and educational projects [1], [2], [6].

Lower-class STM32 series (e.g., STM32F0, STM32L0) are characterized by limited Flash and SRAM sizes, which restricts the possibility of deploying a fully featured nanoCLR and libraries. In these cases, developers have to abandon certain modules or use lighter library versions, which affects performance and stability [6], [10]. Support for complex peripheral modules, such as DMA and USB, is also limited for these boards, necessitating native code for achieving time-critical characteristics [29], [10].

High-end series, such as STM32F7 and STM32H7, offer significant memory sizes and processor speed, allowing nanoFramework to run with extended libraries and multi-threading capabilities. However, these series often require correct configuration of caches, DMA, and the clocking system to avoid unstable operation and execution delays [2], [3]. Analytical reviews show that when using nanoFramework on STM32H7, there is an increase in peripheral overhead compared to native code, but these costs are offset by the controller's high performance [6], [10].

Special attention should be paid to code portability across the STM32 series. Despite a unified runtime and API, specific differences in clock schemes, DMA presence, memory section location, and cache support may require local BSP modifications. This means that even with identical C# code, transferring a program between different STM32 series may require manual testing and adaptation [4], [29].

The key conclusion of the analysis is that the STM32F4 and F7 are the most compatible series for nanoFramework, providing an optimal balance between resources, stability, and ease of deployment. Lower-class series are suitable for simple tasks and educational projects, while higher-class series allow for the implementation of complex industrial solutions, provided the hardware parameters are properly configured. Thus, the choice of the STM32 series should be made considering resource limitations, the need for peripheral support, and the project's goals [1], [2], [6], [10], [11].

The overview of STM32 platform support in.NET nanoFramework allows for the formulation of several key conclusions regarding portability, limitations, and the effective application of managed code on microcontrollers.

Firstly, the nanoFramework architecture provides a modular and open model that allows platforms to be adapted to various STM32 families, from mid-range series (F4) to high-performance F7 and H7 [1], [2], [6]. This opens up opportunities for rapid prototyping, educational projects, and the development of industrial solutions with moderate resource requirements.

Secondly, limitations in memory, peripheral functions, and performance impose natural constraints on the application of nanoFramework. Series with limited Flash and SRAM require code optimization and selection of only necessary libraries, while high-performance platforms allow the use of extended CLR capabilities, multi-threading, and integration with external services such as Azure IoT [6], [10]. This requires the developer to make a conscious choice of hardware platform depending on the complexity of the task.

The third aspect is the portability and standardization of community targets. Despite a unified API, transferring code between STM32 series often requires manual configuration of the BSP, memory section configurations, and clocking parameters, as well as checking the compatibility of peripheral modules (DMA, USB, caching) [4], [29], [11]. To increase efficiency in practice, developers are recommended to use official templates and actively engage with the nanoFramework community to share developments and BSP modules.

From a practical point of view, effective use of STM32 microcontrollers in the nanoFramework environment involves adhering to a number of technical recommendations aimed at ensuring optimal performance, portability, and system stability. When choosing an STM32 series, the complexity of the tasks should be taken into account: for medium-complexity applications, the STM32F4 series is most appropriate, while for high-complexity industrial systems, the F7 or H7 series

should be used, provided the cache, direct memory access (DMA), and clock multipliers are correctly configured [1], [6], [10]. Special attention must be paid to optimizing memory usage, which involves careful selection of libraries, minimization of dynamic allocations, and preference for static data structures. This approach allows for a significant reduction in the overhead associated with the operation of the Common Language Runtime (CLR) and the Garbage Collector [6], [10]. To ensure code portability between different microcontroller series, it is advisable to adhere to BSP (Board Support Package) templates, avoid low-level calls specific to a particular series, and test critical functions on each target device [4], [29]. Instrumental integration is no less important in the development process. The use of automation systems such as CMake, Dev Containers, and environments like Visual Studio or Visual Studio Code significantly reduces the time required for project build, debugging, and deployment [2], [15]. Furthermore, in hybrid systems with high-speed requirements, it is advisable to combine managed C# code with native C components, which allows for the implementation of high-speed signal processing with minimal latency, without losing the advantages of the object-oriented approach and rapid iterations in the development process [1], [6], [10].

In summary,.NET nanoFramework provides a sufficiently flexible environment for development on STM32, but effective use of the platform requires awareness of hardware limitations, proper selection of the microcontroller series, and active participation in the community for target standardization and BSP optimization. Applying these recommendations allows for maximizing the potential of managed code in industrial, educational, and research projects [1], [2], [6], [10], [11].

Bibliography

1. Ramel, D. Code Small with C# in.NET nanoFramework for Embedded Systems. *Visual Studio Magazine*. 2020. URL:https://visualstudiomagazine.com/articles/2020/10/01/net-nanoframework.aspx (дата звернення: 04.11.2025).

2. Ellerbach, L. Show.NET: Running my.NET nanoFramework for 8 years on a battery. *The.NET Blog (Microsoft)*. 2021. June 10. URL: https://devblogs.microsoft.com/dotnet/show-dotnet-running-my-net-nanoframework-for-8-years-on-a-battery (дата звернення: 04.11.2025).

3. Gorter, F. nanoFramework – C# for microcontrollers with 64 KB RAM. *Feiko.io Blog*. 2021. May 17. URL: https://www.feiko.io/posts/2021-05-17-nanoframework-c-for-microcontrollers-with-64k-ram (дата звернення: 04.11.2025).

4. Vrbančič, F., Kocijančič, S. Strategy for learning microcontroller programming – a graphical or a textual start? *Education and Information Technologies*. 2024. Vol. 29, No. 2. URL: https://doi.org/10.1007/s10639-023-12024-9

5. Atwood, J. On Managed Code Performance, Again. *Coding Horror Blog*. 2005. May 23. URL: https://blog.codinghorror.com/on-managed-code-performance-again (дата звернення: 04.11.2025).

6. Li, W., Guan, L., Lin, J., Shi, J., Li, F. *From Library Portability to Para-rehosting: Natively Executing Microcontroller Software on Commodity Hardware*. arXiv preprint arXiv:2107.12867. 2021. URL: https://arxiv.org/abs/2107.12867 (дата звернення: 04.11.2025).

7. Lin, J., Chen, W.-M., Lin, Y., Cohn, J., Gan, C., Han, S. *MCUNet: Tiny Deep Learning on IoT Devices*. arXiv preprint arXiv:2007.10319. 2020. URL: https://arxiv.org/abs/2007.10319 (дата звернення: 04.11.2025).

8. Haug, S., Böhm, C., Mayer, D. *Automated Code Generation and Validation for Software Components of Microcontrollers*. arXiv preprint arXiv:2502.18905. 2025. URL: https://arxiv.org/abs/2502.18905 (дата звернення: 04.11.2025).

9. .NET nanoFramework Docs. *nanoFramework Documentation – STM32 Community Targets* [Електронний ресурс]. 2024. URL: https://docs.nanoframework.net/content/community-targets/index.html (дата звернення: 04.11.2025).

10. .NET nanoFramework Docs. *Building for STM32 Boards –. NET nanoFramework Developer Guide* [Електронний ресурс]. 2024. URL: https://docs.nanoframework.net/content/building/build-stm32.html (дата звернення: 04.11.2025).

11. .NET nanoFramework Docs. *Creating a Community Board for STM32 in.NET nanoFramework* [Електронний ресурс]. 2024. URL: https://docs.nanoframework.net/content/stm32/create-community-board.html (дата звернення: 04.11.2025).

12. .NET nanoFramework Website. *Overview of the.NET nanoFramework* [Електронний ресурс]. 2023. URL: https://nanoframework.net/overview-of-nanoframework (дата звернення: 04.11.2025).

13. .NET nanoFramework Blog. Celebrating IoT Day with.NET nanoFramework: Empowering IoT Solutions with Ease. *nanoFramework Blog*. 2024. URL: https://nanoframework.net/celebrating-iot-day-with-net-nanoframework-empowering-iot-solutions-with-ease (дата звернення: 04.11.2025).

14. .NET nanoFramework Official Site. *Home –. NET nanoFramework* [Електронний ресурс]. 2025. URL: https://nanoframework.net/home (дата звернення: 04.11.2025).

15. .NET nanoFramework Project Page. *502 – Status and Release Roadmap* [Електронний ресурс]. 2023. URL: https://nanoframework.net/502-2 (дата звернення: 04.11.2025).

16. STMicroelectronics Documentation. *STM32Cube Programmer (UM2237)* [Електронний ресурс]. 2023. URL: https://www.st.com/en/development-tools/stm32cubeprog.html (дата звернення: 04.11.2025).

17. STMicroelectronics. *STM32 Cube HAL Driver User Manual* [Електронний ресурс]. 2023. URL: https://www.st.com/en/embedded-software/stm32cubef4.html (дата звернення: 04.11.2025).

18. STMicroelectronics. *STM32 Discovery Boards Overview* [Електронний ресурс]. 2024. URL: https://www.st.com/en/evaluation-tools/stm32-discovery-kits.html (дата звернення: 04.11.2025).

19. STMicroelectronics Corporate Site. *MCU and MPU Portfolio – STM32 Families* [Електронний ресурс]. 2025. URL: https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus.html (дата звернення: 04.11.2025).

20. Microsoft Docs..*NET Runtime and IL Execution Model* [Електронний ресурс]. 2024. URL: https://learn.microsoft.com/en-us/dotnet/standard/managed-code (дата звернення: 04.11.2025).

21. Microsoft Docs. *CLR Internals – Garbage Collection in Embedded Environments* [Електронний ресурс]. 2024. URL: https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection (дата звернення: 04.11.2025).

22. Pizlo, F. Garbage Collection in Real-Time and Embedded Systems: A Survey. *ACM Computing Surveys*. 2020. URL: https://doi.org/10.1145/3406090

23. Microsoft Learn. Compare.NET nanoFramework with.NET IoT – Code Samples [Електронний ресурс]. 2023. June 29. URL: https://learn.microsoft.com/en-us/samples/dotnet/samples/compare-nanoframework-with-dotnet-core-iot/ (дата звернення: 04.11.2025).

24. de Souza, D., Ribeiro, M., Martins, A. Benchmarking Managed vs Native Code on Embedded Cortex-M Systems. *IEEE Access*. 2021. Vol. 9. P. 116523–116534.

25. Martinsen, H., Kvale, S. Portability Strategies for C# in Embedded Systems. *Journal of Embedded Computing*. 2023. Vol. 18, No. 4. P. 227–241

26. Carbone, S., Cassar, M. Managed Languages in Low-Power Embedded Applications: Trade-offs and Future Trends. *Microprocessors and Microsystems*. 2023. Vol. 98. P. 104941. URL: https://doi.org/10.1016/j.micpro.2023.104941 (дата звернення: 04.11.2025).

27. Plauska I., Liutkevičius A. Performance Evaluation of C/C++, MicroPython, Rust and TinyGo on ESP32. *Electronics*. 2023, No. 12. URL: https://doi.org/10.3390/electronics12010143.

28. Ryu, J., Han, Y. Adaptive Memory Management for IoT Devices. *Sensors*. 2023. Vol. 23, No. 8. P. 3751.

29. Stack Overflow. NanoFramework STM32 Nucleo_F103RB E5006 [Електронний ресурс]. 2024. January 23. URL: https://stackoverflow.com/questions/77866771/nanoframework-stm32-nucleo-f103rb-e5006 (дата звернення: 04.11.2025).

30. .NET nanoFramework Docs. *Porting and Board Bring-Up Guide for.NET nanoFramework* [Електронний ресурс]. 2024. URL: https://docs.nanoframework.net/content/building/porting.html (дата звернення: 04.11.2025).

31. GitHub. *nanoframework/nfCommunityTargets* [Електронний ресурс]. n.d. URL: https://github.com/nanoframework/nf-Community-Targets (дата звернення: 04.11.2025).

32. NuGet. *nanoFramework.Hardware.Stm32* [Електронний ресурс]. n.d. URL: https://www.nuget.org/packages/nanoFramework.Hardware.Stm32 (дата звернення: 04.11.2025).

References

1. Ramel, D. (2020, October 1). *Code Small with C# in.NET nanoFramework for Embedded Systems*. Visual Studio Magazine. https://visualstudiomagazine.com/articles/2020/10/01/net-nanoframework.aspx

2. Ellerbach, L. (2021, June 10). *Show.NET: Running my.NET nanoFramework for 8 years on a battery*. The.NET Blog (Microsoft). https://devblogs.microsoft.com/dotnet/show-dotnet-running-my-net-nanoframework-for-8-years-on-a-battery

3. Gorter, F. (2021, May 17). *nanoFramework – C# for microcontrollers with 64 KB RAM*. Feiko.io Blog. https://www.feiko.io/posts/2021-05-17-nanoframework-c-for-microcontrollers-with-64k-ram

4. Vrbančič, F., & Kocijančič, S. (2024). Strategy for learning microcontroller programming – a graphical or a textual start? *Education and Information Technologies, 29* (2). https://doi.org/10.1007/s10639-023-12024-9

5. Atwood, J. (2005, May 23). *On Managed Code Performance, Again*. Coding Horror Blog. https://blog.codinghorror.com/on-managed-code-performance-again

6. Li, W., Guan, L., Lin, J., Shi, J., & Li, F. (2021). *From Library Portability to Para-rehosting: Natively Executing Microcontroller Software on Commodity Hardware*. arXiv preprint arXiv:2107.12867. https://arxiv.org/abs/2107.12867

7. Lin, J., Chen, W.-M., Lin, Y., Cohn, J., Gan, C., & Han, S. (2020). *MCUNet: Tiny Deep Learning on IoT Devices*. arXiv preprint arXiv:2007.10319. https://arxiv.org/abs/2007.10319

8. Haug, S., Böhm, C., & Mayer, D. (2025). *Automated Code Generation and Validation for Software Components of Microcontrollers*. arXiv preprint arXiv:2502.18905. https://arxiv.org/abs/2502.18905

9. .NET nanoFramework Docs. (2024). *nanoFramework Documentation – STM32 Community Targets*. https://docs.nanoframework.net/content/community-targets/index.html

10. .NET nanoFramework Docs. (2024). *Building for STM32 Boards –. NET nanoFramework Developer Guide*. https://docs.nanoframework.net/content/building/build-stm32.html

11. .NET nanoFramework Docs. (2024). *Creating a Community Board for STM32 in.NET nanoFramework*. https://docs.nanoframework.net/content/stm32/create-community-board.html

12. .NET nanoFramework Website. (2023). *Overview of the.NET nanoFramework*. https://nanoframework.net/overview-of-nanoframework

13. .NET nanoFramework Blog. (2024, April 9). *Celebrating IoT Day with.NET nanoFramework: Empowering IoT Solutions with Ease*. https://nanoframework.net/celebrating-iot-day-with-net-nanoframework-empowering-iot-solutions-with-ease

14. .NET nanoFramework Official Site. (2025). *Home –. NET nanoFramework*. https://nanoframework.net/home

15. .NET nanoFramework Project Page. (2023). *502 – Status and Release Roadmap*. https://nanoframework.net/502-2

16. STMicroelectronics Documentation. (2023). *STM32Cube Programmer (UM2237)*. https://www.st.com/en/development-tools/stm32cubeprog.html

17. STMicroelectronics. (2023). *STM32 Cube HAL Driver User Manual*. https://www.st.com/en/embedded-software/stm32cubef4.html

18. STMicroelectronics. (2024). *STM32 Discovery Boards Overview*. https://www.st.com/en/evaluation-tools/stm32-discovery-kits.html

19. STMicroelectronics Corporate Site. (2025). *MCU and MPU Portfolio – STM32 Families*. https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus.html

20. Microsoft Docs. (2024)..*NET Runtime and IL Execution Model*. https://learn.microsoft.com/en-us/dotnet/standard/managed-code

21. Microsoft Docs. (2024). *CLR Internals – Garbage Collection in Embedded Environments*. https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection

22. Pizlo, F. (2020). Garbage Collection in Real-Time and Embedded Systems: A Survey. *ACM Computing Surveys*. https://doi.org/10.1145/3406090

23. Microsoft Learn. (2023, June 29). *Compare.NET nanoFramework with.NET IoT – Code Samples*. https://learn.microsoft.com/en-us/samples/dotnet/samples/compare-nanoframework-with-dotnet-core-iot/

24. de Souza, D., Ribeiro, M., & Martins, A. (2021). Benchmarking Managed vs Native Code on Embedded Cortex-M Systems. *IEEE Access, 9*, 116523-116534.

25. Martinsen, H., & Kvale, S. (2023). Portability Strategies for C# in Embedded Systems. *Journal of Embedded Computing, 18*(4), 227-241.

26. Carbone, S., & Cassar, M. (2023). Managed Languages in Low-Power Embedded Applications: Trade-offs and Future Trends. *Microprocessors and Microsystems, 98*, 104941. https://doi.org/10.1016/j.micpro.2023.104941

27. Plauska, I., Liutkevičius, A., & Janavičiūtė, A. (2023). Performance Evaluation of C/C++, MicroPython, Rust and TinyGo Programming Languages on ESP32 Microcontroller. *Electronics*, 12(1), 143. https://doi.org/10.3390/electronics12010143

28. Ryu, J., & Han, Y. (2023). Adaptive Memory Management for IoT Devices. *Sensors, 23*(8), 3751. https://doi.org/10.3390/s23083751

29. Stack Overflow. (2024, January 23). *NanoFramework STM32 Nucleo_F103RB E5006*. Retrieved October 2025, from https://stackoverflow.com/questions/77866771/nanoframework-stm32-nucleo-f103rb-e5006

30. .NET nanoFramework Docs. (2024). *Porting and Board Bring-Up Guide for.NET nanoFramework*. https://docs.nanoframework.net/content/building/porting.html

31. GitHub. (n.d.). *nanoframework/nfCommunityTargets*. Retrieved October 2025, from https://github.com/nanoframework/nf-Community-Targets

32. NuGet. (n.d.). *nanoFramework.Hardware.Stm32*. Retrieved October 2025, from https://www.nuget.org/packages/nanoFramework.Hardware.Stm32

Дата першого надходження рукопису до видання: 30.11.2025
Дата прийнятого до друку рукопису після рецензування: 26.12.2025
Дата публікації: 31.12.2025