Date of publication DEC-31, 2025, date of current version DEC-24, 2025.
www.computingonline.net / computing@computingonline.net

Print ISSN 1727-6209
Online ISSN 2312-5381
DOI 10.47839/ijc.24.4.4339

A Modular Framework for High-Performance Graphical Interfaces on STM32 Microcontrollers

OLEKSANDR STELMAKH, INNA V. STETSENKO, ANTON DYFUCHYN, ALEXANDER ZARICHKOVYI, OLEKSANDRA DYFUCHYNA

[1]Igor Sikorsky Kyiv Polytechnic Institute, 37, Prospect Beresteiskyi, Solomyanskyi district, Kyiv, 03056, Ukraine

Corresponding author: Oleksandr Stelmakh (e-mail: stelmahwork@gmail.com).

ABSTRACT This paper presents a modular framework for STM32 microcontrollers designed for high-performance graphical user interfaces. The proposed framework combines direct low-level hardware access with a structured, template-based configuration approach in modern C++, aiming to reduce the overhead commonly associated with universal abstraction layers while preserving architectural clarity and type safety. The framework architecture is organized into two layers: a low-level hardware interaction layer based on standardized microcontroller interfaces, and a higher-level declarative configuration layer that simplifies peripheral setup and system integration. This design enables predictable behavior, improved resource efficiency, and adaptability across different STM32 microcontroller families. The performance of the proposed solution was evaluated on a representative STM32 development platform using a widely adopted embedded graphics library. Experimental results demonstrate a substantial improvement in rendering efficiency and memory utilization compared to a reference implementation, while maintaining stable graphical output under varying workloads. In addition, the framework supports flexible rendering pipelines, predefined peripheral configurations, and integration with a real-time operating system for multitasking graphical applications. The obtained results indicate that the proposed approach provides an efficient and specialized alternative to general-purpose embedded frameworks for resource-constrained graphical systems.

KEYWORDS STM32; embedded systems; graphical user interface; light and versatile embedded graphics library (LVGL); common microcontroller software interface standard (CMSIS); free real-time operating system (FreeRTOS); configuration framework; performance optimization

I. INTRODUCTION

The development of software for microcontrollers increasingly faces the challenge of simultaneously achieving high performance, efficient utilization of limited hardware resources, and support for scalable architectural solutions [1]. These requirements are especially critical in projects involving complex graphical user interfaces. In such systems, not only rendering speed but also precise control over peripheral components, such as display controllers, memory interfaces, and touch sensors, plays a vital role.

Traditional approaches to embedded systems design are typically based on either low-level programming with direct register access or on high-level frameworks with automated code generation. Direct register access ensures high control and efficiency but requires significant architectural effort. In contrast, high-level frameworks abstract hardware details, often at the expense of flexibility and scalability. As a result, developers are frequently forced to make trade-offs. These trade-offs typically involve performance, transparency, code readability, and component reusability.

This paper introduces a novel approach designed to overcome these compromises through the development of a specialized framework called STM Components (STMCMP) specifically oriented toward embedded graphical applications. The STMCMP architecture combines low-level efficiency with modern C++ design principles. It relies on direct CMSIS-based register manipulation and template-based modular configuration. The framework aims to deliver high performance and type safety. In addition, it supports extensibility across various STM32 families.

Within this study, we present an experimental performance comparison between the proposed solution and a reference LVGL port for the STM32F746G-DISCO board. The benchmark results allow for a quantitative assessment of the

benefits that STMCMP offers in the context of embedded graphical systems development.

This work addresses the problem of high resource consumption and execution overhead in graphical user interface implementations on STM32 microcontrollers. The goal of this study is to evaluate how system-level architectural decisions at the CMSIS layer influence graphical performance and memory usage. To achieve this goal, a modular CMSIS-based framework is designed and experimentally evaluated using the LVGL graphics library on the STM32F746 platform, demonstrating measurable improvements in frame rate and resource utilization compared to a reference implementation.

II. LITERATURE REVIEW

In the field of software development for STM32 microcontrollers, various approaches and frameworks are used, with the most common being HAL, Arduino, Mbed OS, and libraries for graphical user interface development.

The Hardware Abstraction Layer (HAL) by STMicroelectronics provides a high-level interface for interacting with STM32 hardware resources, simplifying the configuration of General-Purpose Input/Output (GPIO), timers, Universal Asynchronous Receiver-Transmitter (UART), and other modules. However, the general-purpose nature of HAL introduces additional overhead, which may reduce performance and increase firmware size. This is particularly critical for resource-constrained systems.

In paper [2], a method for automated code generation and validation based on HAL for the STM32F407 is proposed. The approach relies on Abstract Syntax Trees (AST) and the Retrieval-Augmented Generation (RAG) mechanism. The approach involves building an AST to analyze code and employing RAG to synthesize missing HAL functions based on contextual information. The generated code is validated through compilation, automated testing, and simulation in the Renode environment. The authors demonstrate that their method can effectively extend HAL interfaces without developer intervention, thus automating critical development stages.

The concept of hardware abstraction is also analyzed in work [3], where the author provides a conceptual comparison of direct peripheral access via specific APIs and the use of vendor-provided HAL layers. The paper focuses on performance, testability, and scalability, highlighting key architectural trade-offs in embedded software design. Although not an academic publication, this source is widely referenced in engineering practice and applied research in the embedded systems domain.

A detailed description of low-level peripheral programming without the Hardware Abstraction Layer (HAL) is presented in the monograph [4], which serves as a comprehensive textbook on embedded system development for ARM Cortex-M based microcontrollers. The author covers both assembly and C programming for peripheral configuration, including GPIO, timers, and UART modules. Special attention is paid to direct register-level control without HAL. This resource is particularly valuable for comparing manual hardware setup with abstracted frameworks, demonstrating the practical relevance of the low-level approach for systems with critical timing constraints.

Recent research has revisited the design requirements for hardware abstraction layers in embedded systems, emphasizing the trade-offs between portability, interface stability, and execution overhead. In particular, modern studies highlight that poorly designed abstraction boundaries may compromise deterministic behavior and runtime performance on resource-constrained platforms [28]. These observations align with the motivation for CMSIS-based approaches, which provide standardized low-level interfaces while preserving explicit control over microcontroller peripherals.

Performance optimization at the processor level has also been actively studied for ARM Cortex-M microcontrollers. Recent works demonstrate that memory hierarchy utilization, pipeline behavior, and instruction-level optimizations significantly affect execution efficiency and real-time determinism on Cortex-M cores [29]. This further supports the need for low-level control mechanisms that avoid unnecessary abstraction overhead in performance-critical embedded applications.

Arduino is a popular framework known for its simplicity and large user community. However, on STM32 platforms, Arduino faces limitations, as many of its libraries are designed for AVR architecture and may not function reliably on other hardware. The research [5] presents a thorough bibliometric analysis of Arduino-based microcontroller used in scientific research from 2008 to 2022 included in the Scopus database. The authors examined 1,122 publications covering topics in physics, chemistry, biology, STEM education, and automation. Using the VOSviewer software, they constructed co-authorship networks, keyword frequency maps, and citation analyses. The study concludes that Arduino is effective for educational and simple applications but limited in complex tasks requiring low-level control or real-time capabilities.

The Mbed OS framework offers multitasking, networking, and security features, but Arm announced the end of official support for the platform starting in 2026 [6], making it less suitable for long-term projects and encouraging the search for alternative solutions.

For graphical interfaces in embedded systems, widely used libraries include LVGL, uGFX, and TouchGFX. LVGL stands out due to its open-source nature, optimization for limited resources, and extensive customization capabilities. Publication [7] provides a detailed overview of using LVGL in conjunction with display controllers, highlighting its advantages in developing efficient interfaces on resource-constrained platforms. An alternative is uGFX, which offers a modular structure and ease of integration across various hardware platforms [8]. TouchGFX, a commercial solution by STMicroelectronics, delivers high-quality GUIs but may be more complex for beginners or small projects.

Recent applied studies demonstrate the use of graphical user interfaces on STM32 microcontrollers primarily as human–machine interfaces in industrial and robotic systems, where GUIs serve monitoring and control purposes rather than being the primary subject of performance optimization [26]. A broader architectural perspective on embedded graphical and multimedia processing is presented in [27], where the organization of rendering pipelines, buffering strategies, external memory usage, and hardware accelerators such as LTDC and DMA2D is shown to play a decisive role in overall system responsiveness. These findings motivate the focus of the present work on architectural-level optimization and predictable integration of graphical subsystems on STM32 platforms.

III. PROPOSED SOLUTION

The STMCMP framework implements a component-oriented architecture, in which each STM32 peripheral interface is represented as a separate, isolated module with clearly defined responsibilities. The central architectural concept is the

combination of modularity and declarative configuration, allowing to quickly configure peripherals using type-safe C++ builders without directly interacting with register-level details. Unlike traditional general-purpose libraries (such as STM32Cube HAL or the Arduino framework), STMCMP deliberately avoids generalization by providing separate low-level implementations for each STM32 microcontroller series (F1, F4, F7, H7). This design makes it possible to capture all hardware-specific features of each series without concealing them behind abstraction layers. However, excessive abstraction typically introduces overhead and may lead to trade-offs in performance and flexibility.

In STMCMP, register-level configuration options are represented using strongly typed abstractions that reflect valid hardware configurations at compile time. This approach enables direct control over peripheral registers while reducing configuration errors and maintaining consistency with vendor reference documentation. This approach reduces the likelihood of errors by enforcing compile-time type checking [9], ensures full consistency with the STM32 reference documentation, and provides direct control over every bit of peripheral configuration. All these aspects constitute essential requirements for time-critical embedded applications.

A. FRAMEWORK ARCHITECTURE

The architecture of the STMCMP framework is designed with a strict separation of responsibilities and a modular, layered organization. The core design principle is to combine low-level CMSIS-based register access with a declarative C++ configuration interface, thereby providing both fine-grained hardware control and architectural clarity [10]. Unlike general-purpose abstraction layers such as STM32Cube HAL or the Arduino framework, STMCMP avoids unnecessary generalization and instead provides dedicated implementations for each STM32 microcontroller family (F1, F4, F7, H7). This ensures that device-specific hardware characteristics are preserved and exposed to the developer without additional overhead.

At the lowest level, the framework directly reflects the hardware registers as defined in the CMSIS [11] headers. This approach provides direct correspondence between configuration code and hardware behavior, while maintaining type safety through compile-time validation. For example, the MODER register for GPIO configuration is mapped to an enum class which enforces valid bitfield values at compile time. This prevents configuration errors, improves maintainability, and ensures consistency with the STM32 reference manuals.

Above this level, the configuration layer exposes a declarative API for peripheral setup. Instead of manually writing register values, developers can use C++ builder patterns to configure system clocks and SDRAM timing while still mapping directly to the underlying hardware. The design philosophy emphasizes clarity without sacrificing control.

The framework further extends this layered design with a set of peripheral modules that encapsulate functionality required for embedded graphical applications, forming a bridge between hardware registers and application-level code.

STMCMP includes a set of peripheral modules that encapsulate functionality required for graphical applications on STM32 platforms. These modules address external memory initialization, display controller configuration, and hardware-accelerated graphics operations, forming the hardware foundation for framebuffer-based rendering and GUI integration. Communication with capacitive touch controllers such as FT6X06 or FT5436 is handled by the I2C module, while GPIO and USART modules provide basic input/output and serial communication capabilities.

A dedicated LVGL port integrates the peripheral modules with the LVGL graphics library, providing adaptive flush_cb callbacks, configurable buffering strategies (single-buffer, double-buffer, or mixed SRAM/SDRAM), and LVGL-compatible input drivers. This design enables efficient GUI rendering while preserving explicit control over the rendering pipeline.

At the system level, STMCMP relies on two foundational layers. CMSIS [11] provides standardized register structures, interrupt definitions, and startup code that directly map to STM32 hardware, ensuring consistency with vendor documentation and ARM Cortex-M conventions. FreeRTOS [12] integration enables optional multitasking support through a lightweight Task wrapper, allowing tasks to be created using modern C++ constructs while maintaining low-level control over scheduling and memory usage.

Together, these layers form a predictable execution environment for real-time graphical applications, ensuring deterministic timing and compatibility with industry-standard embedded software stacks.

To provide a comprehensive overview of the framework structure, a layered architectural view is presented in Fig. 1, illustrating the hierarchical organization of STMCMP.

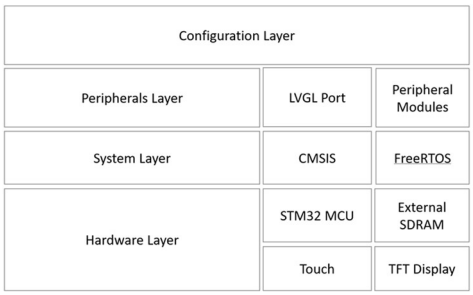

Figure 1. Layered architectural view of the STMCMP framework.

This diagram organizes the framework into four conceptual layers, illustrating the hierarchical separation between hardware, system services, peripheral modules with LVGL integration, and the declarative configuration layer. The layered view highlights how hardware-specific complexity is progressively encapsulated and exposed through clean, type-safe configuration interfaces.

In addition, Fig. 2 presents a data flow diagram of the rendering pipeline, highlighting the sequence of operations from LVGL scene generation through buffer flushing and optional DMA2D acceleration to final output via LTDC. This view complements the architectural diagrams by focusing on runtime behavior rather than structural relationships.

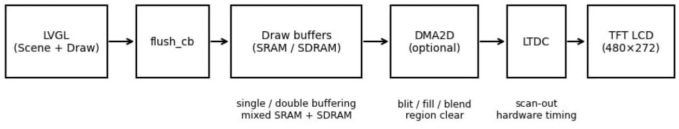

Figure 2. Class diagram of the STMCMP framework

In addition to the layered view, a class diagram is presented (Fig. 3) to capture the internal structure of the framework at the level of software entities. It depicts the relationships between builder classes, peripheral modules, and integration components that connect STMCMP to the LVGL library, as well as the lightweight FreeRTOS Task wrapper [14].

Together, these diagrams provide complementary macroscopic and microscopic perspectives on the STMCMP architecture, illustrating both its high-level organization and the concrete realization of its modular design.

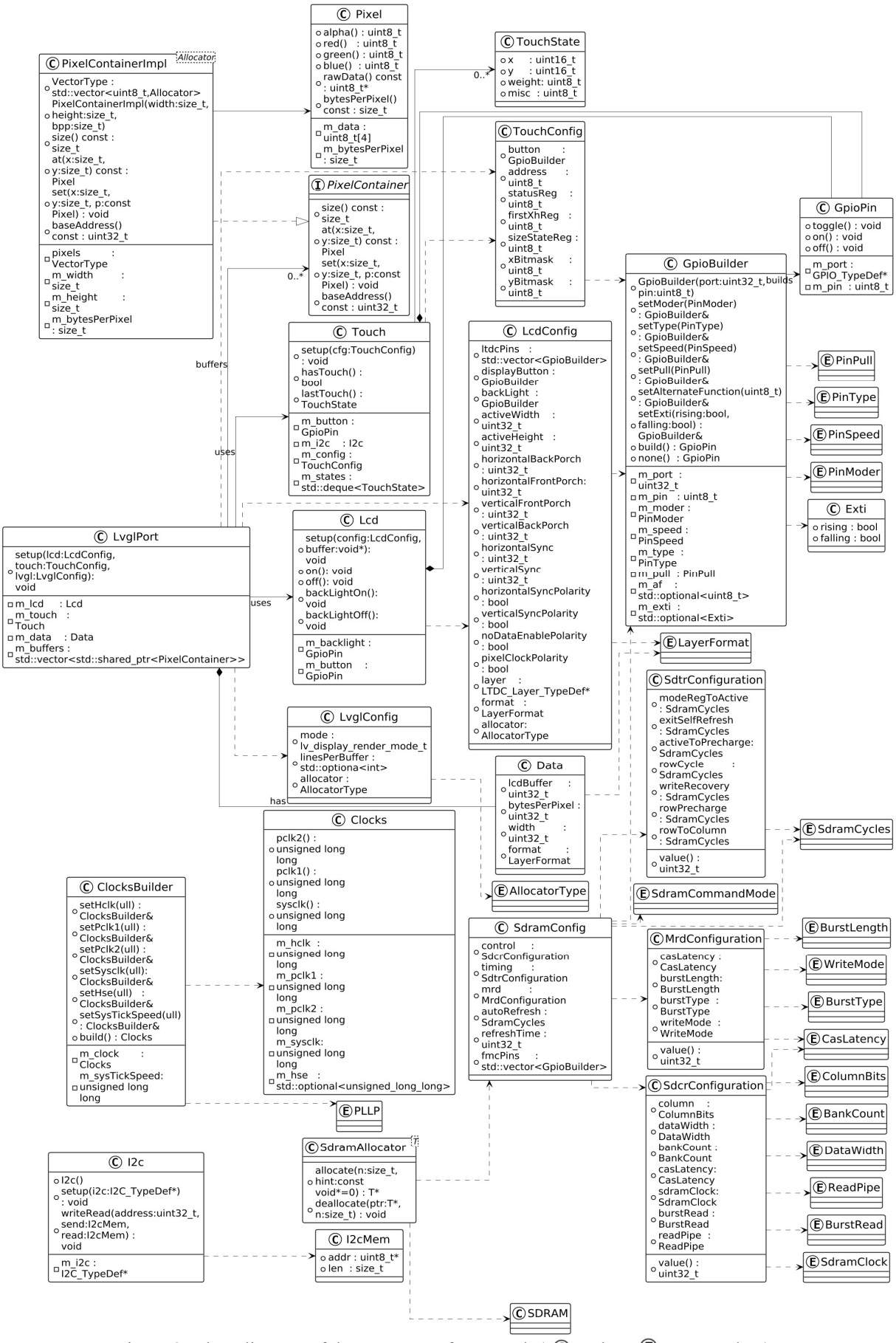

Figure 3. Class diagram of the STMCMP framework (Ⓒ – class, Ⓔ – enum class).

B. SUPPORTED PERIPHERALS AND PREDEFINED CONFIGURATIONS

The STMCMP framework for STM32 F7 series provides comprehensive implementation for a wide range of peripheral interfaces, with a focus on components that critically affect the performance of graphical applications. All modules are developed as independent units that interact directly with CMSIS-level registers and have no external dependencies on HAL or other abstraction libraries. The following key peripherals are supported:

- GPIO providing basic input/output control,
- FMC (Flexible Memory Controller) providing external SDRAM connection [15],
- LTDC (LCD-TFT Display Controller) accessing layer configuration, buffering, resolution, and color formats [16],
- DMA2D providing hardware acceleration for buffer copying, ARGB blending, and region clearing [17],
- I2C [18] presenting communication with touch controllers such as FT6X06 and FT5436,
- System Clock providing precise PLL/HSI/HSE configuration [19] tailored for graphical rendering frequency targets.

In addition to peripheral support, the framework includes predefined configurations for commonly used components, which can be used as is or serve as templates for further adaptation. Among these components:

- Memory MT48LC4M32B2B5-6A – a 64-Mbit (4Mx32) SDRAM chip with optimized timing and initialization for the STM32F746. The configuration for this chip includes for delays, bank count, CAS latency, SDRAM frequency (TwoHCLK), auto-refresh mode, and refresh timing.
- Touch controllers FT6X06 / FT5436 – I2C-compatible multitouch screens. A driver, reading touch coordinates and dispatches events to LVGL, is provided by STMCMP.
- Display controller support – the display resolution of 480×272 and ARGB8888 (or RGB565) format can be configured using the LtdcBuilder class of STMCMP, which automatically selects appropriate PLL frequencies and timing parameters. Both double buffering and DMA2D-assisted modes are supported.

An example of SDRAM configuration on the STM32F746-DISCO board is shown in Fig. 4.

C. INTEGRATION WITH LVGL AND FREERTOS

The STMCMP framework provides full support for both the LVGL graphics library and the FreeRTOS operating system, enabling the development of robust graphical applications with multitasking capabilities [20], clear separation of responsibilities, and high runtime stability.

The integration with LVGL is designed to support multiple rendering and buffering strategies, allowing the evaluation of different data flow organizations between the graphics library and the underlying hardware. FreeRTOS is used to enable concurrent execution of graphical rendering and application logic, facilitating the analysis of GUI behavior under multitasking conditions [21].

The framework also supports LVGL-compatible input device drivers for touchscreens, including coordinate transmission, multitouch handling, and gesture processing.

```
inline SdramConfig MT48LC4M32B2B56A()
{
    return SdramConfig {
        .control = {
            .column = ColumnBits::Bits8,
            .dataWidth = DataWidth::Bits16,
            .bankCount = BankCount::Four,
            .casLatency = CasLatency::Latency2,
            .sdramClock = SdramClock::TwoHCLK,
            .burstRead = BurstRead::Enabled,
            .readPipe = ReadPipe::NoDelay,
        },
        .timing = {
            .modeRegToActive = SdramCycles::Cycles2,
            .exitSelfRefresh = SdramCycles::Cycles7,
            .activeToPrecharge = SdramCycles::Cycles4,
            .rowCycle = SdramCycles::Cycles7,
            .writeRecovery = SdramCycles::Cycles2,
            .rowPrecharge = SdramCycles::Cycles2,
            .rowToColumn = SdramCycles::Cycles2,
        },
        .mrd = {
            .casLatency = CasLatency::Latency2,
            .burstLength = BurstLength::One,
            .burstType = BurstType::Sequential,
            .writeMode = WriteMode::SingleLocationAccess,
        },
        .autoRefresh = SdramCycles::Cycles8,
        .refreshTime = 781,
        .fmcPins = fmcPins(),
    };
}
```

Figure 4. An example of SDRAM memory configuration.

FreeRTOS integration is achieved via a lightweight Task wrapper, which abstracts task creation using `std::function<void()>`, allowing modern C++ constructs to be seamlessly embedded in traditional RTOS environments without performance loss. An example of FreeRTOS task creation is shown in Fig. 5.

```
makeTask([] {
    while (true) {
        // update screen or poll input
    }
}, "RenderTask", TaskStackSize::Default, TaskPriority::Normal);
```

Figure 5. An example of making FreeRTOS task.

Stack size (`TaskStackSize`) and priority (`TaskPriority`) are specified via type-safe enum class values, eliminating configuration errors. At the same time, it provides a higher level of abstraction while preserving control over task objects by means of `std::shared_ptr<Task>`.

Owing to this level of integration, STMCMP enables the construction of architecturally clean applications in which the GUI, event handling, data processing, and system-level tasks can run concurrently, without interference, fully leveraging the advantages of FreeRTOS in real-time [22] environments.

IV. COMPARATIVE ANALYSIS WITH THE REFERENCE LVGL PORT FOR STM32F746-DISCO

This section presents an analysis of two LVGL-based graphical interface implementations: the reference port for the STM32F746G-DISCO board and a custom implementation created using the STMCMP framework. The purpose of this

comparison is not only to evaluate performance in terms of FPS and frame rendering time, but also to identify architectural advantages provided by declarative peripheral configuration, flexible buffering strategies, and precise memory management.

The analysis further includes a comparison of resource utilization such as firmware size and RAM consumption, and examines specific scenarios where the architectural flexibility of STMCMP offers clear benefits over the fixed design of the reference port.

A. TESTING METHODOLOGY

The testing was conducted using the STM32F746G-DISCO development board, which features an STM32F746NG microcontroller running at 216 MHz, an integrated TFT display with a resolution of 480×272 pixels, and a FT5436 capacitive touch controller. Both implementations – the reference port and the one based on STMCMP – used the same version of the LVGL library and an identical `lv_conf.h` configuration file. This ensured that all test results were unaffected by differences in LVGL settings.

To eliminate the influence of multitasking, FreeRTOS was excluded from both implementations. This decision was based on the fact that the reference LVGL port for the STM32F746-DISCO board, available in the `lv_port_stm32f746_disco` repository, does not include an RTOS implementation. As a result, both systems were tested under comparable conditions, with all graphical updates executed in the main loop without interruptions from external tasks.

The system clock configuration remained unchanged for all test cases: the microcontroller operated at its full frequency of 216 MHz, with properly configured PLL and peripheral buses. To evaluate the architectural impact of the rendering pipeline, several buffering strategies were tested:

- Rendering into internal SRAM with subsequent copying to SDRAM;
- Direct rendering into SDRAM using a single buffer;
- Double-buffering in SDRAM with buffer switching via LTDC.

These modes were tested only within the STMCMP implementation, as the reference port supports only a single hardcoded buffering strategy designed for general-purpose performance, with no option to switch or reconfigure the rendering pipeline.

Performance evaluation was carried out using the standard LVGL benchmark demo [23], which sequentially displays graphical scenes of varying complexity – from simple primitives to animated widgets and full-screen ARGB images. LVGL's built-in performance monitoring tools were used to record the average frames per second (FPS) and average frame render time for each test. All values were averaged after interface stabilization to ensure accurate and consistent measurements.

Both implementations are available as open-source projects: the custom STMCMP-based implementation [24] and the reference LVGL port for STM32F746G-DISCO [25].

B. PERFORMANCE COMPARISON: FPS AND FRAME RENDER TIME

The comparison results of average frames per second (FPS) and average frame render time between the STMCMP-based implementation and the reference LVGL port are presented in Table 1. The benchmark includes a range of graphical scenes, from simple primitives to complex compositing scenarios, as defined in the LVGL demo suite.

The results of the benchmark demo execution demonstrate a significant performance improvement of the STMCMP-based implementation across most test scenes.

Table 1. Comparison STMCMP and reference LVGL port

Scene	Ref LVGL port		STMCMP	
	FPS	render time, ms	FPS	Render time, ms
Empty screen	43	10	62	5
Animated wallpaper	35	21	61	8
Single rectangle	51	1	61	0
Multiple rectangles	52	10	61	3
Multiple RGB images	53	4	61	0
Multiple ARGB images	49	10	61	3
Rotated ARGB images	9	94	24	39
Multiple labels	51	6	61	2
Full-screen text	8	111	17	55
Multiple arcs	52	3	61	1
Containers	51	12	61	4
Containers with overlay	21	38	59	16
Containers with opacity	45	19	61	5
Containers with opa_layer	31	30	61	10
Containers with scrolling	20	41	50	18
Widgets demo	9	53	28	24
Average across all scenes	36	28	53	12

In simple and moderately loaded scenarios such as rendering one or more rectangles, displaying labels, arcs, or RGB images, the framework reaches FPS values close to the limits imposed by the display refresh configuration.

In these cases, the frame render time approaches zero or remains within a few milliseconds, indicating an efficient graphics pipeline.

In more demanding scenes, involving rotated ARGB images, full-screen text rendering, or complex containers with overlays and transparency, the performance of the reference port degrades significantly. The FPS in such cases often drops to single-digit values, while frame rendering times exceed 50-100 ms. STMCMP implementation shows a more gradual performance degradation under load: although FPS also decreases under load, the drop is much more gradual, and the frame processing time remains several times lower, maintaining acceptable interactivity under intensive conditions.

This performance behavior is illustrated in Fig. 6, which compares the achieved FPS and frame rendering time of the STMCMP-based implementation and the reference port across all benchmark scenes.

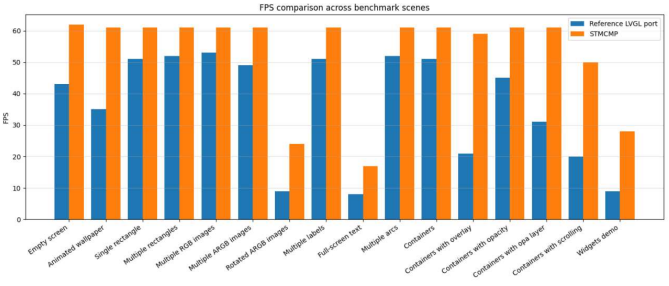

Figure 6. Comparison STMCMP and reference LVGL port

On average across all scenes, the STMCMP-based implementation achieves an FPS increase of approximately 47% compared to the reference port (53 FPS vs. 36 FPS), while

the average frame render time is reduced by more than half (from 28 ms to 12 ms). This performance dynamic is especially important for resource-constrained systems, where every millisecond is critical for maintaining interactivity. The results confirm that STMCMP not only delivers higher performance but also demonstrates more consistent timing characteristics under load, which is crucial for real-time latency-sensitive user interfaces.

C. RESOURCE USAGE ANALYSIS: RAM AND FLASH

A comparison of the compiled ELF files from both implementations enables an assessment of memory usage efficiency, particularly regarding Flash memory (for code and constants) and RAM (for variables, stack, and buffers). For the sake of objective comparison, only the memory sections that directly contribute to actual firmware size and runtime memory consumption were considered: `.text`, `.rodata`, `.data`, `.bss`, and the user stack (`_user_heap_stack`).

In the STMCMP-based implementation, the combined size of the `.text` and `.rodata` sections, which determine Flash usage, is approximately 515 KB (225,612 + 290,212 bytes). In contrast, the reference port consumes about 651 KB (380,392 + 271,208 bytes), representing a reduction of nearly 21% in flash size. This reduction is associated with the absence of the HAL layer and a framework structure focused on direct peripheral configuration.

In terms of RAM usage, the combined size of the .data, .bss, and user stack sections in the STMCMP implementation is about 135 KB (1,480 + 131,960 + 1,536 bytes), whereas the reference port consumes over 271 KB (1,692 + 263,836 + 6,144 bytes). This reduction in RAM usage (over 50%) can be leveraged either to lower overall power consumption or to allocate more resources to other components such as larger LVGL buffers or FreeRTOS task stacks.

Thus, the resource analysis confirms that the STMCMP-based implementation offers substantial savings in both Flash and RAM memory, making the framework particularly well-suited for building graphical applications on resource-constrained embedded systems without compromising performance.

Thus, the resource analysis confirms that the STMCMP-based implementation offers substantial savings in both Flash and RAM memory, making the framework particularly well-suited for building graphical applications on resource-constrained embedded systems without compromising performance.

The proposed framework is subject to several practical limitations. Its performance characteristics depend on the availability of hardware accelerators such as LTDC and DMA2D, as well as on external memory resources used for framebuffer storage. Display resolution, pixel format, and memory bandwidth directly affect achievable rendering performance. Consequently, the results presented in this work are specific to the evaluated hardware configuration.

The experimental evaluation in this work is limited to the STM32F746 microcontroller. While the architectural approach of STMCMP can be adapted to other STM32 families, such as STM32F4 or STM32H7, this adaptation requires dedicated register mappings and peripheral-specific implementations for each target device. Therefore, scalability in this study refers to architectural applicability rather than demonstrated performance portability across multiple STM32 series.

V. CONCLUSION

This work introduces and evaluates the STMCMP framework, which simplifies and enforces type-safe configuration of STM32 peripherals, with a specific focus on graphical applications built using the LVGL library. The proposed solution provides a modular architecture with a focus on type-safe peripheral configuration, while achieving significant advantages in both performance and resource efficiency.

The experimental comparison with the reference LVGL port for STM32F746G-DISCO demonstrates that the STMCMP-based implementation provides more consistent rendering performance across evaluated scenarios, a higher frame rate (FPS increase an average up to 47%), and a twofold decrease in frame render time. In addition, the firmware size is reduced by over 20%, and RAM usage is cut by more than half, which is critical for resource-constrained embedded systems.

A key advantage of STMCMP framework lies in its intentional avoidance of universality, a trait commonly associated with HAL and similar frameworks. Instead, STMCMP offers a highly specialized configuration model, tightly integrated with the hardware characteristics of each STM32 series. This approach results not only in high performance but also in deterministic system behavior.

An additional benefit is the support of multiple rendering strategies: rendering to internal SRAM followed by copying to SDRAM, direct rendering to SDRAM, and double buffering with LTDC buffer switching. This level of flexibility allows developers to adapt the rendering pipeline to specific requirements, for example, reducing RAM usage in widget-intensive scenarios.

The research results confirm the viability of domain-specific frameworks for embedded systems, particularly those targeting performance and low-level control. The developed framework STMCMP demonstrates clear potential for future extension including support for additional microcontroller families (e.g., STM32H7), display configurations, and sensor technologies.

References

[1] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*, 2nd ed. Springer, New York, 2011, 396 p.

[2] S. Haug, C. Böhm, and D. Mayer, "Automated code generation and validation for software components of microcontrollers," *Proceedings of the 20th ACM International Conference on Computing Frontiers (CF'23)*, Bologna, Italy, May 9–11, 2023, pp. 202–208.

[3] J. Beningo, *Embedded basics – API's vs HAL's*, Beningo Embedded Group, Apr. 2016, [Online]. Available at: https://www.beningo.com/embedded-basics-apis-vs-hals/

[4] Y. Zhu, *Embedded Systems with Arm Cortex-M Microcontrollers in Assembly Language and C*, 2nd ed., E-Man Press LLC, 2015.

[5] N. K. Prabowo and I. Irwanto, "The implementation of Arduino microcontroller boards in science: A bibliometric analysis from 2008 to 2022," *Journal of Engineering Education Transformations*, vol. 37, no. 2, pp. 107–120, 2023. https://doi.org/10.16920/jeet/2023/v37i2/23154.

[6] Arm Ltd., "Important update on Mbed end-of-life," *Mbed Community Forum*, July 2024, [Online]. Available at: https://forums.mbed.com/t/important-update-on-mbed-end-of-life/23644

[7] I. V. Filippenko, V. R. Korniyenko, and H. K. Kulak, "Overview of graphics libraries for embedded platforms," *Radioelectric and Informatic*, no. 1, pp. 47–53, 2020. (in Ukrainian)

[8] G. İşnas and N. Şenyer, "Comparison of TouchGFX and LVGL embedded hardware performance for graphical user interfaces," *Gazi University Journal of Science Part C: Design and Technology*, vol. 9, no. 2, pp. 215–224, 2021. https://doi.org/10.29109/gujsc.915163.

[9] D. Vandevoorde, N. M. Josuttis, and D. Gregor, *C++ Templates: The Complete Guide*, 2nd ed. Addison-Wesley, Boston, 2017, 822 p.

[10] B. P. Douglass, *Real-Time UML Workshop for Embedded Systems*. Elsevier, Amsterdam, 2014, 378 p.

[11] Arm Ltd, *Common Microcontroller Software Interface Standard (CMSIS)*, 2022, [Online]. Available at: https://arm-software.github.io/CMSIS_5/

[12] R. Barry, *Mastering the FreeRTOS Real Time Kernel*, 2nd ed. Real Time Engineers Ltd., 2020, 399 p.

[13] J. Aynsley, *Modern C++ in embedded systems – myth and reality*, Embedded.com, 2015, [Online]. Available at: https://www.embedded.com/modern-c-in-embedded-systems-myth-and-reality/

[14] J. J. Labrosse, *µC/OS-III: The Real-Time Kernel for the STM32*, Micrium Press, Weston, 2010, 888 p.

[15] STMicroelectronics, *AN2784: Using the high-density STM32F10xxx FSMC peripheral to drive external memories*, Application Note, 2017, [Online]. Available at: https://www.st.com/resource/en/application_note/an2784-using-the-highdensity-stm32f10xxx-fsmc-peripheral-to-drive-external-memories-stmicroelectronics.pdf

[16] STMicroelectronics, *AN4861: LCD-TFT Display Controller (LTDC) on STM32 MCUs*, Application Note, 2017, [Online]. Available at: https://www.st.com/resource/en/application_note/an4861-introduction-to-lcdtft-display-controller-ltdc-on-stm32-mcus-stmicroelectronics.pdf

[17] STMicroelectronics, *AN4943: How to use Chrom-ART Accelerator (DMA2D) to refresh an LCD-TFT display on STM32 MCUs*, Application Note, 2017. [Online]. Available at: https://www.st.com/resource/en/application_note/an4943-how-to-use-chromart-accelerator-to-refresh-an-lcdtft-display-on-stm32-mcus-stmicroelectronics.pdf

[18] STMicroelectronics, *AN4235: I²C timing configuration tool for STM32F0/F3 microcontrollers*, Application Note, 2013. [Online]. Available at: https://www.st.com/resource/en/application_note/an4235-i2c-timing-configuration-tool-for-stm32f3xxxx-and-stm32f0xxxx-microcontrollers-stmicroelectronics.pdf

[19] STMicroelectronics, *UM1718: STM32CubeMX for STM32 configuration and initialization C code generation*, User Manual, 2014, [Online]. Available at: https://www.st.com/resource/en/user_manual/um1718-stm32cubemx-for-stm32-configuration-and-initialization-c-code-generation-stmicroelectronics.pdf

[20] A. Marongiu and L. Benini, "An OpenMP compiler for efficient use of distributed scratchpad memory in MPSoCs," *IEEE Transactions on Computers*, vol. 61, no. 2, pp. 222–236, 2012. https://doi.org/10.1109/TC.2010.199.

[21] STMicroelectronics, *AN4943* How to use Chrom-ART Accelerator to refresh an LCD-TFT display on STM32 MCUs Application Note, 2024, [Online]. Available at: https://www.st.com/resource/en/application_note/an4943-how-to-use-chromart-accelerator-to-refresh-an-lcdtft-display-on-stm32-mcus-stmicroelectronics.pdf

[22] J. Arm et al., "Measuring the Performance of FreeRTOS on ESP32 Multi-core," *IFAC-PapersOnLine*, vol. 55, issue 4, pp. 292-297, 2022. https://doi.org/10.1016/j.ifacol.2022.06.048.

[23] LVGL Developers, *lv_demo_benchmark: Performance benchmarking demo for LVGL*, 2024, [Online]. Available at: https://github.com/lvgl/lvgl/tree/master/demos/benchmark

[24] O. Stelmakh, *STMCMP: A modular CMSIS-based framework for STM32*, [Online]. Available at: https://github.com/StelmakhAleksandr/stmcmp

[25] LVGL Developers, *STM32F746G-DISCO LVGL demo project*, [Online]. Available at: https://github.com/lvgl/lv_port_stm32f746_disco

[26] K. B. A. Borowski and K. Wojtulewicz, "Implementation of robotic kinematics algorithm for industrial robot model using microcontrollers," IFAC-PapersOnLine, vol. 55, no. 10, pp. 108–113, 2022. https://doi.org/10.1016/j.ifacol.2022.09.063.

[27] Y. Krainyk, "Embedded systems multimedia framework for microcontroller devices," *Advances in Cyber-Physical Systems*, vol. 8, no. 1, pp. 43–49, 2023. https://doi.org/10.23939/acps2023.01.043.

[28] A. Author et al., "Relevant HAL Interface Requirements for Embedded Systems," arXiv preprint arXiv:2512.14514, 2025.

[29] H. Yoon, J. Kim, and S. Ha, "Performance optimization techniques for ARM Cortex-M processors in embedded systems," *IEEE Access*, vol. 8, pp. 207789–207801, 2020. DOI: 10.1109/ACCESS.2020.3038211.

***OLEKSANDR STELMAKH** PhD, Senior lecturer in the Department of Computer Science and Software Engineering, Igor Sikorsky Kyiv Polytechnic Institute. Research interests include neural networks, embedded systems, and performance optimization.*

***INNA V. STETSENKO** Doctor of Science, Professor in the Department of Computer Science and Software Engineering, Igor Sikorsky Kyiv Polytechnic Institute. Research interests include parallel computing, artificial intelligence, simulation, and Petri nets.*

***ANTON DYFUCHYN** PhD, Senior lecturer in the Department of Computer Science and Software Engineering, Igor Sikorsky Kyiv Polytechnic Institute. Research interests include visual programming languages, parallel computing, and systems simulation.*

***ALEXANDER ZARICHKOVYI** PhD, Assistant if the Department of Computer Science and Software Engineering, Kaggle competition master, Igor Sikorsky Kyiv Polytechnic Institute. Specializing on research in computer vision field, video recognition, embedded systems, and neural network optimization.*

***OLEKSANDRA DYFUCHYNA** PhD, Senior lecturer in the Department of Computer Science and Software Engineering, Igor Sikorsky Kyiv Polytechnic Institute. Research interests include multithreaded programming, systems simulation, and performance optimization.*