EVALUATION OF RESPONSE TIME AND JITTER IN REAL-TIME EMBEDDED SYSTEMS ESP32-S3

Abdusamea Omer
Libyan Center for Electronic Systems, Software, and Aviation Research, Libya
Department of Computer and Information Technology, Faculty of Engineering, Sabratha University, Sabratha, Libya
abdusamea@gmail.com

Salah Eddin Aribi
Faculty of Science, University of Zawiya, Zawiya, Libya
salah.ariby@gmail.com

Ali Alissawi Ahmed AlQudairi
Faculty of Science, University of Zawiya, Zawiya, Libya
Alialgader53@gmail.com

Abstract

This paper presents a hardware-based experimental evaluation of latency and jitter on an ESP32-S3 running FreeRTOS. Four scenarios were examined on a physical development board: baseline single-task execution (S1), concurrent multi-task scheduling (S2), unprotected shared-resource contention (S3), and contention with the Priority Inheritance Protocol (S4). Interrupt-to-task latency was measured with microsecond-resolution software timestamps, and mean, minimum and maximum latency, standard deviation, mean jitter and maximum jitter were computed for 10,000 samples per scenario. The baseline showed a mean latency of 719.57 µs and a mean jitter of 0.49 µs. Concurrent scheduling raised the mean jitter to 2.57 µs, with occasional outliers. Unprotected contention produced the largest variability (mean jitter 88.93 µs, maximum jitter 164.89 µs). Priority Inheritance changed the results only marginally (mean jitter 87.28 µs; standard deviation 89.52 µs against 91.18 µs) and did not reduce the maximum jitter (167.27 µs). The results show that scheduling and synchronization design strongly affect temporal predictability on low-cost embedded platforms, and the methodology offers a reproducible basis for further evaluation.

Keywords: ESP32-S3; FreeRTOS; Latency; Jitter; Priority Inheritance.

1. Introduction

Real-time embedded systems are widely deployed in industrial automation, robotics, intelligent transportation, healthcare devices, aerospace and the Internet of Things (IoT) [1], [2]. The correctness of such a system depends not only on producing logically correct outputs but also on delivering them within predefined timing constraints [3]. Temporal predictability is therefore essential for reliability and safety in time-critical applications.

Latency and jitter are two of the most widely used temporal metrics. Latency is the elapsed time between an event and the system response, whereas jitter is the variation of that response time across repeated executions [4]. Both influence quality of service, synchronization accuracy and stability in industrial control, robotics, wireless sensor networks and cyber-physical systems [5].

The ESP32 family from Espressif Systems has become a popular low-cost platform for embedded and IoT applications because of its dual-core architecture, integrated wireless communication, low power consumption and native support for FreeRTOS [6], [7]. The ESP32-S3 variant adds higher processing performance and improved peripheral support [8]. FreeRTOS offers task scheduling, synchronization, inter-task communication and resource management [9], but its temporal behavior depends on task priorities, context-switching overhead, interrupt handling, execution load and shared-resource contention [10], [11]. Priority inversion and prolonged blocking can erode predictability and cause deadline violations; protocols such as Priority Inheritance are used to bound blocking time [12], [13].

1.1 Related work

Several studies have examined the temporal behavior of ESP32-based and similar systems. Kozioł et al. [14] investigated multi-core scheduling in FreeRTOS-based systems and found that task migration between cores can increase jitter under high workloads; their analysis focused on jitter and did not include a detailed latency evaluation. Bello et al. [15] examined latency-aware scheduling for Industrial IoT on ESP32 platforms and reported that higher priority for time-critical tasks improved response-time stability, but without a comprehensive jitter analysis. García et al. [16] proposed a hardware-assisted method for measuring interrupt-response latency in multi-core systems that improved measurement precision at microsecond granularity; it was not specific to the ESP32 and did not address synchronization effects. Zhang et al. [17] analyzed ESP32-based industrial wireless control under heavy communication load and found that communication overhead increased timing variability and jitter, with limited analysis of synchronization and contention. Kim et al. [18] evaluated core-affinity optimization on the ESP32 and found that pinning critical tasks to dedicated cores reduced timing fluctuation, but considered few scheduling configurations and no synchronization-induced instability. Broader surveys of multiprocessor scheduling and locking are given by Davis and Burns [19] and Brandenburg [20]. Table 1 summarizes these studies.

Table 1. Comparative analysis of previous studies

Platform	Environment	Metrics	Focus	Key findings	Limitations
ESP32	FreeRTOS	Jitter	Multi-core scheduling	Core migration increased timing variability	No latency analysis

Platform	Environment	Metrics	Focus	Key findings	Limitations
ESP32	FreeRTOS	Latency	Task-priority scheduling	High-priority tasks improved response stability	Limited jitter evaluation
Multi-core embedded	RTOS	Interrupt latency	Hardware timing measurement	Improved timing measurement precision	Not ESP32-specific
ESP32	FreeRTOS	Jitter	Wireless communication load	Communication overhead increased jitter	Limited synchronization analysis
ESP32	FreeRTOS	Jitter, scheduling	Core-affinity optimization	Task pinning improved predictability	Limited operational scenarios
ESP32-S3 (this study)	ESP-IDF + FreeRTOS	Latency and jitter	Scheduling and synchronization	Integrated evaluation of scheduling, contention and Priority Inheritance	Single-board experiment

1.2 Research gap and contributions

Previous work has mostly examined isolated aspects of timing behavior (scheduling overhead, interrupt latency, wireless effects or core affinity), and many studies evaluate either latency or jitter rather than both. Because the two metrics together characterize predictability, evaluating only one gives an incomplete picture. In addition, the effect of priority inversion and of Priority Inheritance has received limited integrated experimental analysis alongside scheduling and resource-sharing scenarios on ESP32-S3 hardware under controlled conditions [21]. This study addresses that gap. Its contributions are:

- a hardware-based experimental framework for evaluating temporal behavior of ESP32-S3 FreeRTOS systems;
- a joint analysis of latency and jitter under four scheduling and synchronization scenarios, with 10,000 samples per scenario;
- an experimental examination of the effect of task scheduling, resource contention and task priority on temporal predictability;
- an experimental evaluation of the Priority Inheritance Protocol under priority-inversion conditions; and
- a reproducible methodology for later work on scheduling analysis and synchronization mechanisms.

2. Methodology

A hardware-based experimental methodology was used. The framework runs on a physical ESP32-S3 development board under FreeRTOS and measures latency and jitter under four scheduling and synchronization scenarios.

2.1 Experimental hardware and software environment

Experiments used an ESP32-S3 development board with a dual-core Xtensa LX7 processor. The board was programmed with the Arduino framework built on the ESP-IDF ecosystem. Timestamps were obtained with the ESP-IDF function esp_timer_get_time(), which has microsecond resolution. Wi-Fi and Bluetooth were disabled in all experiments to isolate scheduling and synchronization effects. The configuration is listed in Table 2, and the board is shown in Figure 1.

Table 2. Experimental hardware and software configuration

Parameter	Specification
Microcontroller	ESP32-S3
Processor architecture	Dual-core Xtensa LX7
CPU frequency	Up to 240 MHz
Operating system	FreeRTOS
Development framework	Arduino framework (ESP-IDF core)
Programming language	C/C++
Timing instrumentation	esp_timer_get_time()
Interrupt source	Hardware timer interrupt
Task notification	vTaskNotifyGiveFromISR()
Synchronization mechanism	FreeRTOS mutex (with Priority Inheritance)
Power supply	USB, 5 V
Communication interfaces	Disabled during experiments
Sample size per scenario	10,000 samples



Figure 1. The ESP32-S3-WROOM-1 development board used in the experimental setup, showing the two Type-C interfaces used for firmware flashing and debugging.

2.2 Experimental task architecture

A hardware timer generates periodic interrupts. In the interrupt service routine (ISR) a timestamp is captured and a high-priority FreeRTOS task is notified with vTaskNotifyGiveFromISR(). The task resumes and records a second timestamp, which gives the interrupt-to-task response latency. The architecture comprises a high-priority measurement task, a medium-priority workload task and a low-priority resource-holding task.

2.3 Timing measurement

Latency is the interval between the timestamp captured at the interrupt (t_{event}) and the timestamp recorded when the notified task resumes ($t_{response}$):

$$\boldsymbol{Latency = t_{response} - t_{event}} \quad \textbf{(1)}$$

Jitter is the absolute deviation of each latency measurement from the mean latency over all samples:

$$\boldsymbol{Jitter_i = |T_i - T_{avg}|} \quad \textbf{(2)}$$

where T_i is the i-th latency sample and T_{avg} is the mean latency of the scenario. Each scenario was measured over consecutive timer events.

2.4 Experimental scenarios

Four scenarios (Table 3) were implemented. In S1 a hardware timer periodically activates a single high-priority task with no competing tasks or shared resources, so the measured latency reflects the intrinsic interrupt-handling and scheduling overhead of the platform. In S2 an additional processor-intensive task runs concurrently with the measurement task, adding scheduling pressure and context switching. In S3 three tasks of different priority are used: a high-priority measurement task, a medium-priority workload task and a low-priority resource-owning task. A mutex-protected resource is shared by the high- and low-priority tasks. When the low-priority task holds the mutex and the medium-priority task preempts it, the high-priority task is blocked and priority inversion can occur. S4 keeps the S3 task configuration with the FreeRTOS Priority Inheritance mechanism active: when the high-priority task requests a mutex held by the low-priority task, the kernel temporarily raises the owner's priority to limit blocking.

Table 3. Experimental scenarios

Scenario	Description
S1	Baseline interrupt-to-task latency under isolated single-task execution
S2	Interrupt-to-task latency under concurrent scheduling and processor workload
S3	Resource contention and priority-inversion effects under multitasking with a shared mutex
S4	Resource contention with synchronization-aware execution using the FreeRTOS Priority Inheritance Protocol

2.5 Data collection and statistical analysis

For each scenario, latency was collected over N = 10,000 consecutive executions driven by hardware timer interrupts. The mean, minimum and maximum latency, standard deviation, mean jitter and maximum jitter were computed.

2.6 Validity and limitations

Software-based timestamps (esp_timer_get_time()) have microsecond resolution but can be affected by interrupt activity and operating-system overhead. The same method was applied to all scenarios, so differences between scenarios can be attributed mainly to the scheduling and synchronization mechanism under test; absolute values, however, are specific to this instrumentation and configuration. Wireless subsystems and non-essential peripherals were disabled to limit external noise. All experiments used a single development board in a controlled environment, so variation between boards, firmware versions and power-supply conditions was not examined, and the results describe the evaluated configuration rather than all embedded platforms.

3. Results and Discussion

This section reports the latency and jitter measured on the ESP32-S3 in the four scenarios. Table 4 gives the statistical summary; Figures 2–13 show, for each scenario, the first 1,000 samples, the latency profile over the full capture, and the latency histogram.

Table 4. Latency and jitter statistical summary (N = 10,000 per scenario)

Scenario	Mean (µs)	Min (µs)	Max (µs)	Std (µs)	Mean jitter (µs)	Max jitter (µs)
S1 (Baseline)	719.57	719	720	0.495	0.490	0.57
S2 (Concurrent)	710.70	436	713	16.12	2.574	274.70
S3 (Contention)	657.89	493	736	91.18	88.929	164.89
S4 (Priority Inh.)	661.27	494	737	89.52	87.280	167.27

3.1 Baseline and concurrent scheduling (S1, S2)

The baseline (S1) is very stable: the mean latency is 719.57 µs, the standard deviation is 0.495 µs and the mean jitter is 0.490 µs, with all samples between 719 and 720 µs. With no competing task or shared resource, the scheduler on the ESP32-S3 therefore behaves almost deterministically. Figure 2 shows the first 1,000 samples alternating between 719 and 720 µs. Figures 3 and 4 show the latency over the full capture and its histogram; they also show a later segment of the capture at 726 µs, outside the samples summarized in Table 4.

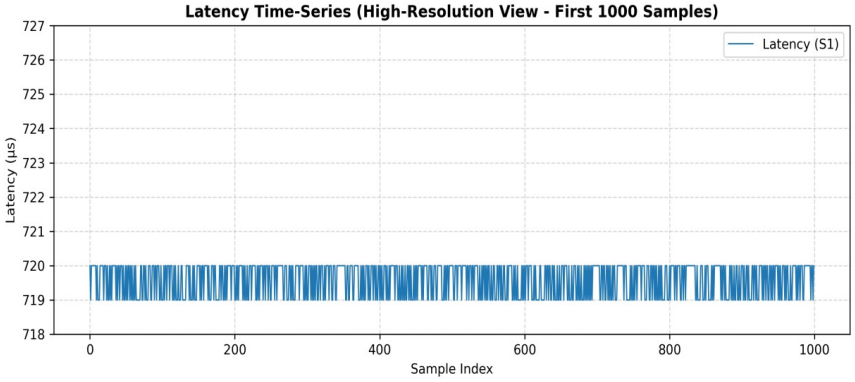


Figure 2. Latency time series, first 1,000 samples, baseline scenario (S1).

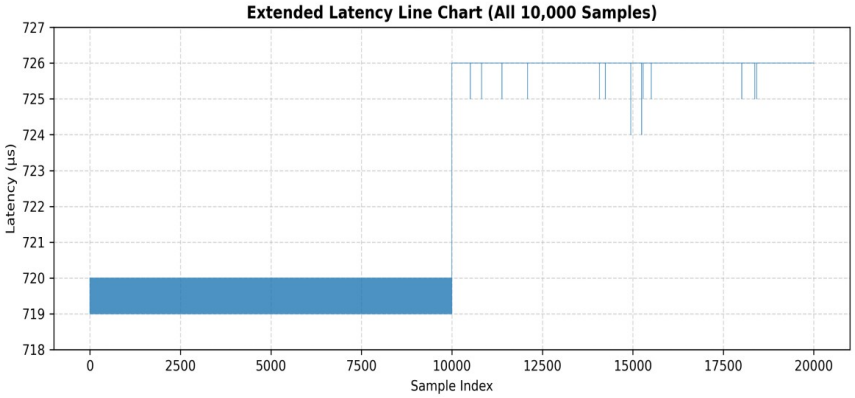


Figure 3. Latency profile over the full capture, scenario S1.

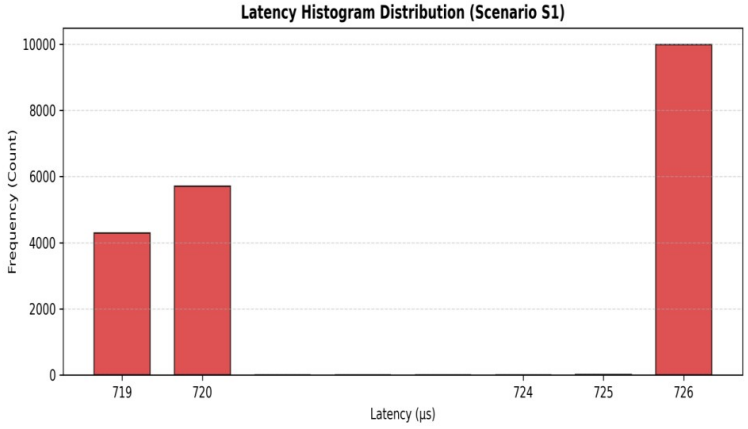


Figure 4. Latency histogram, scenario S1.

In the concurrent scenario (S2) the additional workload task forces the scheduler to perform repeated context switches. The mean latency is 710.70 µs and the mean jitter rises to 2.574 µs, while the standard deviation rises to 16.12 µs. The minimum latency falls to 436 µs, and the maximum jitter of 274.70 µs corresponds to this low-latency outlier (710.70 − 436 µs), not to a delayed response; the maximum latency (713 µs) is close to the mean. Figure 5 shows the first 1,000 samples near the baseline level with

no excursion visible at the scale of the plot. Figure 6 shows episodic groups of lower-latency samples later in the capture, and Figure 7 shows a histogram dominated by one cluster near 710 µs with a sparse tail toward lower latencies. The preemption introduced by concurrency is thus visible mainly as occasional outliers, not as a general widening of the distribution.

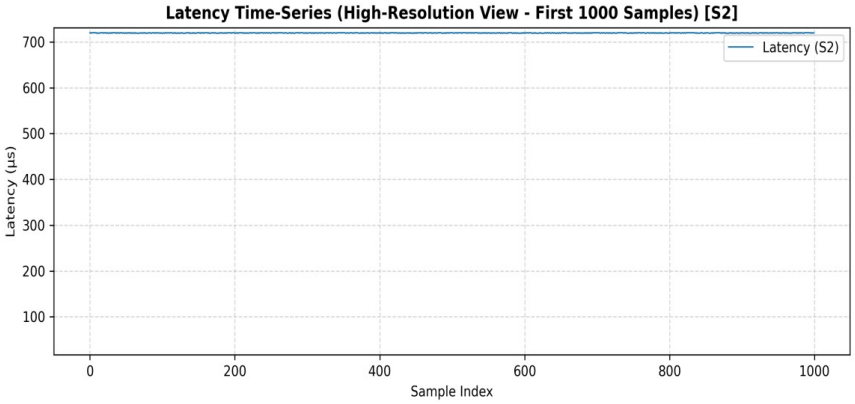


Figure 5. Latency time series, first 1,000 samples, concurrent scheduling (S2).

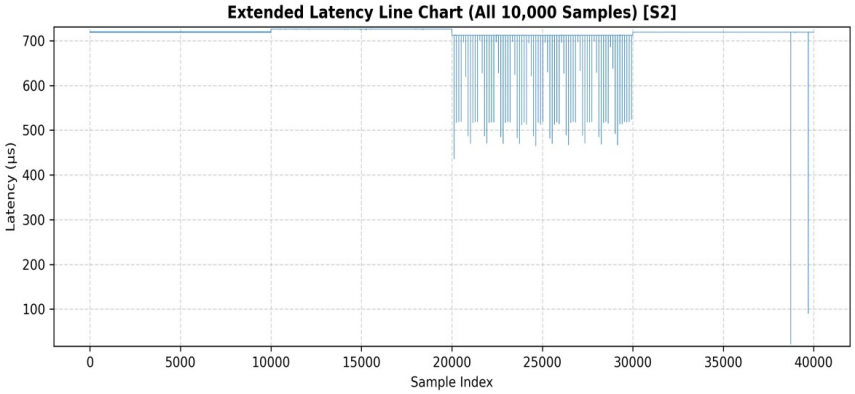


Figure 6. Latency profile over the full capture, scenario S2.

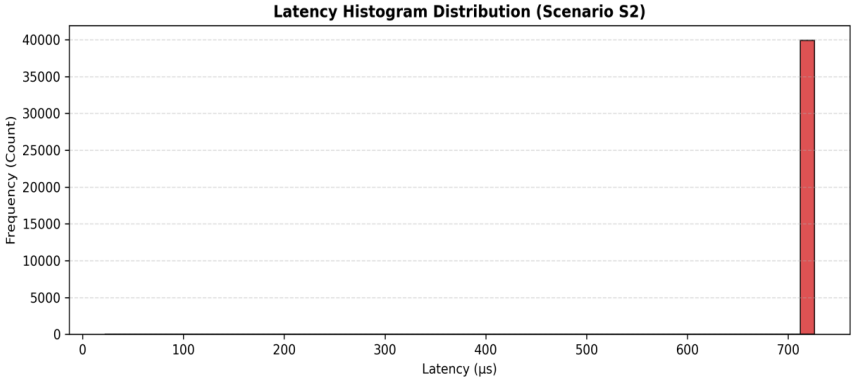


Figure 7. Latency histogram, concurrent scheduling (S2).

3.2 Shared-resource contention and Priority Inheritance (S3, S4)

Shared-resource contention (S3) causes the largest degradation. The standard deviation rises to 91.18 µs, the mean jitter to 88.929 µs and the maximum jitter to 164.89 µs. The mean latency (657.89 µs) is lower than in S1 because a group of samples lies between roughly 500 and 580 µs (Figure 10); a lower mean here does not indicate better timing, since the spread of the samples is far greater. Latency values reach 736 µs. Figures 8 to 10 show the first 1,000 samples, the profile over the full capture, and the histogram, which has a main cluster near 710–735 µs and a secondary cluster at about 520–580 µs.

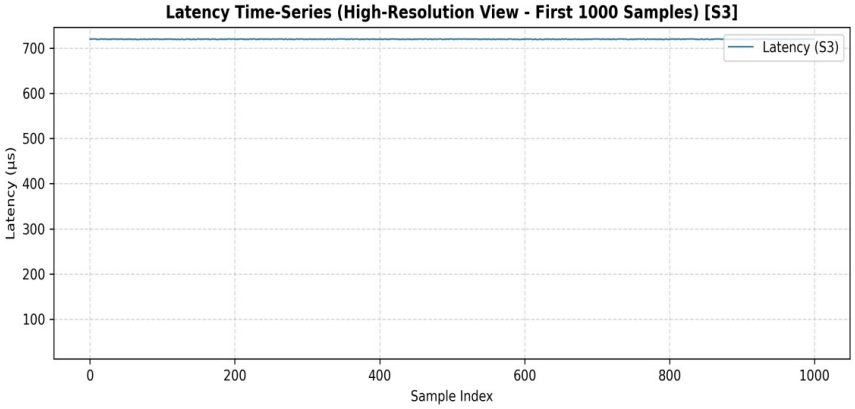


Figure 8. Latency time series, first 1,000 samples, unprotected shared-resource contention (S3).

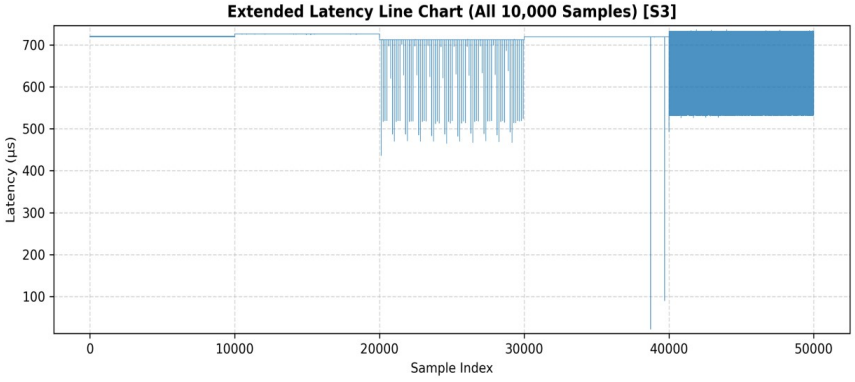


Figure 9. Latency profile over the full capture, scenario S3.

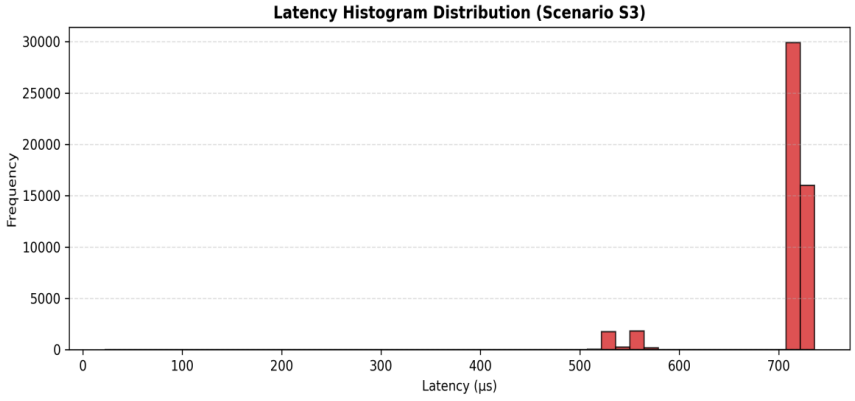


Figure 10. Latency histogram, unprotected contention (S3).

In S4 the Priority Inheritance Protocol temporarily raises the priority of the low-priority mutex holder to that of the requesting task, which prevents the medium-priority task from preempting it while the high-priority task waits. The measured effect is small: the mean latency is 661.27 µs, the standard deviation 89.52 µs (1.8% lower than S3) and the mean jitter 87.280 µs (1.9% lower than S3). The maximum latency (737 µs) and maximum jitter (167.27 µs) are marginally higher than in S3. Figures 11 to 13 show a profile and histogram that closely resemble those of S3. Because the differences are small compared with the spread of the data and no statistical test was applied, they should not be read as clear evidence of improvement; within this experiment, Priority Inheritance provided at most a marginal reduction in variability.

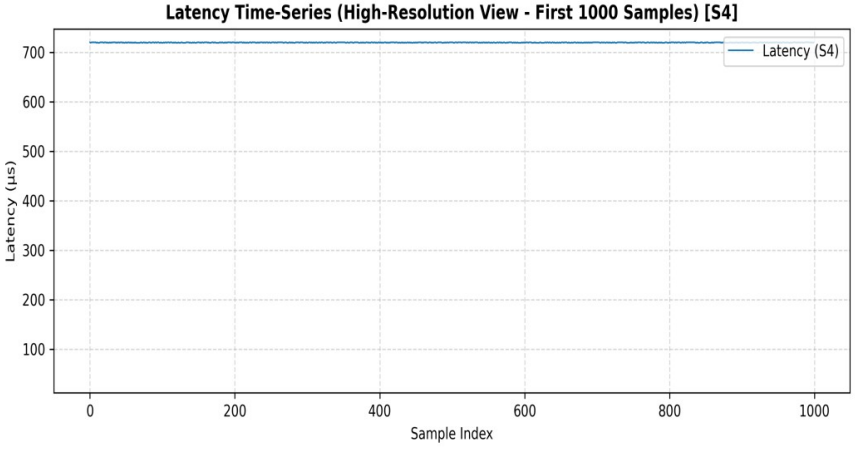


Figure 11. Latency time series, first 1,000 samples, Priority Inheritance Protocol (S4).

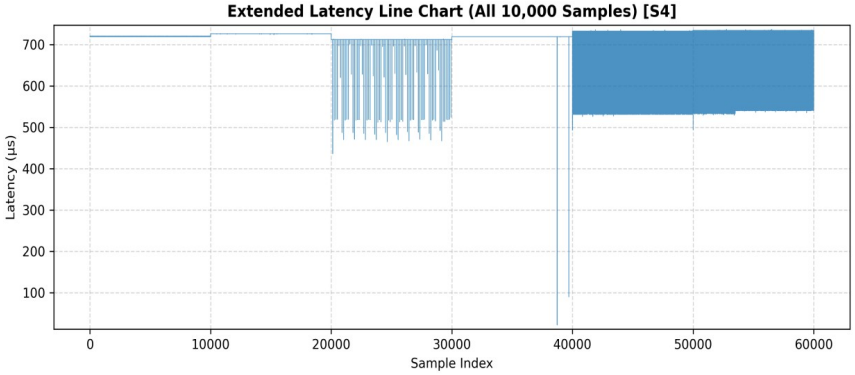


Figure 12. Latency profile over the full capture, scenario S4.

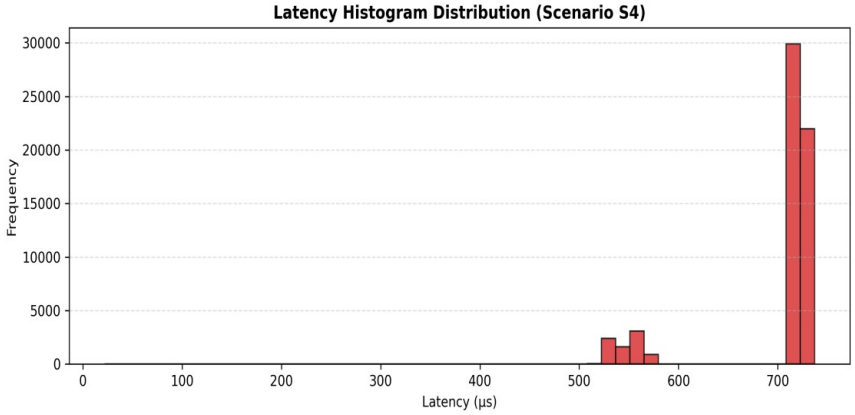


Figure 13. Latency histogram, Priority Inheritance Protocol (S4).

3.3 Integrated discussion

The four scenarios show how software structure affects temporal determinism on the ESP32-S3. Isolated execution (S1) gives near-ideal predictability. Enabling multitasking (S2) adds context-switching effects that appear as occasional outliers and a mean jitter of 2.57 µs. Sharing a resource between tasks of different priority (S3) is the largest source of variability, increasing the mean jitter by a factor of about 36 relative to S2. Priority Inheritance (S4) is the standard remedy for unbounded blocking, but in these measurements it left most of the variability in place. The residual variability suggests that priority protocols should be combined with other design measures, such as core-affinity pinning and short critical sections, as indicated by related work [18]. Confirming this would require further experiments, including external hardware instrumentation such as a logic analyzer or oscilloscope.

4. Conclusion

This study evaluated latency and jitter on an ESP32-S3 under FreeRTOS in four scenarios, using 10,000 samples per scenario. Baseline single-task execution gave a mean latency of 719.57 µs and a mean jitter of 0.49 µs. Concurrent scheduling raised the mean jitter to 2.57 µs, mainly through occasional outliers. Unprotected shared-

resource contention caused severe variability, with a mean jitter of 88.93 µs and a maximum jitter of 164.89 µs. Priority Inheritance lowered the mean jitter slightly, to 87.28 µs, but did not reduce the maximum jitter, so the residual variability remains significant. Task scheduling and synchronization design are therefore critical determinants of temporal predictability on this platform, and the proposed framework provides a reproducible basis for further study.

Future work includes replicating the experiments on other ESP32-S3 boards and ESP32 variants; testing under system load with Wi-Fi, Bluetooth and peripheral traffic; adding metrics such as deadline-miss ratio, full response-time distributions and worst-case execution time; comparing Priority Inheritance with the Priority Ceiling Protocol, core affinity and multi-core partitioning; and validating the software timestamps against logic analyzers or oscilloscopes.

References

[1] IEEE (2021) *IEEE standard glossary of software engineering terminology*. New York: IEEE Standards Association.

[2] ISO/IEC/IEEE (2022) *ISO/IEC/IEEE 24765: Systems and software engineering – vocabulary*. Geneva: International Organization for Standardization.

[3] Liu, J.W.S. (2000) *Real-time systems*. Upper Saddle River, NJ: Prentice Hall.

[4] Kopetz, H. (2011) *Real-time systems: design principles for distributed embedded applications*. 2nd edn. New York: Springer.

[5] Sheikholeslami, A. and Gulak, P.G. (2018) *Embedded systems architecture*. Elsevier.

[6] Espressif Systems (2024) *ESP32 series datasheet*. Shanghai: Espressif Systems.

[7] Espressif Systems (2025) *ESP32-S3 technical reference manual*. Shanghai: Espressif Systems.

[8] Espressif Systems (2024) *ESP-IDF programming guide, v5.x*. Shanghai: Espressif Systems.

[9] FreeRTOS (2023) *FreeRTOS reference manual*. Seattle: Amazon Web Services.

[10] Barry, R. (2024) *Mastering the FreeRTOS real time kernel*. FreeRTOS Foundation.

[11] Buttazzo, G. (2011) *Hard real-time computing systems: predictable scheduling algorithms and applications*. 3rd edn. New York: Springer.

[12] Sha, L., Rajkumar, R. and Lehoczky, J.P. (1990) Priority inheritance protocols: an approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9), pp. 1175–1185.

[13] Baker, T.P. (1991) Stack-based scheduling of real-time processes. *Real-Time Systems*, 3(1), pp. 67–99.

[14] Kozioł, Initials et al. (YEAR) Title; Journal/Conference; Volume and pages.

[15] Bello, Initials et al. (YEAR) Title; Journal/Conference; Volume and pages.

[16] García, Initials et al. (YEAR) Title; Journal/Conference; Volume and pages.

[17] Zhang, Initials et al. (YEAR) Title; Journal/Conference; Volume and pages.

[18] Kim, Initials et al. (YEAR) Title; Journal/Conference; Volume and pages.

[19] Davis, R.I. and Burns, A. (2011) A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys*, 43(4).

[20] Brandenburg, B. (2011) *Scheduling and locking in multiprocessor real-time operating systems*. Ph.D. thesis, University of North Carolina at Chapel Hill, USA.

[21] Baruah, S. (2022) Techniques for multiprocessor global scheduling. *Real-Time Systems*.