IOT-653: ENHANCING SECURITY, RELIABILITY AND REAL-TIME PERFORMANCE IN RESOURCE CONSTRAINED IOT DEVICES

Tom Springer and Peiyi Zhao

Fowler School of Engineering, Chapman University, Orange CA, USA

ABSTRACT

This paper presents a novel approach designed to enhance the security, reliability, and real-time performance of resourced-constrained Internet-of-things (IoT) devices. We improved system performance by implementing a lightweight partitioning approach that emulates key features of the ARINC-653 standard without all the overhead. Our solution was tested on Linux and VxWorks platforms to ensure compatibility and effectiveness. Additionally, as part of the test and evaluation efforts, we created an IoTbased Simulation Testbed using Simics, which allowed us to simulate and evaluate the overall system behavior in a controlled environment. Our initial results verified that using a lightweight partitioning approach based on the ARINC-653 standard enhanced reliability and security. These findings validate our solution as a viable method for improving the performance of IoT devices, offering a robust and efficient framework for future IoT deployments. The potential impact of our work is significant, as it can pave the way for more secure and reliable IoT deployments.

KEYWORDS

Internet-of-Things, Security, Reliability, Resource-Constrained Devices, ARINC-653 Standard

1. INTRODUCTION

Over 18 billion Internet of Things (IoT) devices are connected worldwide, a number that is projected to experience a substantial increase to 32 billion by 2030 [1]. This growth in IoT devices is not just a number, it's a potential disruptor, driving the transformation of numerous industries by enhancing connectivity, automation and enabling real-time data processing. By using remote monitoring systems or wearable health trackers to collect data and perform realtime analysis, industries such as healthcare are being revolutionized by IoT technology, promising improved patient outcomes while reducing the strain on overburdened healthcare systems. Other examples include industrial automation integrating intelligent sensors and robotics for improved manufacturing analytics, boosting productivity, and reducing operational costs. IoT applications are also critical to the future development of smart cities where smart grids can improve energy efficiency, intelligent transportation systems can reduce traffic congestion, and connected public services can provide better public safety and services. Overall, the growth in IoT devices and applications will play an increasingly critical role in making these industries more efficient, responsive, and sustainable.

However, to advance this transformation, IoT devices must be reliable, secure, and real-time to guarantee system performance, protect sensitive data, and provide timely responses to critical applications. This need for timely responses underscores the importance of our work in the context of IoT performance. However, ensuring reliability, security, and real-time performance in these resource-constrained devices presents several significant challenges. For instance, many

DOI : 10.5121/ijesa.2024.14401

IoT devices have limited processing power, restricting the capability of these devices to implement traditional compute-intensive encryption or intrusion detection algorithms, leading to unauthorized exposure or loss of private information [2]. Additionally, the wide variety of IoT devices makes adopting uniform security standards difficult and even more critical across heterogeneous platforms. IoT devices' resource-constrained nature can also affect their reliability, where limited memory and processing power can lead to performance bottlenecks. Maintaining energy efficiency is another critical challenge because many IoT devices use battery power, limiting their lifespan and affecting reliability.

To address the issues related to the growth of IoT we propose that many of the challenges associated with these systems can be resolved by applying a standardized lightweight partition based computing framework. Partitioning techniques such as application segregation where memory and CPU time can be utilized in an isolated environment. Application isolation could ensure that a fault or failure in one application does not affect others, enhancing overall system reliability. Resource allocation prioritization is another example where critical applications could be properly allocated, which ensures that essential functions receive the necessary resources to operate effectively, even under constrained conditions. Partitioning can also be leveraged to help improve security by isolating sensitive data and operations where unauthorized access or breaches in one partition would not compromise the entire system. System scalability is another example where new applications could be added without affecting existing ones, making it easier for the IoT system to scale as new functionalities or devices are introduced. System certification and device maintenance are additional benefits because partition boundaries make it easier to certify and maintain the system, reducing complexity and making the system more manageable. Each partition can be developed, tested, and certified independently, further reducing complexity and cost. Lastly, partitioning resources can be managed more efficiently by optimizing performance and power consumption, which are crucial for resource-constrained IoT devices.

As a consideration for adopting such an applicable framework, ARINC-653 stands out among partitioning mechanisms due to its proven reliability in safety-critical environments. ARINC653 supports strong isolation and deterministic behavior with a well-documented API that has been extensively tested and validated. The standard specifies various partitioning strategies, including time and space partitioning. It allows for custom solutions based on specific system requirements, making it an ideal candidate for managing the complexities of resource constrained IoT environments. The ARINC-653 specification has a proven track record in the aviation industry that ensures the reliability and safety of avionics systems. Its robust space and time partitioning framework provide a compelling choice for managing other complex domains like IoT.

Unfortunately, a fully compliant ARINC 653 implementation is not always practical for resource-constrained IoT devices. One reason is the high cost of a real-time operating system that complies with the ARINC 653 standard. Operating system implementations like VxWorks 653 are designed for high-assurance, safety-critical environments but have significant licensing and maintenance costs. These licensing costs can be prohibitive for IoT applications, which often operate on tight budgets. The complexity of ARINC-653 systems is another reason, which includes extensive partitioning and scheduling mechanisms that could be excessive for many IoT applications. Resource usage is another issue because typical ARINC-653 implementations must handle multiple high-critical applications simultaneously, which demands substantial CPU, memory, and power resources. System certification challenges present another roadblock because the rigorous certification process (e.g., DO-178C) associated with ARINC-653 systems adds another layer of cost and complexity. System certifications are essential for avionics but may not be necessary or feasible for many IoT applications. Given these challenges, a lighter weight partitioning solution could be more appropriate for resource-constrained IoT devices.

As a result of the challenges previously mentioned, we present a lightweight partitioning framework that emulates the key features of ARINC-653 and adapts it to IoT devices. Our framework, IoT-653, is developed as a set of libraries and services that implement the scheduling and partitioning of ARINC-653 on top of specific POSIX-compatible operating systems. This framework provides the necessary interfaces and mechanisms to create partitions and manage inter-partition communication. The approach ensures enhanced real-time performance, fault isolation, and security while maintaining compatibility and interoperability with existing POSIX-compatible operating systems. This solution offers a scalable and efficient way to implement ARINC-653 functionalities without the overhead of a fully compliant ARINC-653 system.

The remainder of this paper describes the main contributions of our approach, including the lightweight ARINC 653-inspired solution, its implementation, and key performance results. Section 2 details the Background and Related Work, Section 3 details the Architecture of the partitioning framework, Section 4 provides implementation details while Section 5 describes the IoT Simulation Testbed we developed to test our framework, and Section 6 provides Evaluation and Result analysis. Section 7 summarizes with Conclusions and Future Work.

2. BACKGROUND AND RELATED WORK

ARINC-653 is a specification for an application executive used in avionics systems based on the integrated modular avionics (IMA) platform (i.e., multiple software systems on the same processor platform) for modern aircraft. The overall architecture [3][4], depicted in Figure 1, is divided into several layers, which include the hardware (processor, memory, I/O devices, …), the hardware abstraction layer (HAL) or board support package (BSP), and the operating system. The ARINC-653 standard specifies space and time partitioning support in safety-critical avionic real-time applications. The standard defines a set of software interfaces as the application executive interface (APEX) that an ARINC-653- compliant operating system provides for application development. The APEX interface defines how to create applications to meet ARINC653 requirements. The primary components of the APEX interface include partition management, process management, time management, inter and intra partition communication, and health monitoring

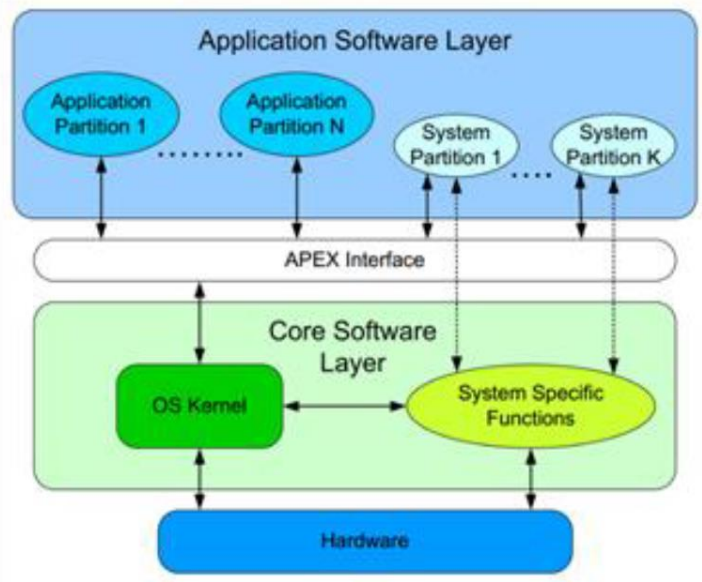


Figure 1: ARINC-653 OS Architecture Specificatio

Partition management is the cornerstone of ARINC-653, where each partition operates as a separate, distinct entity, encapsulating a specific software application. Separation between partitions prevents cross-partition interference, thereby protecting the integrity of critical systems and ensuring the safety and security of the avionic systems. Process management, another component, is essential for ensuring avionic systems' reliable and safe operation. It involves creating and initializing processes within partitions using the APEX interface, which assigns attributes like priority and stack size to each process. The scheduling system uses a twolevel hierarchical scheduling scheme where the first level schedules each partition in a fixed, cyclic manner, and the second level schedules each process within a partition in a fixed-priority pre-emptive manner. Processes can exist in various states, such as dormant, ready, running, waiting, or suspended, with state transitions managed by the APEX interface. The time management component uses temporal partitioning, where the system's execution time is divided into discrete intervals known as time frames or time slots. Each partition is allocated a specific time frame to execute its tasks without interruption from other partitions, ensuring temporal isolation and predictability.

Inter-partition communication involves the exchange of data between different partitions. The communication is handled through message-passing mechanisms using ports and channels. There are two primary types of ports: sampling ports and queuing ports. Sampling ports are used for periodic data updates, where the most recent message is retained, ensuring that the latest data is always available to the receiving partition. On the other hand, queuing ports store messages in a queue, allowing multiple messages to be buffered and processed in the order they were sent. Intra-partition communication, on the other hand, refers to exchanging data within the same partition to communicate and synchronize. This partition includes buffers, which provide a queue for passing data messages, and blackboards, which store only the most recent message. Semaphores are used for synchronization, ensuring processes can coordinate their actions. Additionally, events signal state changes or specific conditions to other processes within the same partition.

The health monitoring component of ARINC-653 is designed to ensure the reliability and safety of the avionics system by detecting, reporting, and responding to faults or failures. The health monitoring process continuously monitors various components and processes to detect anomalies or deviations from expected behavior. It logs the fault details and notifies relevant processes or systems if a fault is detected. The system then isolates the source of the fault to prevent it from affecting other parts of the system, which may involve isolating the faulty component or process. Importantly, the system is equipped to take recovery actions to restore regular operations, such as restarting processes, reconfiguring resources, or switching over to redundant systems.

Some existing technologies enhance the reliability and security of resource-constrained devices but are limited in scope and fragmented across hardware and software solutions. Memory protection units (MPUs) are one example that is integrated into the CPU and designed to enforce memory access control. MPUs provide a lightweight alternative to the larger memory management units (MMUs), which handle more complex tasks like virtual memory management. In contrast, MPUs focus only on memory protection, making them more suitable for resource-constrained devices. While MPUs offer strong hardware-enforced isolation, they are hardware-dependent and unavailable on all embedded processors. Therefore, an ARINC-653 software-based partitioning solution is more flexible and portable across all embedded platforms.

Software fault isolation (SFI) is a software-based technique used to enhance the reliability and security of embedded systems by isolating faults within software components. Software such as WebAssembly (Wasm) [5] for microcontrollers enforces the isolation of applications using sandboxing, preventing faults in one part of the system from affecting others. Ultra-lightweight

SFI [6] is a specific implementation of SFI designed for resource-constrained embedded devices, such as those used in IoT applications like smart home devices, industrial sensors, or wearable health monitors. SFI techniques work by adding runtime checks to an application to enforce isolation. However, they are not optimized for real-time performance and can introduce additional computational overhead and extra layers of complexity. In comparison, a lightweight ARINC-653 solution is designed for real-time and safety-critical systems, providing predictable and deterministic behavior.

Other isolation techniques include mixed-criticality task-based isolation (MCTI) [7], a hardware/software co-design solution devised to improve the reliability and predictability of embedded systems that handle tasks with different levels of criticality. Domain-enforced memory isolation (DEMI) [8] provides another technique designed to enhance the security and reliability of embedded systems by isolating memory into distinct domains. However, MCTI techniques require more complex hardware and software support, whereas lightweight ARINC653 can achieve similar isolation goals with lower overhead and a more straightforward implementation.

In addition to spatial partitioning, time partitioning techniques are also available, such as the constant bandwidth server (CBS) [9][10], which allocates CPU time to tasks based on their criticality and timing requirements. These partitioning techniques ensure high-priority tasks receive the necessary resources while maintaining overall system stability. Server-based scheduling techniques offer flexible time partitioning but are designed for traditional computing environments, where resources such as CPU, memory, and bandwidth are relatively abundant. Other isolation techniques include mixed-criticality task-based isolation (MCTI), which is a hardware/software co-design solution aimed at improving the reliability and predictability of embedded systems that manage tasks with varying levels of criticality. Another method is domain-enforced memory isolation (DEMI), designed to enhance the security and reliability of embedded systems by segregating memory into distinct domains. However, MCTI techniques necessitate more complex hardware and software support, while the lightweight ARINC-653 can achieve similar isolation goals with lower overhead and a more straightforward implementation.

In summary, a lightweight ARINC-653 is more portable and easier to integrate than hardwarebased solutions like MPUs. Unlike other software-based techniques like SFI and Wasm sandboxing, an ARINC-653-based solution provides predictable and deterministic behavior, which is crucial for real-time applications. This predictability and determinism, combined with its simplicity, offer a more straightforward and cost-effective solution than the more complex mixed-criticality and server-based scheduling techniques. Overall, a lightweight ARINC 653 framework can provide a comprehensive and cohesive approach, combining the benefits of software-based partitioning and isolation with the efficiency and simplicity needed for resource constrained IoT environments.

3. ARCHITECTURAL OVERVIEW

The IoT-653 framework is a lightweight POSIX-based implementation of ARINC-653 principles designed to enhance security, reliability, and real-time performance in resource constrained IoT systems. IoT-653 integrates a lightweight version of DDS [11] (Data Distribution Service) to facilitate scalable and efficient data exchange, providing a robust middleware layer for real-time communication. The framework emulates time and space partitioning by managing independent execution environments, each with dedicated processes, communication channels, and synchronization primitives. A singleton-based partition management system ensures strict isolation and scalability, allowing seamless transitions between partition states.

Inter-partition communication with mechanisms like blackboards, buffers, sampling ports, and queuing ports is enhanced with DDS-based integration tailored for specific messaging requirements, such as asynchronous messaging or periodic data exchange. Fault isolation is achieved through robust error handling, dedicated resource management, and custom signal handling, ensuring faults remain within individual partitions. By combining the benefits of ARINC-653 principles with DDS, the IoT-653 provides a comprehensive framework for building secure, reliable, and scalable real-time systems optimized for IoT devices.

3.1. Time and Space Partitioning

Time partitioning allocates specific time slots to each partition, ensuring every application gets a predictable amount of CPU time. Space partitioning assigns memory regions to each partition. This isolation ensures that applications cannot interfere with each other's memory, enhancing system stability and security. IoT-653 implements time partitioning using process scheduling, task timing attributes, partition states, and scheduling events. IoT-653 supports two scheduling policies: round robin, where time slices are evenly distributed among processes of the same priority, and fixed priority, where processes with higher priority execute until they yield or are preempted by a higher-priority process. These policies are programmable, enabling flexibility for specific use cases. Each process is managed by a task attribute vector, which defines the task period, task execution time, and task deadline (HARD, SOFT). IoT-653 logs process timing details, allowing for the inspection and validity of real-time behavior. Each partition operates in modes and transitions between these modes to ensure system initialization, idle time management, and active operation. A partition mode set function controls these transitions, ensuring tasks are activated or paused based on the partition state. Scheduling events synchronize task scheduling within a partition, contributing to the deterministic execution of processes.

Space partitioning in IoT-653 is supported by memory isolation, synchronization, and interprocess communication mechanisms. Isolation is achieved using partition-level mutexes that ensure that critical sections within a partition are accessed by one task at a time and private process descriptors encapsulating process-specific details. Vectors (dynamic arrays that can grow or shrink in size as needed) of read/write locks are used to ensure task synchronization, and an event vector is used to signal events, ensuring controlled interaction between processes. Partition communication is achieved by using shared message spaces that allow only one process to read or write at a time, with message size and count constraints. Additionally, each communication channel is uniquely identified and managed, ensuring no overlap or interference occurs.

The partitioning approach in IoT-653 supports deterministic behavior in processes that execute in strict time intervals, guaranteeing predictability for real-time applications. ARINC-653 isolation is supported because faults in one process or partition are contained, preventing cascading failures. Additionally, the lightweight design enables deployment on resourceconstrained IoT devices without sacrificing partitioning principles.

3.2. Partition Communications

IoT-653 implements several features to emulate inter-partition and intra-partition communication (IPC), a key component of ARINC-653 systems. These features ensure safe, predictable, and efficient exchange mechanisms between processes while maintaining the integrity and isolation of individual partitions. Message exchange mechanisms, including Blackboards, Buffers, Sampling Ports, and Queuing Ports, are designed to provide a high level of predictability.

Blackboards are used for asynchronous communication between processes and functions as named shared memory regions where one process can post a message, and others can read it at

their convenience. Messages remain on the blackboard until explicitly cleared or overwritten, enabling multiple processes to read the same data. Each blackboard is identified by a unique name stored in a buffer, and the size attribute ensures that memory is bounded, preventing excessive resource usage. Buffers provide FIFO-based communication where messages are queued for processing by recipient tasks. Messages are stored in a fixed-size queue with blocking and non-blocking access. Buffers are managed in a vector, ensuring each partition has its own isolated communication queue. Sampling ports are designed for periodic data communication. They hold a single message overwritten with each new send operation, ensuring the latest data is always available. Queuing ports allow messages to be sent and received in order without overwriting. They are useful for tasks requiring sequential processing of events or data.

3.3. Fault Isolation

Fault isolation is a crucial aspect of any real-time system, especially in resource-constrained IoT environments where failures in one part of the system can have significant ripple effects. IoT653 implements multiple mechanisms to ensure faults are contained within a single partition or process, preserving the system's reliability and stability.

IoT-653's use of custom signal handlers for signal handling and recovery is a testament to its robustness. By registering handlers for critical signals, IoT-653 enables partitions to respond gracefully to interruptions. This includes halting the execution of processes without corrupting shared resources, logging errors, and safely releasing communication channels. The process level termination further ensures that if a partition fails, the process is stopped, and its resources are reclaimed, while other processes in the same or different partitions remain unaffected.

3.4. Platform Compatibility

The IoT-653 framework provides a middleware class for interfacing with the emulated APEX interface to achieve platform-specific functionality. The middleware abstracts platform-specific details through helper methods and namespaces, allowing the same interface to be compiled and executed across VxWorks and Linux with minimal modifications. Health monitoring functions provide fault detection and recovery mechanisms, ensuring robust operation on both VxWorks and Linux. The middleware employs reusable functions for common tasks like process creation, event signaling, and message queuing. Using reusable functions reduces code duplication and ensures consistent behavior across platforms. Table 1 provides a comparison of the high-level differences between the VxWorks and Linux implementations.

3.4.1. VxWorks Integration

The middleware interfaces with the VxWorks real-time kernel by leveraging scheduling attributes in the APEX implementation to create scheduled tasks using preemptive prioritybased or round-robin scheduling. Partition and process management functions allow direct control of tasks within VxWorks partitions. This maps to the ARINC-653-based task management features that VxWorks natively supports. The middleware creates custom signal handlers that reattach VxWorks signals, allowing for graceful interruption and fault isolation. Additionally, event and semaphore management functions utilize VxWorks synchronization primitives, ensuring compatibility with the kernel.

3.4.2. Linux Integration

The middleware is based on the POSIX APIs, which are fully supported by Linux, allowing for straightforward integration. The IoT-653 API utilizes POSIX-compliant system calls, ensuring

compatibility across Linux-based distributions. ARINC-653 processes are implemented as threads, which leverage the pthreads library for lightweight execution—synchronization mechanisms like mutexes and semaphores are directly mapped to the POSIX pthread APIs, enabling efficient inter-thread communication. Signal handling functions bind Linux signals to custom handlers, providing fault tolerance and process recovery similar to VxWorks. Shared memory and queuing mechanisms are emulated using kernel primitives, such as shared memory segments or message queues, to mimic the ARINC-653-style IPC.

Table 1 :Comparison of VxWorks and Linux Integration

ARINC-653 Feature	**VxWorks**	**Linux**
Process Management	Tasks for DKMs and RTP's for userspace processes	Pthreads and user-space process emulation
Signal Handling	Real-time signals	POSIX signals
Partition Communication	VxWorks shared memory/queues for DKMs and POSIX based shared memory/queues for RTPs	POSIX shared memory and message queues.
Health Monitoring	Integrated health monitoring with kernel	User-space health monitoring integration

4. IMPLEMENTATION DETAILS

IoT-653 is designed as a lightweight, modular, and extensible framework that emulates the ARINC-653 standard, offering robust time and space partitioning, partition communication, and fault isolation for resource-constrained IoT and embedded systems. The IoT-653 architecture centers around the APEX class, the primary interface for managing partitions, processes, and communication channels. The library is organized into distinct core modules: partition management, process management, partition communication, fault handling, health monitoring, and time management. Each module encapsulates specific functionalities, promoting modularity, maintainability, and scalability, that ensures the framework can grow with system needs.

To support the ARINC-653 scheduling model, the IoT-653 framework emulates a two-level partition and process scheduling framework. Emulating ARINC-653 requires leveraging the underlying operating system's capabilities to mimic both levels of scheduling. In Linux, this is achieved using real-time scheduling policies (SCHED_FIFO, SCHED_RR). SCHED_FIFO provides priority-based preemption for tasks within a partition. VxWorks, being inherently realtime, naturally supports similar constructs. VxWorks Real-Time Processes (RTPs) can emulate partitions with isolated address spaces, and its priority-based task scheduler directly aligns with ARINC-653's intra-partition scheduling requirements. Additionally, VxWorks supports cyclic scheduling through customized system clock hooks, which can allocate deterministic time slices to tasks.

For IoT-653, the APEX System Manager manages the system's resources and provides for the temporal execution between partitions. The manager assigns partitions a specific time slot for execution, preventing interference between partitions. Each partition is configured with a rate of execution (period) and time capacity within each period that defines how often and for how long each partition will execute. The System Manager is configured with a fixed cyclic schedule that repeats after a certain period, known as the hyperperiod. The schedule is pre-determined and

ensures that each partition gets its allocated time slots in a predictable manner. Each partition is also assigned a scheduling window, defined by an offset from the start of the hyperperiod and duration. Only one partition can be executed within its designated window, ensuring no overlap. The System Manager is also responsible for verifying that the schedule is valid before the system can be initialized (i.e., ensuring all windows fit within the hyperperiod).

The APEX Partition Scheduler is charged with scheduling tasks within a partition. The scheduler uses a priority-based pre-emptive scheduling algorithm where processes are scheduled based on their rate (period) and priority. The scheduler ensures that all processes complete their execution within their specified deadline, so if a process fails to meet its deadline, the faulty process is blocked from further execution.

4.1. IoT-653 Library Structure and Core Modules

Partition management is a critical component of IoT-653, emulating the ARINC-653 standard's time and space partitioning to ensure isolation, resource control, and predictable execution. The partition management module in the API is implemented through the APEX class and its related functions. Each partition is defined as a singleton class where all methods and class members are static. No instance of this class is ever created or destroyed. While this approach will not offer hardware-level isolation, it can still provide encapsulation to achieve a degree of logical isolation.

4.1.1. Partition Management

The partition's state is defined by the partition class operational mode, which controls how a partition behaves and interacts with other partitions and the system as a whole. The primary partition modes are defined as follows:

1. COLD_START: This mode is used when the partition is initialized from a powered-off state. It involves full initialization of the partition's resources and environment.
2. WARM_START: This mode is used when the partition is reinitialized without a complete power cycle. It involves partial initialization, typically reloading the application while preserving some state information.
3. NORMAL: This is the standard operational mode where the partition performs its intended functions. The partition executes its application code in this mode and interacts with other partitions and the system.
4. IDLE: In this mode, the partition performs no active tasks. It is in a standby state, conserving resources and waiting for a transition to another mode.

The partition mode transitions are controlled by the system's health monitoring and scheduling mechanisms. For example, a partition might transition from **COLD_START** to **NORMAL** after successful initialization, or from **NORMAL** to **IDLE** if it needs to conserve resources. Figure 2 shows the allowable transitions in a partition's operating mode.

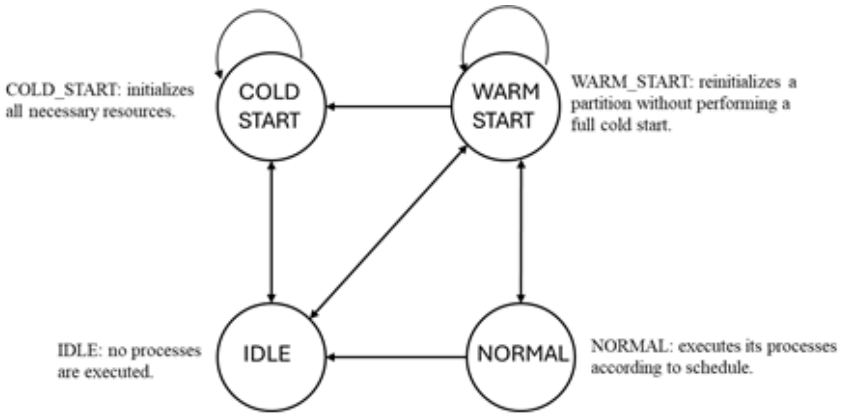


Figure 2: Partition State Transition Modes

4.1.2. Managing Processes

Processes are programming entities contained within a partition. Each process runs concurrently with other processes in the same partition. The process consists of the following components: the executable program, data and stack areas, program counter, stack pointer, and priority deadline. Processes are created using the CREATE_PROCESS service, which defines the process attributes such as priority, stack size, and entry point. Since the service can be called only during a partition's warm or cold start, creation attributes cannot be changed after a partition is initialized. Each process is created only once during the life of the partition.

There are two types of processes: periodic processes and aperiodic processes. A periodic process is a process that is activated at regular times (defined by the PERIOD creation attribute). At activation time, the process becomes eligible for scheduling. When that happens, the state of the process changes to RUNNING or READY if a higher-priority process preempts it. An aperiodic process is the same as a periodic process but without an activation time. Figure 3 illustrates an example of process scheduling. Processes are scheduled according to the POSIX SCHED_FIFO method. In the example, P2 is not complete during the second time period. It is preempted by P3 most of the time since P3 has a higher priority, so it misses its deadline.

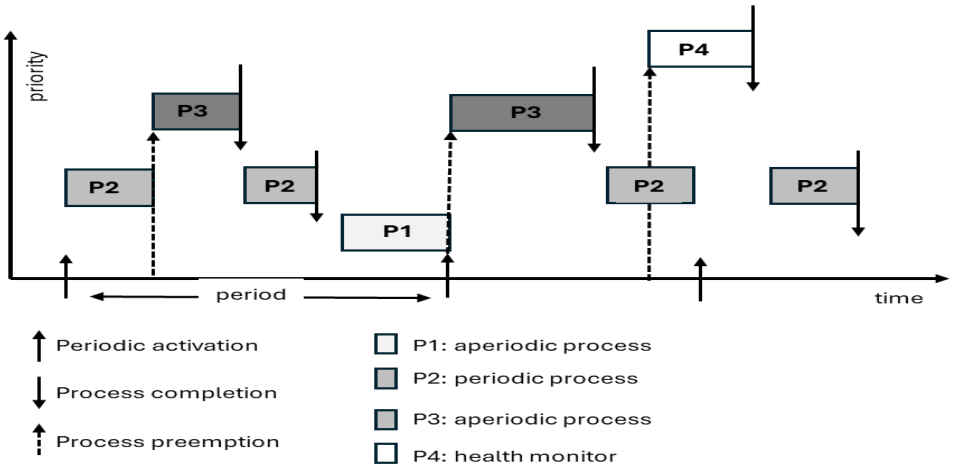


Figure 3: Processes Scheduled with POSIX_SCHED_FIFO Scheduling

Process state transitions defines several states that a process can be in, and the transitions between these states are governed by specific rules and conditions. The primary process states and their transitions are illustrated in Figure 4 below:

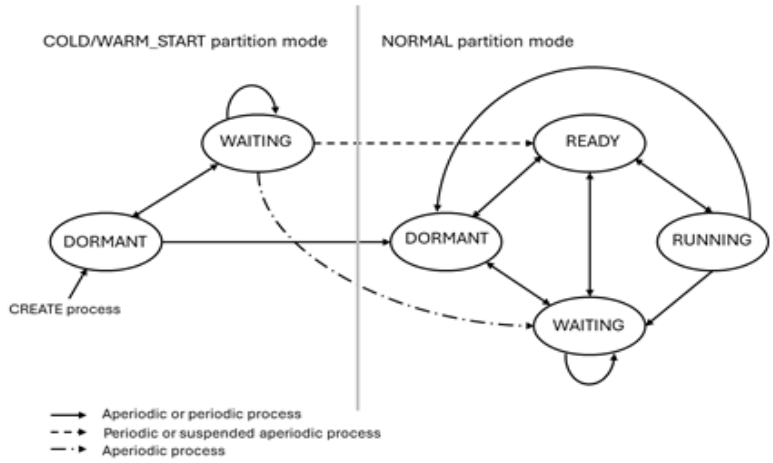


Figure 4: Process State Transition Diagram

The time management services let partitions control their process states. At the end of each processing cycle, a periodic process requests the PERIODIC_WAIT service to get a new deadline. The new deadline is calculated from that process's next periodic release point. For all processes, the TIMED_WAIT service lets the process suspend itself for a minimum amount of elapsed time. After the wait time has elapsed, the process becomes available to be scheduled. The REPLENISH service lets a process postpone its current deadline by the time that has already passed. Each process in a partition can specify the amount of elapsed time (called the time capacity) it is allowed to consume in order to satisfy its processing requirement. This time capacity is used to set a processing deadline time that the IoT framework periodically checks to determine whether the process is satisfactorily completed within the allotted time.

Each process has a fixed time capacity associated with it, representing the response time allotted to satisfy its processing requirements. The deadline determines whether the process is satisfactorily completing its processing within its time capacity. There are three types of deadlines:

- Hard deadlines - If a process fails to meet a hard deadline within the specified time period, IoT-653 raises a DEADLINE_MISSED event.
- Soft deadlines - If a process fails to meet a soft deadline within the specified time period, the failure is recorded, and processing continues. A DEADLINE_MISSED event is also raised that distinguishes between a hard or soft deadline miss.
- No deadline - No action is taken if a process fails to complete processing within the specified time.

4.1.3. Partition Communications

Partition communication includes all communication between processes within the same partition or between two or more partitions in an IoT-653 platform. Inter-partition communication includes all communication between partitions, and intra-partition communication encompasses all communication within a partition. Partitions communicate through messages, ports, and channels, while intra-partition communication processes

communicate within a partition using buffers, blackboards, semaphores, and events. Messages can be sent from one source port to one or more destination ports. Processes read from these destination ports (see Figure 5). Messages can be of fixed or variable lengths. Fixed length means a fixed size for every occurrence of a particular message. A variable-length message can vary in size. The sender specifies the length of time the message will be sent. A message can be sent periodically or on demand (aperiodically). The messaging system operates independently of the content of the messages it transmits

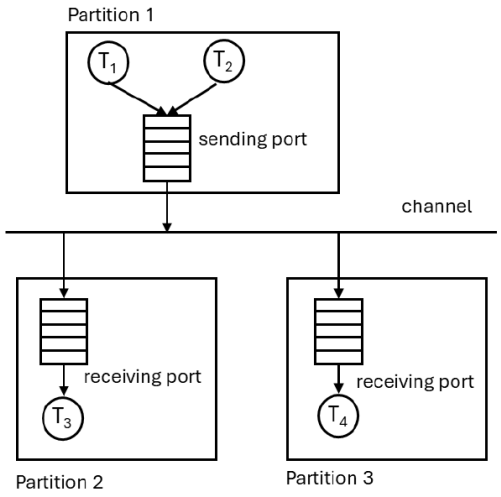


Figure 5: Inter-Partition Communication

A channel defines the logical link between one source port and one or more destination ports and the mode of transfer of the messages from the source to the destination. Two modes of transfer are defined as sampling mode or queuing mode. In sampling mode, messages typically carry similar but updated data. No queuing is performed. A message remains in the source port until it is sent or overwritten. Messages arrive at the destination port or ports in the order they were sent. Each new message overwrites the previous one when it reaches the destination port and remains there until it is overwritten. In queuing mode, each new message instance may contain uniquely different data. Therefore, overwriting previous messages is not allowed during the transfer. Messages are queued in the source port until they are sent, and no message is lost. Messages are stored in the receiver port until a process reads them. In IoT-653, we integrated MicroDDS-XRCE to provide a common communication framework to simplify the integration of inter-partition communication. DDSXRCE topics are then mapped to sampling ports or queuing ports in the IoT-653 framework, where the data written can be published to the corresponding DDS-XRCE topic [11]. For instance, sensor data from one partition can be published on a DDS-XRCE topic in another partition where multiple partitions can subscribe to this topic to receive real-time data, ensuring timely and reliable data distribution across the system [12].

IoT-653 also supports process communication within a partition using the standard APEX objects, which include buffers, blackboards, semaphores, and events. Buffers support a single message type between multiple source and destination processes. Communication is indirect: participating processes address the buffer rather than the opposing processes. Buffers store multiple messages in message queues, and no messages are lost. Blackboards support a single message type between multiple source and destination processes. Like buffer communication, it is indirect: participating processes address the blackboard rather than the opposing processes..

In IoT-653, semaphores are synchronization primitives used to manage access to shared resources among processes within a partition. Semaphores help ensure that only one process can access a critical section of code or a shared resource at a time, preventing race conditions and ensuring data integrity. Semaphores can only be created when a partition is being initialized. Processes can create as many semaphores as are supported by the pre-allocated memory for the partition's semaphores. An event is a synchronization mechanism that signals and coordinates activities between processes within a partition. Events notify processes about specific occurrences, allowing them to respond appropriately.

4.1.4. Partition Health Monitoring

IoT-653 supports process-level health monitoring according to the ARINC-653 standard. When a partition detects an error, it issues the RAISE_APPLICATION_ERROR service with an error code and a fault message. Depending on its nature and scope, an error raised at the process level can propagate to the partition level, where it is processed. A faulty process can continue to run only in the cases of APPLICATION_ERROR or DEADLINE_MISSED. An IoT-653 application creates an error handler process for a partition by issuing the CREATE_ERROR_HANDLER service with the error handler entry point and stack size. The application supplies the error handler code. The error handler is an aperiodic process that runs in the partition window with the highest priority of any process in the partition. It preempts any process regardless of its priority, even if preemption is disabled for the partition. It cannot be accessed by other processes within the partition. Other processes cannot suspend it, stop it, or change its priority. An application developer writes the error handler.

5. IOT SIMULATION TESTBED

The testing and evaluation of IoT software is inherently complex due to the extensive, distributed nature of IoT systems, which must operate in real-world environments. While managing a single node is relatively straightforward, handling hundreds of nodes significantly complicates testing, development, and staging. Physical hardware testing, with its requirement for wireless nodes to be spread over a large area to reduce interference, often necessitates the use of entire buildings or campuses as labs. Setting up and maintaining such environments involves substantial effort, and physical labs are typically constrained by the number and variety of nodes and topologies they can accommodate.

These challenges can be addressed using virtual platforms and simulations of wireless networks and environments [13] [14]. This approach transforms the complex hardware into software simulations that are easier to create, configure, and control. Therefore, to effectively test and analyze our IoT-653 framework, we created an IoT Simulation Testbed based on a virtual platform that simulates an IoT environment's hardware and networking components [15]. For this work, we utilized the virtual platform system provided by Wind River Simics. Simics can simulate the entire IoT network, including all hardware and software components, creating a digital twin of a physical IoT system. This comprehensive simulation, which is as realistic as possible, ensures we can test our framework with confidence, knowing that it closely represents an actual deployment scenario.

5.1. Simulating Nodes and Gateways

The IoT-653 testbed is based on fast, transaction-level virtual platforms for individual gateways and IoT nodes. These virtual platforms simulate the hardware of an embedded target system and run the same binaries as the physical system. Developing the hardware for an embedded target

involves creating detailed virtual representations of physical components and systems. These hardware models are then integrated into the simulation environment, allowing for the interaction and interconnection of various device components.

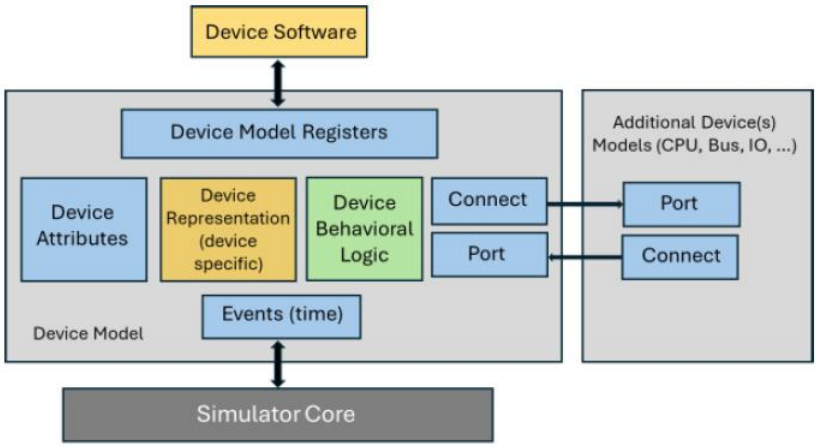
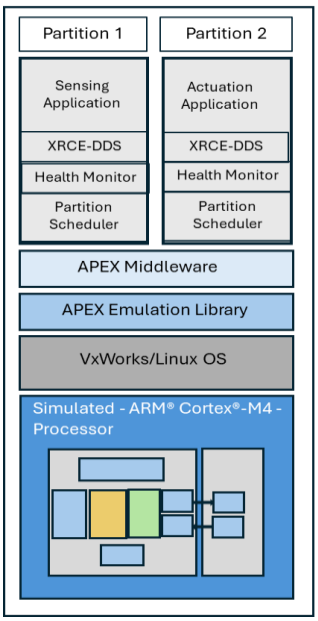


Figure 6: DML Device Model Block Diagram

Using Simics DML (Device Modeling Language), we can model the behavior and characteristics of a hardware component within a virtual platform [16][17]. The primary interfaces of a generic device model is illustrated in Figure 6. The model specifies how the device interacts with other components as well as the software running on the virtual platform. For the lower-end IoT edge devices, we utilized the existing ARM® Cortex®-M4 processor model simulating various sensors and actuators with which the IoT node interacts. The virtual IoT node architecture is depicted in Figure 7(a) and for the more powerful gateway node, we used the ARM® Cortex®-A7 processor model depicted in Figure 7(b).



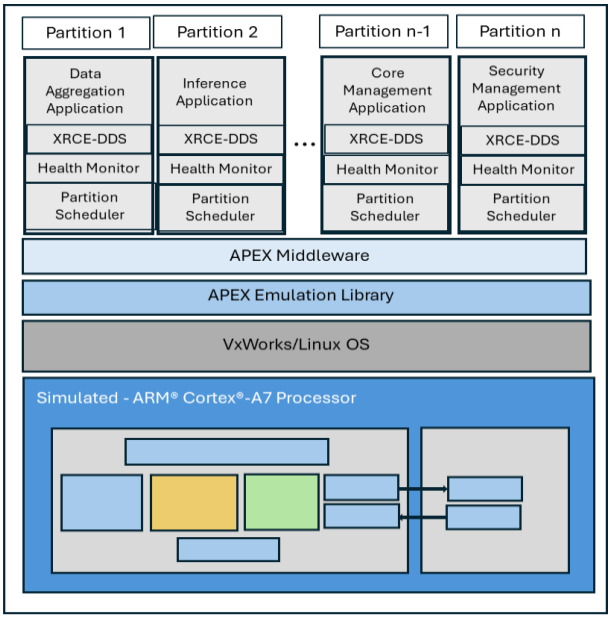


Figure 7(a): Simulated IoT Node Figure 7(b): Simulated Gateway Node

As illustrated in Figure 8, the IoT simulation consists of a large number of individual systems, the hardware of each of which is simulated using virtual platforms. These virtual platforms accurately model relevant aspects of the physical system for the target software, including processor core instruction sets, device registers, RAM, ROM, flash, memory maps, interrupts, timers, and the functionality of other peripheral devices. The architecture and hardware of the virtual platforms are entirely independent of the host system; for example, running code compiled for low-end microcontroller nodes on a powerful Intel Xeon®–based server.

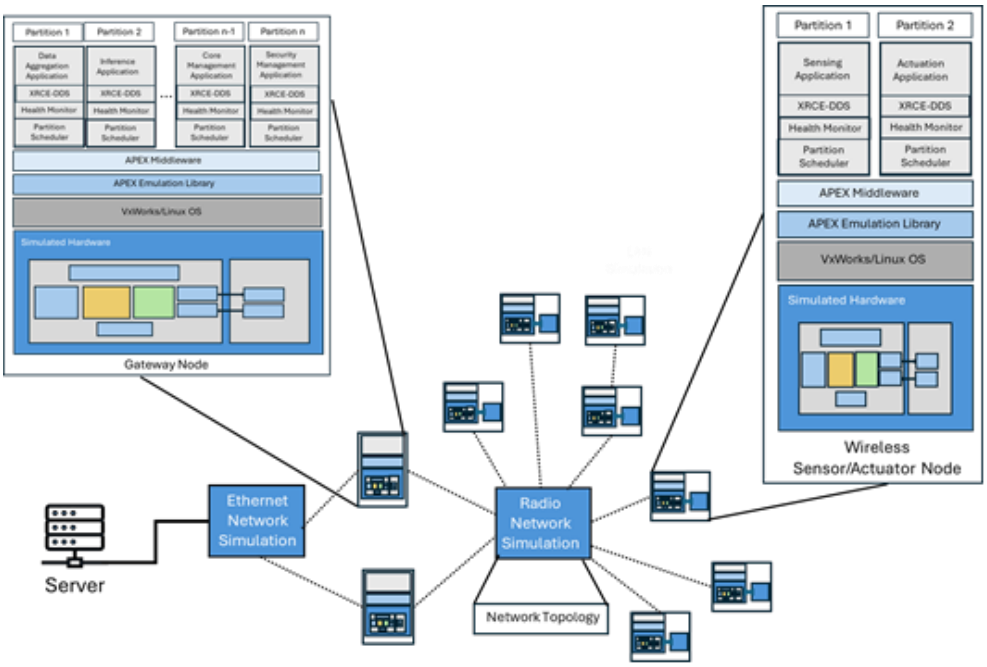


Figure 8: IoT-653 Simulation Testbed - Node and Gateway Simulation

The target software on the virtual platform includes low-level firmware and boot loaders, operating systems, drivers, middleware, and the IoT-653 framework applications. Wind River redesigned VxWorks, so it features two kernels, one for large processors and the other, a microkernel, for smaller, more resource-constrained processors. For the IoT-653 testbed, the edge devices use the micro-kernel, and the more powerful gateway devices use the full kernel. Drivers for I/O devices are part of the setup, and sensors and actuators are represented by simulations of their software-visible interfaces, which include memory-mapped registers, interrupts, and direct memory access (DMA). Multiple heterogeneous targets can be simulated along with the networks connecting them. Our IoT-653 testbed can also connect to the outside world for additional hardware-in-the-loop testing.

5.2. IoT Network Simulation

Accurately modeling and simulating concurrent access is very costly because all simulated nodes must frequently synchronize to check the current state of the shared medium, significantly slowing down the simulation [14]. For the IoT-653 testbed, we used the existing Simics IEEE 802.15.4 network model, which uses a transaction-based network model that abstracts the physical layer into a packet-level message system. This simulation moves entire packets as units and exposes the system software to relevant software-visible effects without overburdening the simulation with excessive synchronization. This approach allows for both parallel and distributed simulation, enhancing scalability.

As shown in Figure 9, all nodes are connected to the same wireless network, enabling any node to send a message to any other node. Messages are transmitted as complete units (packets) over the network without attempting to detect overlapping or simultaneous transmissions from multiple nodes. This approach, commonly used in transaction-level modeling of wired networks, is now applied to wireless networks. The network packet contents include all the fields specified in the IEEE 802.15.4 standard. Radio models manage network addresses, such as rejecting messages with incorrect recipient addresses. The network simulation delivers the message to all nodes capable of receiving it, mimicking a physical radio network.

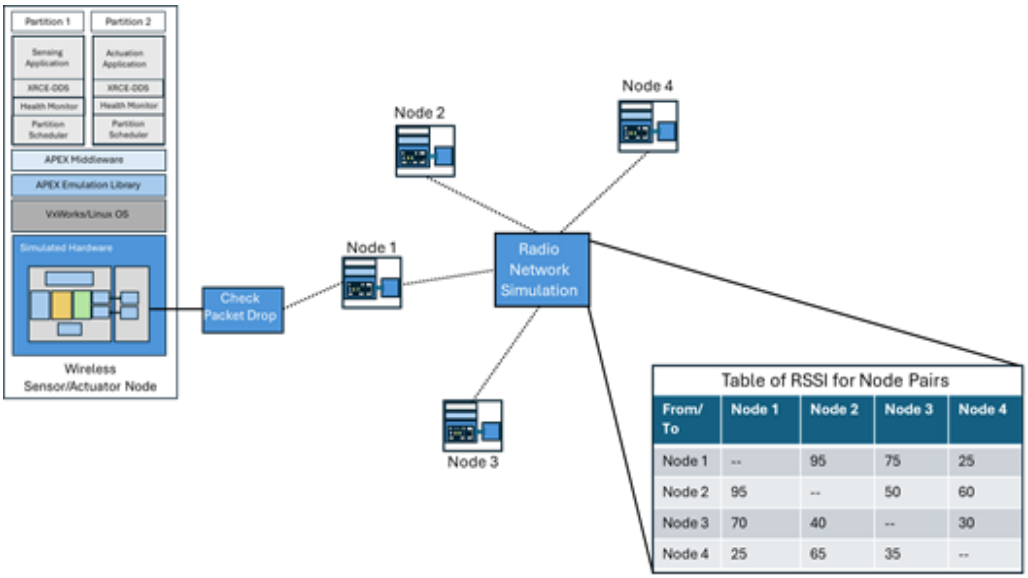


Figure 9: IEEE 802.14.15 IoT Testbed Network Simulation

Using Simics to test, analyze, and validate our framework offers several significant benefits. Simics allows us to simulate the entire IoT network, including all hardware and software components. By creating a simulation testbed, we can test our framework in a realistic environment, closely mimicking an actual deployment scenario. The deterministic nature of Simics allows us to reproduce and debug issues consistently, which is crucial for validating our framework's reliability and real-time performance. Simics supports fault injections, which provides a means to test the robustness and fault tolerance of the IoT nodes. As a result, we can simulate various fault conditions and determine how well the framework handles those faults. Simics can also be integrated with physical hardware components, allowing for hardware-inthe-loop simulations. This flexibility enables us to validate our framework in a mixed environment of simulated and physical devices.

6. IOT-653 TESTING AND EVALUATION

Initial testing was performed in Linux (Ubuntu 22.04) and VxWorks using VxSim; both downloadable kernel modules (DKMs) and real-time processing (RTPs) variants were tested in VxWorks. Unit tests consisted of testing for the various components of our IoT-653 framework, including partition status testing, buffers, blackboards, semaphores, and events, as well as queueing and sampling ports and channels. Integration testing was set up using different scenarios. Single partition testing was used to validate intra-partition communication, and multiple partition testing was used to validate inter-partition communication. Complete system

testing and evaluation were done using the IoT Simulation Testbed discussed in Section 5 to evaluate the effectiveness of our approach for a lightweight, more robust IoT framework.

6.1. IoT-653 Footprint

To evaluate the lightweight approach of IoT-653, we compared the size of our solution to the Partitioned Operating Kernel (POK) Kernel [17], the only other lightweight open-source system that supports partitioning with time and space isolation using ARINC-653 compatible standards. The SLOC counts for IoT-653, and the POK are provided in Table 2 below. Our IoT-653 framework, while exhibiting slightly higher SLOC counts compared to the POK approach, offers significant advantages in terms of portability and overall system functionality. Only minimal hardware is currently implemented with the POK kernel, and the APEX API defined by the ARINC-653 standard is not supported. Therefore, the slight increase in SLOC counts is a minor trade-off for the enhanced portability that IoT-653 provides, making it a more robust and portable solution for various hardware platforms and application environments.

Table 2: SLOC Count Comparisons - IoT-653 vs. POK Kernel

ARINC-653 Component	**IoT-653 SLOC Count**	**POK Kernel SLOC Count**
Blackboard	197	102
Buffer	223	117
Partition Management	286	114
Events	243	269
Health & Management	395	N/A
Process Management	893	293
Queue Port	428	154
Sampling Port	384	136
Partition Scheduling	415	556
Semaphore	245	178
Time Management	245	79
Middleware	1531	759
*Micro-XRCE-DDS	20412	N/A

*****Note: Micro-XRCE-DDS is integrated into the VxWorks kernel, so the SLOC count does not explicitly count against the IoT-653 framework but is considered as part of the overall image load size. Additionally, the POK Kernel is a BSD based ARINC-653 kernel that is about 22K SLOC while the VxWorks microkernel is about 20K SLOC.

6.2. IoT-653 Reliability

To verify the reliability of our IoT-653 framework, we utilized fault injection techniques in Simics. By simulating various fault conditions, we could test our framework's resilience under adverse scenarios. We injected a range of faults, including bit flips, stuck-at faults, and communication errors, into various nodes in our IoT network.

The fault injection process involved creating an error handler code included in the IoT-653 health monitor to manage process-level error codes such as APPLICATION_ERROR or HARDWARE_FAULT. We can inject several faults using Simics' fault injection framework. For example, a bit-flip fault is injected as follows:

1. Define the fault injector

```
simics> @fault_injector = SIM_create_object("fault_injector", "bit_flip")
simics> @fault_injector.location = "adc temp" simics> @fault_injector.address
= 0x40010000 simics> @fault_injector.bit = 3
```

2. Attach the Fault Injector to the Sensor Node

```
simics> @sensor_node = SIM_get_object("sensor_node") simics>
@sensor_node.add_fault_injector(fault_injector)
```

3. Inject the Fault:

```
simics> @fault_injector.inject()
```

By simulating various fault scenarios, we were able to identify potential vulnerabilities and ensure our framework could handle unexpected conditions. This approach allowed us to enhance our IoT solutions' overall resilience and stability, providing greater confidence in their performance in real-world applications.

6.3. IoT-653 Security

We conducted a series of tests on our IoT-653 framework using Simics to simulate a mild Denial of Service (DoS) attack. We could evaluate the framework's security under stress by generating controlled traffic surges. Specifically, we configured Simics to simulate compromised nodes sending excessive requests to our IoT-653 framework, mimicking a realworld DoS scenario. A sample DoS attack on a sensor node using Simics is provided below:

1. Define the DoS Attack Parameters

```
simics> @dos_attack = SIM_create_object("dos_attack", "tcp_syn_flood")
simics>@dos_attack.target = "sensor_node 1" simics> @dos_attack.rate =
1000  # packets per second simics> @dos_attack.duration = 60  # duration
in seconds
```

2. Configure the Attacker Node(s)

```
simics> @attacker1 = SIM_get_object("attacker_node1") simics>
@attacker1.add_attack(dos_attack)
```

3. Initiate the DoS Attack `simics> @dos_attack.start()`

To evaluate the effectiveness of detecting a DoS attack, we utilized the health monitoring process error handling facility to identify the attack by recognizing that the sensor node was missing its deadlines. In the DEADLINE_MIISED event, the error handling procedure was configured to isolate the affected sensor and shut down the partition. Note that other less intrusive approaches could have been used, such as adjusting scheduling priorities, throttling incoming traffic, or implementing rate limiting to reduce the load on the sensor node. For this testing scenario, we generated 24 virtual sensor nodes and one gateway node in the testbed, similar to the architecture depicted in Figures 8 & 9. We created a mild DoS attack starting from 1 to 10 random sensor nodes in the simulated IoT network. The test scenario configured the partitions with and without health monitoring to detect the DoS attack. As depicted in Figure 10, the nodes that had health monitoring enabled in their partition performed far better than when monitoring was not enabled. This improvement is because as the health monitor starts to detect missed deadlines, it shuts down the partition, which, in effect, thwarts the DoS attack.

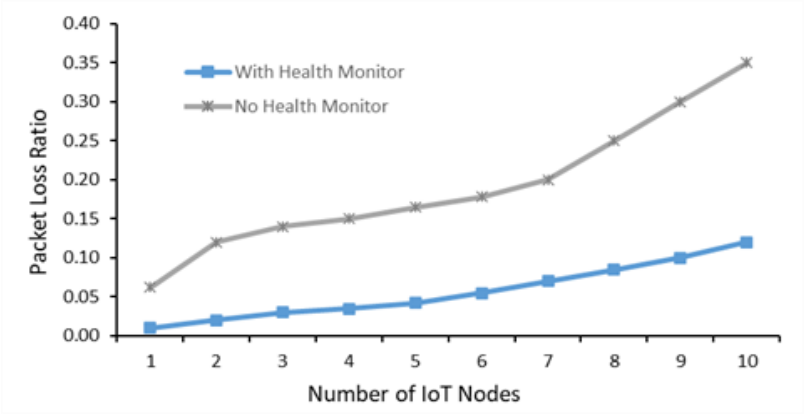


Figure 10: Packet Loss Ratio During DOS Attack

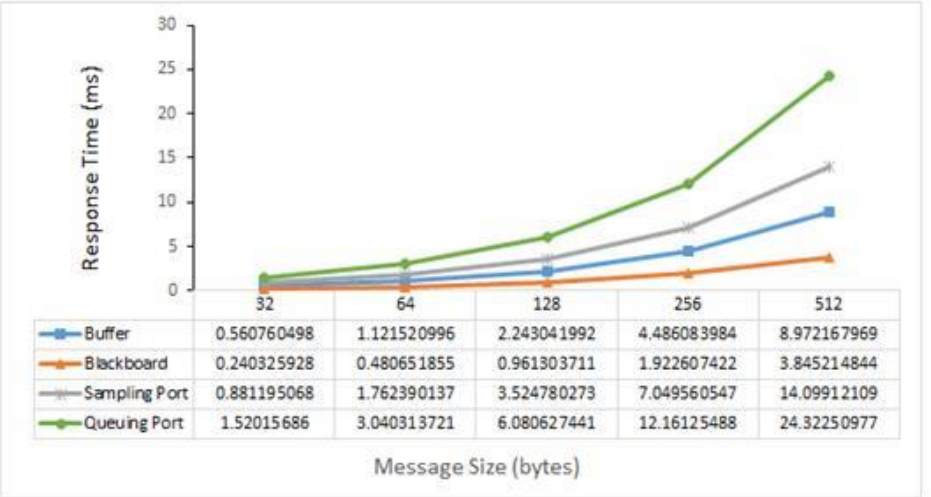
6.4. IoT-653 Real-Time Performance

Real-time deterministic behavior was analyzed by measuring the response time and jitter for the APEX interfaces for partition and process management. Note: that Simics is a simulator that runs on a host instead of specific target hardware therefore we expect even better deterministic behavior when running on a physical target processor. Table 3 shows the measurement results with the APEX interfaces for partition and process management.

Table 3 : APEX Interface Response Time – VxWorks (Simics)

APEX Interfaces	**Avg (µs)**	**Avg Jitter (µs)**
GET_PARTITION_STATUS	18.85	1.92
GET_PROCESS_STATUS	20.47	1.31
GET_MY_ID	21.98	1.53
GET_PROCESS_ID	12.77	2.80
START_PROCESS	28.76	4.92
STOP_PROCESS	164.34	15.21
PROCESS_CREATE	76.18	17.11

This section tests four inter-partition and intra-partition communication methods using the standard APEX API to measure partition communication delays. The results, shown in Figure 11, indicate that sampling mode and blackboard have faster transmission speeds because new data can overwrite current data without waiting for scheduling.



	32	64	128	256	512
Buffer	0.560760498	1.121520996	2.243041992	4.486083984	8.972167969
Blackboard	0.240325928	0.480651855	0.961303711	1.922607422	3.845214844
Sampling Port	0.881195068	1.762390137	3.524780273	7.049560547	14.09912109
Queuing Port	1.52015686	3.040313721	6.080627441	12.16125488	24.32250977

Figure 11: Response Times for Partition Communications

In contrast, queueing mode and buffer use a queue mechanism and have slightly slower transmission speeds. Overall, all methods demonstrate good functional characteristics. This realtime performance testing demonstrates that our IoT framework demonstrates real-time performance by ensuring efficient and predictable communication between partitions.

7. CONCLUSIONS AND FUTURE WORK

We developed a lightweight solution that enhances security, reliability, and real-time performance for resource-constrained IoT devices. The IoT-653 framework leverages a lightweight partitioning approach that emulates key features of the ARINC 653 standard, known for its robust partitioning and scheduling capabilities in avionics systems. Our implementation was tested on Linux and VxWorks platforms to ensure compatibility and real-time performance. We also created an IoT simulation testbed using Simics that provided a digital twin of a physical network, allowing us to simulate and evaluate the system's behavior in a controlled environment. The testing results for enhanced reliability, security, and real-time performance demonstrated that our approach is viable for resource-constrained IoT devices. By rigorously evaluating the system on both Linux and VxWorks platforms and utilizing an IoT simulation testbed, we confirmed that our lightweight partitioning solution not only meets but exceeds the performance of existing solutions like the Portable Operating System (POK). The results showed significant improvements in reliability and security, validating our approach as a robust and effective method for enhancing the performance of IoT devices in real-world applications.

For future work, we plan to incorporate hardware-in-the-loop testing, for various hardware platforms such as the STM32 B-L475E-IOT01A IoT node and the QorIQ LS1028A gateway node. This level of testing will allow us to evaluate the solution's performance in real-world scenarios. Additionally, we aim to conduct more comprehensive testing to identify potential security vulnerabilities by simulating various attack vectors. Additionally, we will implement more error testing procedures to evaluate the system's reliability under different failure conditions, which could help us identify any potential weaknesses while improving the overall robustness of our solution.

ACKNOWLEDGEMENTS

This work was funded in part by a grant from the Wind River Corporation grant #US1133982.

REFERENCES

[1] Number of Internet of Things (IoT) connections worldwide from 2022 to 2023, with forecasts from 2024 to 2033. https://www.statista.com/statistics/1183457/iot-connected-devicesworldwide
[2] Canavese, D.; Mannella, L.; Regano, L.; Basile, C. Security at the Edge for Resource-Limited IoT Devices. Sensors 2024, 24, 590. https://doi.org/10.3390/s24020590
[3] S. H. VanderLeest, “Arinc 653 hypervisor,” in 29th Digital Avionics
[4] Avionics Application Software Standard Interface Part 1-2, ARINC Specification 653P1-2, 2005.
[5] G. Peach, R. Pan, Z. Wu, G. Parmer, C. Haster and L. Cherkasova, "eWASM: Practical Software Fault Isolation for Reliable Embedded Devices," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 39, no. 11, pp. 3492-3505, Nov. 2020.
[6] Z. B. Aweke and T. Austin, "uSFI: Ultra-lightweight software fault isolation for IoT-class devices," 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE), Dresden, Germany, 2018, pp. 1015-1020.
[7] Hoornaert, D., Ghaemi, G., Bastoni, A. et al. Mcti: mixed-criticality task-based isolation. RealTime Syst 60, 328–365 (2024).
[8] Kim, H.; Larasati, H.T.; Park, J.; Kim, H.; Kwon, D. DEMIX: Domain-Enforced Memory Isolation for Embedded System. Sensors 2023, 23, 3568. https://doi.org/10.3390/s23073568
[9] Luca Abeni, Giuseppe Lipari, and Juri Lelli. 2015. Constant bandwidth server revisited. SIGBED Rev. 11, 4 (December 2014), 19–24. https://doi.org/10.1145/2724942.2724945.
[10] G. Buttazzo and E. Bini, "Optimal Dimensioning of a Constant Bandwidth Server," 2006 27th IEEE International Real-Time Systems Symposium (RTSS'06), Rio de Janeiro, Brazil, 2006, pp. 169-177
[11] ePROSIMA Distributed Middleware: https://www.eprosima.com
[12] Pérez, Héctor and J. Javier Gutiérrez. “Data-centric distribution technology in Arinc-653 systems.” REACTION (2014).
[13] B. Gogan, “Chromecast Testing,” Google Test Automation Conference (GTAC), Boston, MA: November 10–11, 2015.
[14] Daniel Aarno and Jakob Engblom, “Software and System Development using Virtual Platforms—Full-System Simulation with Wind River Simics,” Morgan Kaufmann Publishers, 2014.
[15] Laura Belli, Simone Cirani, Luca Davoli, Andrea Gorrieri, Mirko Mancin, Marco Picone, and Gianluigi Ferrari, “Design and Deployment of an IoT Application-Oriented Testbed,” IEEE Computer, September 2015.
[16] Jakob Engblom, David Kågedal, Andreas Moestedt, and Johan Runeson, “Developing Embedded Networked Products using the Simics Full-System Simulator,” Proceedings of the IEEE 16th International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC), Berlin, Germany: September 11–14, 2005.
[17] Ortiz, A., Ortega, J., Diaz, A. F., & Prieto, A. (2007). Modeling Network Behavior By FullSystem Simulation. J. Software., 2(2), 11-18.
[18] Delange, J., & Lec, L. (2011). POK, an ARINC653-compliant operating system released under the BSD licence.