Towards a Lightweight ARINC-653 Research Platform Based on a Nanokernel

Léo Lefebvre, Tarek Ould-Bachir, Felipe Gohring de Magalhaes
Department of Computer and Software Engineering,
Polytechnique Montréal, PQ, Canada
{leo-julien-guillaume.lefebvre, t.ould-bachir, felipe-2.gohring-de-magalhaes}@polymtl.ca

***Abstract*—Avionics systems rely on real-time operating systems (RTOSs) and require strict spatial and temporal isolation between applications in accordance with industry standards such as ARINC 653. However, very few open-source RTOSs currently comply with the ARINC 653 standard. This lack of accessible RTOSs and tools creates a significant barrier for academic and experimental research, particularly for studying system overhead and error propagation on modern multicore architectures, where proprietary solutions limit low-level analysis. This work presents the first step in adapting an open-source RTOS called ucx-os, which is a portable and lightweight nano-kernel, to the ARINC 653 standard. We incrementally implement core ARINC 653 partition management services, and then we validate compliance with unit tests on a virtualized processor (QEMU). This work provides a foundation for future experimental research on ARINC 653 RTOSs, bridging the limitations of development on proprietary solutions by enabling the use of an open-source kernel.**

***Index Terms*—ARINC 653, Real-Time Operating System (RTOS), Open-Source**

I. Introduction

Advances in embedded computing have enabled the aviation industry to transition from federated architectures to Integrated Modular Avionics (IMA) to reduce weight, power consumption, and maintenance costs in avionics systems [1]. In the IMA architecture, several functions with different criticality levels share the same resources and must meet deadlines. As a result, these systems rely on Real-Time Operating Systems (RTOSs) compliant with the ARINC 653 standard. This standard defines a strict partitioning model to ensure spatial and temporal isolation and specifies an Application Executive (APEX) interface [2].

ARINC 653 is widely adopted in the industry through commercial solutions such as VxWorks 653 [3]; however, these proprietary solutions restrict access to kernel internals, limiting experimental analysis. Due to their high cost and closed-source nature, proprietary solutions constrain academic and experimental research, particularly in the context of modern hardware, where multicore architectures create new challenges in terms of predictability and resource sharing. For instance, recent studies show the complexity of mitigating multicore memory interference [4] or implementing cache locking strategies, which aim to properly use hardware components without compromising system predictability [5]. Addressing these challenges requires full access to the kernel and system architecture.

Nevertheless, open-source ARINC 653–compliant RTOS implementations are sparse. There are only a few solutions, such as POK [6] or its fork JetOS [7], but they suffer from high complexity or limited hardware support. This limits their deployment and usage for experimentation. It is possible to virtualize architectures [8], but it prevents researchers from testing on real hardware.

To address this gap, this paper explores the adaptation of ucx-os, a lightweight open-source nanokernel, as a research-oriented ARINC 653 platform. This RTOS is deliberately simple and available on several processor architectures. To the best of our knowledge, this is the first work that adapts a nanokernel-based RTOS to ARINC 653 with a minimal overhead design explicitly targeting experimental research rather than certification. Relying on ucx-os, the ARINC 653 layer introduces minimal overhead. Experimental results show no meaningful execution overhead when using the added ARINC 653 services compared to the baseline kernel, making it a suitable foundation for studying the feasibility and overhead of introducing an ARINC 653-compliant layer.

The remainder of this paper is organized as follows. Section II describes the ARINC 653 concept and its related implementations. Section III outlines the ucx-os architecture with the methods to make it ARINC 653 compliant and details the implementation of spatial isolation. Section IV deals with the preliminary results on tests and performance. Section V summarizes this work and presents the next steps. Through these sections, we demonstrate the feasibility of the approach and lay the foundations for an open-source platform supporting future research on avionics RTOS behavior.

II. Background

A. The ARINC 653 Concept

ARINC 653 is a key component of the IMA architecture; this specification is designed and implemented for avionics systems [1]. ARINC 653 defines a layered architecture, illustrated in Fig. 1, where Layer 4 ensures robust task isolation through application partitioning. Layer 3 provides the APEX interface, which applications use to manage computing, storage, and communication resources. This interface connects applications to the operating system (Layer 2), which, in turn, relies on the hardware (Layer 1). The module scheduler handles time isolation between partitions by allocating fixed time windows, while a secondary scheduler manages process

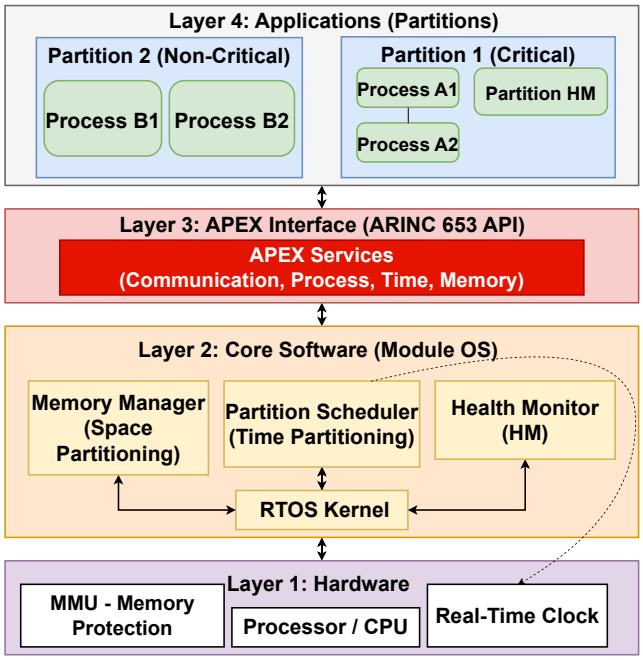

Fig. 1: ARINC 653 architecture

execution within each partition. In parallel, spatial isolation restricts memory access to control resources and confines internal processes.

Data exchange in ARINC 653 relies on distinct inter- and intra-partition communication mechanisms. The inter-partition communication supports a sampling mode for periodically updated data and a queuing mode for steady data between partitions. The intra-partition communication relies on buffers and blackboards to transmit data between processes that are in the same partition.

The integrity of the system and error recovery are handled by the Health Monitoring (HM) service. HM supervises hardware and software states to detect faults, prevent their propagation across partitions, and isolate faulty components. It also reports faults in real-time; therefore, it improves the reliability and availability of the entire system. However, these mechanisms are often discussed at a conceptual level and are rarely evaluated on open, modifiable implementations.

B. Related Work on ARINC 653 implementations

Several projects have explored different approaches, from virtualization to concrete kernel designs, to provide ARINC 653-compliant platforms. Virtualization and simulation are approaches for achieving portable ARINC 653-compliant services. An adaptation of the Xen hypervisor was made to support ARINC 653 [8]. The work aims to reduce certification costs through virtualization technology, and although their work was a prototype, they present future plans for larger simulation environments. Recently, A653MSim was developed as a multicore ARINC 653–compliant simulation platform [9], offering an alternative to costly proprietary solutions. However, platforms are often limited in studying the real-time performance on embedded targets due to the abstraction created by virtualization.

The adaptation of general-purpose RTOS is another strategy. For example, a method to adapt RTEMS, which is an open-source RTOS for embedded systems, has been proposed to offer the APEX interface [10]. Other works have proposed a ”Core Layer” approach based on POSIX to decouple the APEX interface from the underlying OS, promoting code reuse across Linux or RTEMS [11]. However, these solutions remain theoretical, and there is no direct implementation.

Dedicated open-source kernels allow the sharing of a free ARINC 653–compliant RTOS with the community. The most advanced initiative is POK, a kernel designed specifically for partitioning [6]. However, this solution has a heavy architecture and targets a limited set of platforms, such as x86 and PowerPC. Because of these limitations, JetOS was developed as a fork of POK to redesign the scheduler and network stack [7]. Still, these systems remain complex to monitor and debug.

The ecosystem remains an attractive domain, and recent research has proposed a Model-Driven Development (MDD) approach to automate the configuration and validation of ARINC 653 architectures by using tools such as MATLAB and Simulink to generate XML configurations [12]. Moreover, multicore processors have led to new research fields related to interference management; solutions propose execution models to mitigate shared memory contention [4] and cache locking algorithms to improve determinism [5].

These advanced research topics highlight the critical need for a simple, modifiable, and open-source RTOS kernel. Our work on ucx-os aims to provide a lightweight alternative to avoid the complexity of legacy systems such as POK while also enabling execution on real hardware in addition to simulation-based approaches.

III. Methodology

Adapting the RTOS to ARINC 653 requires a detailed understanding of ucx-os and an efficient development workflow for testing and debugging. This section presents the tools and platforms used in the development workflow and outlines the approach adopted to satisfy the first ARINC 653 requirement: spatial isolation and partition management.

A. Experimental Setup and Development Environment

The development and validation flow relies on three core components: the ucx-os kernel, the QEMU emulator, and an ARM-based hardware target (TMS570LC4357) for future performance verification, as shown in Figure 3.

1) The ucx-os Real-Time Kernel: We adapt ucx-os, a lightweight open-source RTOS for resource-constrained embedded systems, featuring high portability and a minimal memory footprint through a clear separation between hardware-dependent and core components.

The overall architecture of ucx-os is shown in Figure 2. It illustrates the different ucx-os layers, in which the kernel, HAL, and application layers are highlighted. This RTOS is

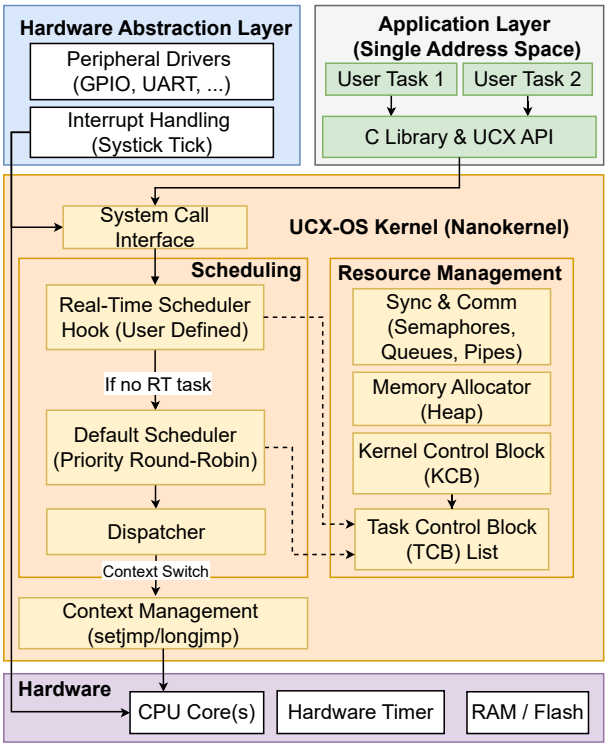

Fig. 2: UCX-OS architecture

structured into three main components. The first is the Hardware Abstraction Layer (HAL), which abstracts processor-specific instructions and peripheral drivers. Currently, ucx-os supports several processor architectures, such as RISC-V, AVR, MIPS, and ARM Cortex-M; thereby enabling cross-platform development and validation. The second component is the kernel core; the kernel implements the fundamental services required for real-time execution. It supports both coroutines and preemptive multitasking. The scheduler manages task execution based on priorities and time constraints and allows the addition of a user-defined real-time scheduler for tasks with other criticality levels. Communication and synchronization are possible thanks to semaphores, mutexes, and message queues. The last component is the user application space; applications run on top of the kernel and interact with system resources through the API.

2) QEMU: We use QEMU as an open-source hypervisor to run applications from specific architectures such as ARM, MIPS, or RISC-V on others like x86. We use this software in the development flow by running ucx-os virtually on a RISC-V target on a computer. Therefore, it is very simple and fast to test new code; moreover, with the help of a Makefile, we can automate the cross-compilation, execution, and debugging of an application on ucx-os on an x86 laptop, as shown in Figure 3. One more advantage is the ease of linking gdb by running QEMU in debug mode; it offers a simple way to explore memory and the execution of a program.

3) TMS570LC4357 target: When ARINC 653 services pass the unit tests on QEMU, real hardware verification and validation are executed on the TMS570LC4357 microcontroller to validate compliance and measure performance metrics, as shown in Figure 3. The TMS570LC4357 is a 32-bit microcontroller from Texas Instruments based on the Arm Cortex-R5F architecture. This target was not available on ucx-os; therefore, we have adapted the ucx-os low-level files to run programs on this target.

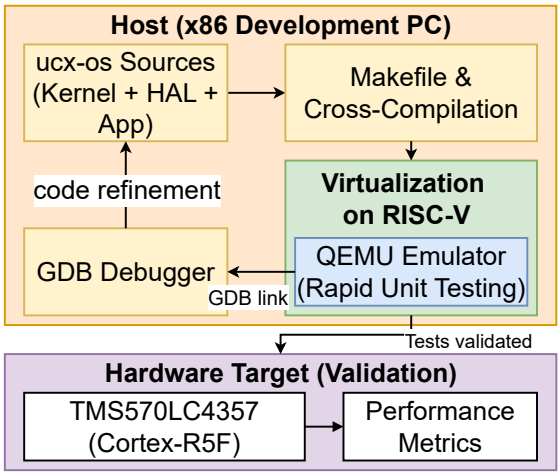

Fig. 3: Development flow

B. Implementation of Partition Management and Spatial Isolation

The first requirement of ARINC 653 is partition management [1]. It enables spatial isolation between programs and exposes functions to manage partitions: GET_PARTITION_STATUS, SET_PARTITION_MODE and GET_MY_PARTITION_ID. Spatial isolation has been achieved thanks to static memory segmentation defined during link time and a software layer to manage contexts at the kernel level. This approach guarantees that each partition is executed in its respective range of addresses. We have modified the linker script (riscv32-qemu.ld) of the qemu target in ucx-os to define these memory regions. For instance, to configure two partitions, we can define the following memory areas with their access rights (RWX):

- Kernel: Confined at `0x80000000` (16 MB) in the RAM area
- Partition 1 (P1): Code (RX) at `0x81000000` (1 MB) and Data (RW) at `0x82000000` (1 MB)
- Partition 2 (P2): Code (RX) at `0x83000000` (1 MB) and Data (RW) at `0x84000000` (1 MB)

Specific sections map code and data into these regions. At the software level, the function *partition_init* configures the environment of a partition; particularly, a Partition Control Block (PCB) is allocated to store metadata and memory requirements of the partition. Each partition has an entry point to a main process, which is in the code region of its partition. This process uses the data section as a stack to confine local variables and function calls inside the partition and avoid memory corruption.

```
--- START TEST SEQUENCE P1 ---
[TEST 1] GET_MY_PARTITION_ID...
-> Expected: 1, Received: 1 [PASS]
[TEST 2] GET_PARTITION_STATUS (Initial)...
-> Expected: IDLE, Received: IDLE [PASS]
[TEST 3] SET_PARTITION_MODE(NORMAL)...
-> Return Code: NO_ERROR [PASS]
[TEST 4] GET_PARTITION_STATUS (Post-Switch)...
-> Expected: NORMAL, Received: NORMAL [PASS]
--- END TEST SEQUENCE P1 ---
```

Fig. 4: Partition Management API Execution Trace

To validate this work, several unit tests are performed. For instance, there are unit tests to validate the static separation between partitions, the correct results of partition API management, and performance characteristics. This approach establishes the foundation for extending ucx-os toward additional ARINC 653 requirements.

IV. Experimental Results

This section evaluates spatial isolation in ucx-os, validates the behavior of the partition management API, and reports preliminary performance measurements based on runtime memory, API, and latency tests.

A. Validation of Spatial Isolation Configuration

To verify the spatial isolation configuration in ucx-os, we compared the memory regions defined in the linker script with the addresses observed at runtime for each partition. Table I shows the results for Partition 1 and Partition 2.

TABLE I: Static vs. Runtime Memory Address Comparison

Part.	Section / Object	Base Addr.	Observed Addr.	Status
Part. 1	Code Base (.text)	0x81000000	0x81000000	**PASS**
	Data Base (.data)	0x82000000	0x82000000	**PASS**
	Var: cnt	0x82xxxxxx	0x8203ffe4	**PASS**
Part. 2	Code Base (.text)	0x83000000	0x83000000	**PASS**
	Data Base (.data)	0x84000000	0x84000000	**PASS**
	Var: cnt	0x84xxxxxx	0x8403ffbc	**PASS**
Kernel	Base Addr.	0x80000000	0x80000000	**PASS**
	Var: cnt	0x80xxxxxx	0x80004260	**PASS**

Table I confirms that the ranges are correct and that the local variable *cnt* is allocated within the expected address ranges.

B. Validation of Partition Management Services

ARINC 653 defines three core services to manage partitions, and we tested their behaviors via unit tests on QEMU, as shown in Figure 4.

C. Performance Considerations

We compared the execution time between the native ucx-os and the adapted version during both initialization and runtime. For each experiment, 200 timing measurements were collected, and average values were computed, as shown in Figure 5. The term *Partition* refers to the adapted ucx-os implementation, whereas *Task* refers to the native version. Runtime measurements were obtained by executing equivalent programs on both versions, which consisted of retrieving the partition or task identifier using ARINC 653 or native API calls, respectively.

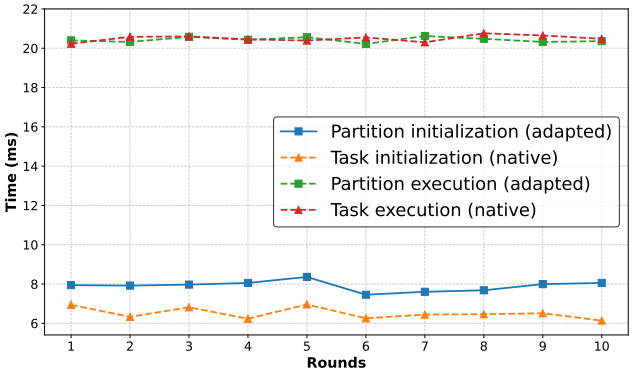

Fig. 5: Initialization and Execution Latency Comparison

Figure 5 and Table II show that during runtime, ARINC 653 API calls do not introduce measurable runtime overhead in the tested configuration. However, the initialization of a partition incurs an overhead of about 1.36 ms compared to native task initialization. This overhead is mainly due to the additional data structures and initialization steps required for partition management compared to native task creation. Nevertheless, this overhead occurs during the application preparation phase. Once the application is ready to execute, when real-time constraints and determinism are most critical, no significant impact on execution behavior was observed under the tested conditions. All measurements were performed on a QEMU-based RISC-V target under identical conditions.

TABLE II: Performance Results

Time (ms)	Average (ms)
Partition initialization	7.90
Task initialization	6.51
Partition execution	20.43
Task execution	20.50

V. Conclusion

This paper presented an initial step towards adapting the open-source RTOS ucx-os to the ARINC 653 standard. Partition management services were implemented, and spatial isolation was validated on a virtualized QEMU-based platform. Experimental results confirm correct isolation behavior and show a minimal impact on execution latency. This work establishes a foundation for an open-source, ARINC 653-compliant platform aimed at supporting experimental research in avionics systems. Future work will extend this effort in two directions. First, the proposed mechanisms will be validated on physical hardware targets. Second, temporal isolation through the implementation of a dual-level scheduler for partitions and processes will be addressed, followed by the integration of additional APEX services, such as inter-partition communication and health monitoring.

REFERENCES

[1] P. J. Prisaznuk, "ARINC 653 role in integrated modular avionics (IMA)," in *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*, 2008, 1.E.5-1-1.E.5–10.

[2] Airlines Electronic Engineering Committee, "ARINC specification 653: Avionics application software standard interface, part 1: Required services," SAE ITC, Warrendale, Pennsylvania, Standard 653P1-6, 2024.

[3] W. Ruan and Z. Zhai, "Kernel-level design to support partitioning and hierarchical real-time scheduling of ARINC 653 for VxWorks," in *2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing*, 2014, pp. 388–393.

[4] S. Park, M.-Y. Kwon, H.-K. Kim, et al., "Execution model to reduce the interference of shared memory in ARINC 653 compliant multicore RTOS," *Applied Sciences*, vol. 10, no. 7, p. 2464, 2020.

[5] A. T. A. Dugo, J.-B. Lefoul, F. G. De Magalhaes, et al., "Cache locking content selection algorithms for ARINC-653 compliant RTOS," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 5s, pp. 1–20, 2019.

[6] J. Delange and L. Lec, "POK, an ARINC653-compliant operating system released under the BSD license," in *13th Real-Time Linux Workshop*, vol. 10, 2011, pp. 181–192.

[7] K. Mallachiev, N. Pakulin, and A. V. Khoroshilov, "Design and architecture of real-time operating system,", vol. 28, no. 2, pp. 181–192, 2016.

[8] S. H. VanderLeest, "ARINC 653 hypervisor," in *29th Digital Avionics Systems Conference*, 2010, 5.E.2-1-5.E.2–20.

[9] J.-B. Lefoul, F. G. de Magalhães, B. Bhatnagar, et al., "A653MSim: An ARINC 653 multicore simulator," in *Simulation Tools and Techniques*, A. A. Juan, J.-L. Guisado-Lizar, M.-J. Morón-Fernández, et al., Eds., Cham: Springer Nature Switzerland, 2025, pp. 285–304.

[10] J. Rufino, S. Filipe, M. Coutinho, et al., "ARINC 653 interface in RTEMS," in *Proc. DASIA*, 2007.

[11] S. Santos, J. Rufino, T. Schoofs, et al., "A portable ARINC 653 standard interface," in *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*, 2008, 1.E.2-1-1.E.2–7.

[12] B. Lukić, S. Friedrich, and U. Durak, "A streamlined approach toward automated generation and validation of ARINC 653–compliant avionics configurations," *Journal of Aerospace Information Systems*, vol. 22, no. 5, pp. 314–324, 2025.