

A Comparative Analysis of Embedded Control Approaches in Robotics: Bare-Metal MCU, RTOS, and ROS

Yizhuo Yan

*School of Energy Systems, LUT University, Lappeeranta, Finland
Ryan2220208@163.com*

Abstract. By evaluating three dominant embedded control architectures in modern robotics: bare-metal microcontroller programming, the Robot Operating System (ROS), and Real-Time Operating Systems (RTOS), this study provides an assessment of their characteristics and performance. As robotic systems evolve to handle increasingly complex autonomous tasks, selecting a control architecture that balances real-time determinism with high-level intelligence has become a critical engineering challenge. Through a literature review and comparative analysis, this study analyzes these platforms from five main perspectives: system architecture, real-time performance, development complexity, hardware resource demands, and engineering costs. The findings show that bare-metal microcontroller systems are optimal for simple, low-power control; RTOS-based systems are indispensable for safety-critical, high-precision operations; and ROS-based frameworks serve as the premier solution for modular system integration. The study concludes that modern robotics is shifting toward a tiered integration model that leverages the strengths of all three architectures to achieve both intelligence and reliability.

Keywords: Embedded systems, Robotics, Microcontroller, Robot Operating System (ROS), Real-Time Operating System (RTOS)

1. Introduction

The rapid proliferation of robotics in industrial automation, smart manufacturing, and service sectors has necessitated a parallel evolution in embedded control architectures. Traditionally, robotic control was restricted to simple microprocessors executing basic sequences. However, modern robots now require the simultaneous processing of high-bandwidth data from LiDAR, 3D cameras, and Inertial Measurement Units (IMUs) while also executing complex path-planning and perception algorithms [1-3]. Although various platforms are available, there remains a research gap in providing a structured comparison framework to support the selection of appropriate architectures based on specific task requirements. Against this background, this paper examines the trade-off between the high-level modularity of ROS-based systems and the low-level deterministic performance of RTOS-based and bare-metal microcontroller systems. Building on a review of relevant literature and comparative analysis, this study explores the technical principles and application boundaries of each architecture. In doing so, it provides a reference for robotic hardware-software co-development and helps guide architecture selection for improved stability and lower cost. The following sections will

detail the fundamental requirements of robotic control before performing a deep dive into each architectural category.

2. Foundations of embedded control for robotics

2.1. Evolution of robotics and embedded systems

The history of robotic control mirrors the continuous progress of embedded computing, driven by ever-increasing demands for mobility and complex environmental interaction. Initially, early robots were restricted to simple, hard-wired logic or basic 8-bit microprocessors capable only of handling repetitive, single-purpose pick-and-place tasks. As applications grew more complex, these centralized, single-processor designs struggled to manage the heavy influx of sensor data while simultaneously computing high-frequency motor commands. This computational bottleneck spurred an evolutionary leap toward distributed embedded architectures [3, 4]. In modern setups, specialized microcontrollers using CAN bus, EtherCAT, or deterministic Ethernet handle individual joints or dedicated sensor arrays. This distribution of intelligence spreads the processing workload, enables parallel computation, and acts as a key technical milestone enabling multi-axis, sensor-rich robotic systems.

2.2. Impact of control architecture on robot performance

The selected control architecture effectively functions as the robot's nervous system, directly dictating the efficiency with which the central processing unit (CPU) processes time-critical sensor data and issues precise motor commands. An inefficient or poorly designed architecture inevitably introduces non-deterministic latency and jitter (variable delay) [4, 5]. This jitter has a catastrophic effect on system stability, particularly within high-speed, high-frequency closed-loop control systems. When feedback signals or control outputs miss their tightly scheduled time windows, the continuity of the control loop breaks down, degrading tracking accuracy. In physical terms, this latency disrupts feedback loops, resulting in severe mechanical vibration, reduced positioning precision, and, in extreme cases, complete system instability leading to mechanical failure or collisions.

2.3. Key design requirements of embedded controllers

The design of effective embedded controllers involves four fundamental, often conflicting technical requirements, which directly stem from the architectural challenges discussed in previous sections. First, real-time response is not a preference but a safety requirement [5, 6]. The system must ensure that critical tasks, like triggering an emergency stop upon obstacle detection, execute within an extremely tight and fixed time window. And this determinism supports high-speed feedback control loops and mitigates the effects of architectural jitter. Second, reliability maintains stable performance and accurate decisions under harsh conditions. In particular, in settings with strong electromagnetic interference or persistent mechanical vibration, the controller must maintain stable operation. To achieve this, hardware-software isolation is essential to prevent environmental noise from corrupting control signals and causing system instability. Third, scalability is required to easily integrate new hardware, such as advanced vision modules, additional sensors, or extra actuators, as the robotic platform evolves. This capability drives the shift to modular, distributed system architectures. Finally, cost constraints are critical for commercial viability. The bill of materials

(BOM) and development time must be carefully balanced against performance metrics. Meanwhile, efficient architectural design enables the use of cost-effective processors while still maintaining real-time deterministic performance, thereby enhancing market competitiveness.

3. The robotic control architectures based on bare-metal MCU, ROS, and RTOS

3.1. Bare-metal microcontroller systems

Through the combination of a processor core, memory, and programmable input/output peripherals, microcontrollers (MCUs) enable direct register-level hardware control [3, 7]. From an implementation perspective, bare-metal MCU development ranges from beginner-friendly frameworks like Arduino IDE to professional setups using C++ and Hardware Abstraction Layers (HAL) for ARM Cortex-M or AVR chips, supporting PWM for motor control and I2C/SPI for sensor communication. As a result, bare-metal MCU systems offer deterministic execution, running instructions predictably without an operating system scheduler. In addition, they are low-power and cost-effective, making them well-suited for time-sensitive and mass-produced robotic applications.

However, the traditional "super-loop" architecture, a single continuous `while(1)` loop, limits multitasking and long-term maintainability [7]. Thus, as robotic code complexity increases, handling asynchronous events, such as processing encoder pulses while reading UART data, without a real-time scheduler leads to disorganized code. Developers must rely on nested interrupt service routines (ISRs) or manual state machines. This creates tight code coupling, where adding a new sensor or minor feature can inadvertently alter the execution timing of the entire loop, vastly increasing bug rates and making debugging an engineering nightmare. Besides, MCUs generally lack the memory and processing power required to run high-level libraries for vision, autonomous planning, or complex perception tasks.

Despite these limitations, bare-metal MCU systems are well-suited for specialized low-level control, especially in brushed and brushless DC (BLDC) motor control applications. For example, in industrial robotic arms or wheeled robots, an ARM Cortex-M4 MCU executes a high-frequency Field-Oriented Control (FOC) loop, reading feedback from quadrature encoders or Hall-effect sensors, computing a PID algorithm, and updating PWM registers at a deterministic 20 kHz rate. This enables precise current and velocity control, while higher-level positioning commands are handled by supervisory systems. In addition, MCUs are also used in educational robots, toy robots, and dedicated sensor-fusion coprocessors.

3.2. ROS-based robotic software architecture

The ROS is a meta-operating system that provides hardware abstraction, device control, and inter-process communication for robotic system development [1, 2]. It is built on a node-based architecture in which processes exchange data via a publish/subscribe model over topics. Key components include TF, which manages coordinate frames over time for spatial consistency between sensors and actuators, and URDF, an XML-based format for robot modeling used in simulation and visualization [1, 2, 8]. ROS supports modular system design and integration through a large open-source ecosystem covering SLAM, navigation, and perception. And its architecture also enables hardware flexibility, allowing components such as LiDAR sensors to be replaced without redesigning the entire system.

However, standard ROS configurations, particularly ROS 1 and default ROS 2 deployments on general-purpose Linux systems such as Ubuntu without real-time patches, exhibit non-deterministic

latency. This is due to operating system scheduling, thread switching, and network stack overhead, which introduce latency and jitter at the millisecond scale, typically 10-50 ms or higher under heavy load. While ROS 2 enables determinism via PREEMPT_RT and DDS QoS, system-level setup is required. In contrast, bare-metal MCU systems support microsecond-level control loops. This timing uncertainty makes unmodified ROS deployments unsuitable for high-frequency feedback control such as joint torque control or active balancing, where delays can lead to system instability or oscillation.

To address latency constraints in complex robotic systems, a common architecture separates high-level coordination from low-level control, using ROS for supervisory functions. In systems such as autonomous warehouse forklifts or humanoid robots, a high-level industrial PC running ROS handles data-intensive tasks, including processing 3D LiDAR point clouds and camera frames for SLAM and global path planning. Debugging and validation are supported by tools such as rviz for real-time visualization of coordinate frames (via TF) and rqt_plot for trajectory analysis. The computed path is then translated into low-frequency velocity commands (geometry_msgs/Twist) and transmitted to a microcontroller layer via a serial link. This division allows ROS to manage computationally intensive tasks while delegating real-time execution to bare-metal controllers.

3.3. RTOS-based embedded robotic systems

The RTOS is designed for deterministic task execution, prioritizing timing predictability over general throughput [5, 6, 9]. Unlike general-purpose systems such as Linux or Windows, it guarantees bounded worst-case execution times (WCET) for critical tasks, ensuring precise timing even under heavy load. Thus, RTOS kernels use preemptive, priority-based scheduling, immediately interrupting lower-priority tasks when higher-priority events occur, such as emergency stops or sensor deadlines. Interrupt response occurs at the microsecond level, while task context switches typically range from tens to hundreds of microseconds depending on processor architecture, enabling deterministic and timely responsiveness. By integrating the low-overhead efficiency of microcontroller systems with structured multitasking, RTOS enables concurrent operations, including sensor polling, trajectory generation, and safety monitoring, without introducing the latency typical of general-purpose operating systems. This deterministic execution supports both reliability and safety in real-time robotic systems.

However, the deployment of an RTOS introduces specific architectural trade-offs and engineering limitations. First, while task context switches are fast, scheduling latency can still arise during nested hardware interrupts, delaying critical software tasks. Moreover, priority-based scheduling is highly vulnerable to priority inversion. This occurs where a medium-priority task preempts a low-priority task holding a shared resource required by a high-priority task, potentially blocking it indefinitely; mitigation typically relies on Priority Inheritance Protocols (PIP). Additionally, an RTOS imposes memory overhead compared to bare-metal systems, as each task requires allocated stack space and a Task Control Block (TCB), raising RAM demands and challenging memory management in resource-limited embedded systems.

Despite these challenges, RTOS-based embedded systems are used in applications requiring strict timing guarantees, including drone flight control, industrial robotics, and medical surgical systems. For example, in quadcopters, the RTOS schedules IMU reading, Kalman filtering, and PWM output tasks, thus maintaining a fixed 1-4 kHz attitude control loop to stabilize flight. Similarly, in industrial robotic arms, it synchronizes joint control computations with EtherCAT fieldbus cycles under 1 ms, ensuring consistent trajectory execution. In surgical robots, it handles real-time haptic

feedback and vision tracking, and triggers immediate actuator shutdown within microsecond-level response when abnormal conditions are detected.

4. Layered approaches to performance and integration in robotic embedded systems

4.1. Performance comparison

Bare-metal MCU programming, RTOS, and ROS differ in architecture, real-time behavior, resource requirements, and typical applications [1, 3, 4, 6]. Bare-metal MCU systems provide deterministic execution with extremely low latency ($<10\ \mu\text{s}$), suitable for high-frequency, low-level tasks such as PWM generation and direct sensor reads. RTOS adds structured preemptive multitasking, enforcing deterministic scheduling with worst-case execution times of $10\text{--}100\ \mu\text{s}$. This allows concurrent task execution while introducing debugging complexities like race conditions and priority inversion. ROS operates on general-purpose OSs in a distributed, node-based framework. It prioritizes modularity and throughput for complex tasks such as SLAM, vision, and trajectory planning, but exhibits millisecond-level latency and is sensitive to CPU load and network congestion.

The systems also vary widely in resource demands, as shown in Table 1 [1, 3-6]. MCUs require minimal RAM (2–512 KB) and processing power, making them cost-effective for simple controllers and educational or consumer robots. RTOS requires more memory (64 KB–8 MB) to support task stacks, control blocks, and inter-process communication, fitting mid-range embedded processors. ROS demands multi-core CPUs, gigabytes of RAM, and full file systems to run its middleware, resulting in higher hardware costs and energy consumption.

Table 1. Comparison of MCU, RTOS and ROS

Criteria	Bare-Metal MCU	RTOS	ROS
Architecture	Super-loop (bare-metal)	Preemptive multitasking kernel	Distributed node-based framework
Real-time Capability	Deterministic execution	Hard real-time	Soft real-time
Typical Latency	$< 10\ \mu\text{s}$	$10\text{--}100\ \mu\text{s}$	$1\text{--}10\ \text{ms}$
RAM Requirement	2–512 KB	64 KB–8 MB	$> 1\ \text{GB}$
CPU Requirement	Low-end MCU	MCU/Embedded CPU	Multi-core CPU
Development Complexity	Low	Medium	High
Cost	Very low	Moderate	High
Scalability	Poor	Moderate	Excellent
Typical Applications	Sensors, simple controllers	Industrial automation, motor control	Autonomous robots, AI systems

Accordingly, modern robotic systems typically combine these paradigms, where bare-metal MCUs or RTOS-based controllers execute microsecond-level, safety-critical tasks at the hardware layer, and ROS on high-performance computers oversees higher-level planning, perception, and system coordination. And this architecture leverages the strengths of each platform while mitigating their respective limitations.

4.2. Integration trends

Modern robotics increasingly emphasizes layered, heterogeneous integration, integrating low-level deterministic control with high-level cognitive frameworks [2, 10, 11]. In this context, engineers leverage a tiered design, where high-level computing nodes handle complex tasks including SLAM, costmap generation, and path planning, while low-level MCUs running RTOS execute microsecond-critical control tasks such as encoder counting, PID velocity tracking, and battery monitoring. Thus, this approach improves safety and flexibility but also brings architectural and engineering challenges, as shown in Table 2.

Table 2. Layered integration in modern robotic systems

Layer	Function	Key Tasks	Typical Hardware/Software
High-Level Node	Planning and perception	SLAM, path planning, costmap generation, fleet coordination	Multi-core PC, Linux, ROS
Low-Level MCU/RTOS	Microsecond-critical control	Encoder reading, PID loops, BMS monitoring	STM32, NXP i.MX RT, FreeRTOS
Edge AI / Peripherals	Local low-latency processing	Wake-word detection, gesture recognition	NPU on MCU, vision chips

Following this, communication between layers is implemented via micro-ROS and DDS-XRCE, enabling MCUs to publish sensor states and receive control commands over CAN-FD or Ethernet [11]. And this architecture ensures seamless coordination from bare-metal actuators to high-level planning nodes. While this layered integration maximizes both performance and safety, it introduces several challenges. Debugging becomes highly complex because faults can originate from multiple layers, including Linux processes consuming excessive CPU resources, preemptive scheduling issues in RTOS, and data inconsistencies generated by NPUs performing edge AI inference. In addition, clock drift and variable message jitter can destabilize low-level velocity and torque controllers, leading to oscillatory motion, actuator vibration, or positioning overshoot. As a result, system stability depends on robust fail-safe mechanisms. Hardware watchdog timers must detect MCU lock-ups, while RTOS emergency routines must immediately trigger safe-state operations, such as disabling motor outputs or activating braking mechanisms, to prevent error propagation, collisions, or hardware damage.

5. Conclusion

This review has compared bare-metal microcontroller programming, RTOS-based systems, and ROS-based frameworks, confirming that each serves a unique and necessary function in the robotic ecosystem. Bare-metal MCU systems remain the cornerstone of low-cost, direct-hardware interaction. RTOS-based architectures provide the essential deterministic framework for precision and safety. Besides, ROS stands as the premier ecosystem for rapid integration and intelligent perception.

Accordingly, the research questions were addressed by identifying the operational boundaries of each platform: MCUs for simple closed-loop control, RTOS for safety-critical multitasking, and ROS for high-level software modularity. From a practical perspective, the findings suggest that a hybrid architecture is often the most effective solution for robotic applications. However, this study

is limited by its reliance on qualitative literature analysis rather than quantitative benchmarking of latency and power consumption on identical hardware platforms. Future research should experimentally evaluate Micro-ROS performance across different RTOS kernels to quantify communication jitter and system responsiveness. As robotics advances toward more intelligent and human-centered applications, the integration of these embedded control architectures will continue to play a central role in robotic system design.

References

- [1] Quigley, M., et al. (2009). ROS: An open-source robot operating system. *ICRA Workshop on Open Source Software*.
- [2] Koubaa, A. (2016). *Robot Operating System (ROS): The complete reference*. Springer.
- [3] Wolf, M. (2012). *Computers as components: Principles of embedded computing system design*. Morgan Kaufmann.
- [4] Marwedel, P. (2018). *Embedded system design: Embedded systems foundations of cyber-physical systems*. Springer.
- [5] Cheng, A. M. (2002). *Real-time systems: Scheduling, analysis, and verification*. Wiley-Interscience.
- [6] Barry, R. (2016). *Mastering the FreeRTOS real time kernel: A hands-on tutorial guide*. Real Time Engineers Ltd.
- [7] Ganssle, J. G. (2008). *The art of designing embedded systems*. Newnes.
- [8] Cousins, S. (2010). Ros on the move. *IEEE Robotics & Automation Magazine*, 17(4), 13-14.
- [9] Labrosse, J. J. (2002). *MicroC/OS-II: The real-time kernel*. CMP Books.
- [10] Gerkey, B., et al. (2003). The Player/Stage project: Tools for multi-robot and distributed sensor systems. *Proceedings of the 11th International Conference on Advanced Robotics*.
- [11] Purdie, S., et al. (2021). Micro-ROS: Bring ROS 2 to microcontrollers. *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.