Interrupting Real-Time IoT Tasks: How Bad Can It Be to Connect Your Critical Embedded System to the Internet?

Ilja Behnke*, Lukas Pirl‡, Lauritz Thamsen*, Robert Danicki*, Andreas Polze‡, and Odej Kao*
* Technische Universität Berlin, Germany, i.behnke@tu-berlin.de
‡ Hasso Plattner Institute, Germany, {first.last}@hpi.de

***Abstract*—Embedded systems have been used to control physical environments for decades. Usually, such use cases require low latencies between commands and actions as well as a high predictability of the expected worst-case delay. To achieve this on small, low-powered microcontrollers, Real-Time Operating Systems (RTOSs) are used to manage the different tasks on these machines as deterministically as possible. However, with the advent of the Internet of Things (IoT) in industrial applications, the same embedded systems are now equipped with networking capabilities, possibly endangering critical real-time systems through an open gate to interrupts.**

This paper presents our initial study of the impact network connections can have on real-time embedded systems. Specifically, we look at three aspects: The impact of network-generated interrupts, the overhead of the related networking tasks, and the feasibility of sharing computing resources between networking and real-time tasks. We conducted experiments on two setups: One treating NICs and drivers as black boxes and one simulating network interrupts on the machines. The preliminary results show that a critical task performance loss of up to 6.67% per received packet per second could be induced where lateness impacts of 1% per packet per second can be attributed exclusively to ISR-generated delays.

***Index Terms*—Internet of Things, Real-time, Interrupts, Cyber Physical Systems, Embedded Systems**

I. Introduction

Embedded devices that control machines in the physical world have been part of industrial processes as well as home and automotive appliances for decades [1]–[3]. In contrast to general-purpose computing, these devices need to fulfill timing constraints. To this end, Real-Time Operating Systems (RTOSs) are used, which are lightweight and make guarantees towards the timing predictability of tasks [4]. Usually, a preemptive task scheduler allows to configure different priorities for different concurrently active tasks, so that the most time-critical tasks always take precedence over less critical ones.

Interrupt Requests (IRQs) are generated by the hardware and inevitable for systems to function. At the same time, they introduce a level of unpredictability to the process flow. Since the corresponding Interrupt Service Routines (ISRs) are handled by the processor, the scheduler of an Operating System (OS) has no control over their execution. However, by keeping the execution times of ISRs minimal and considering worst-case scenarios during the development, most traditional embedded systems can handle IRQs without missing deadlines. Yet, in the past, the environment controlled by embedded systems tended to be self-contained, the number of environmentally-triggered IRQs was typically small, and their impact, therefore, predictable. With the advent of the Internet of Things (IoT) in industrial applications this premise has changed. IoT microcontrollers come with built-in network chips and are increasingly often network-connected for the sake of remote control, monitoring, and maintenance [5]. IoT networks are, however, open by design and thus less policed [6]. Especially for critical real-time tasks on networked embedded microcontrollers, this is a threat: The embedded systems have to handle the additional resource consumption of the non-critical networking tasks and the necessary Network Interface Controller (NIC) introduces a new source of unpredictability as incoming packets trigger IRQs that disturb the flow of scheduled tasks [7]. This might lead to a critical load of interrupts and triggered network tasks in the RTOS, invalidating real-time guarantees and thereby lowering the system's dependability.

This paper analyzes the impact of network loads on critical real-time tasks running on state-of-the-art microcontrollers with modern RTOSs used in the IoT. In an initial study, we evaluate timing measurements of critical tasks on microcontrollers running vendor-supplied RTOSs, network drivers, and network stack tasks under different network-triggered IRQ loads. To expose any existing mitigation in the hardware and closed source drivers, a pseudo network driver is designed to serve as a second IRQ source. Measurements are taken and compared between real and pseudo network packet processing. Building on our methodology, we perform the following contributions:

- Measurement of ISR-induced delay to real-time tasks.
- Evaluation of overhead induced by networking tasks under different network loads.
- Preliminary analysis of the feasibility of IoT programming frameworks and IP networking in real-time scenarios.

Outline. The remainder of the paper is structured as follows. Section II presents the problem statement. Section III introduces our evaluation methodology. Section IV presents our empirical results with two different setups. Section V discusses the results. Section VI describes the related work, while Section VII concludes the paper.

II. Problem Statement

As motivated, the inclusion of network controllers to embedded real-time systems introduces an unpredictable source of interrupts. The goal of this work is the analysis of the impact of these interrupts in IoT environments. This includes the execution of ISRs, network drivers, and network stack tasks, as well as the robustness of the entire system under high network loads. This section provides the scope and assumptions of our work.

A. IoT Environments

Available IoT devices are microcontrollers possessing the means to wirelessly connect to networks and handle a multitude of different network protocols comparable to the network stack implementations in general computing. At the same time, these systems are designed to be deployed in untrusted environments, more or less directly connected to the internet [8]. Packet floods generated by faults and security breaches furthermore lay open potentially critical systems that are deployed in secluded networks. Due to these circumstances, we consider networked embedded systems of increased vulnerability while controlling critical systems with real-time constraints [9].

The high popularity of the IoT led to the mass production of IP capable devices that are cheaply available. To make the somewhat complex programming for embedded systems more available in the IoT context, devices come with programming frameworks masking low-level software like network stacks and drivers. While the investigated modules can technically also be programmed without their respective frameworks, the immense workload of the implementation and inclusion of the network driver and stack tasks makes this unfeasible in most cases. An incorporation of the built-in Wi-Fi chips into fully manageable and transparent real-time systems is hindered by the unavailability of driver source code.

B. Design and Development for IoT MCUs

During the design and development of real-time systems, the developer is responsible for making sure that any deadlines are met in the worst-case scenario. In case of externally caused interrupts, it is therefore necessary for the developer to incorporate the maximum frequency of interrupts and their impact into the design of the system. Network-generated interrupts however leave the developer only with few choices like turning off interrupts or reserving a physical core for networking tasks [10]. While both actions might solve the problem of interrupt floods over the network, the loss of a core (if available) and access to the embedded system from outside might not be feasible.

Additionally, the limited access to low-level functions leads to a loss of control over the real-time system, which should generally be designed holistically.

C. Assumptions

Concluding, we make the following assumptions for our analysis.

- *A1 - Unreliable networks*: IoT devices are commonly deployed in untrustworthy networks and/or might be subject to network faults.
- *A2 - WiFi*: While the issue of unpredictable interrupts is the same across network interfaces, driver availability and specific processing costs differ. Wi-Fi connections belong to the most commonly used interfaces, since no modifications to available hardware has to be made and the technology is mature.
- *A3 - Shared Resources*: Networking tasks, drivers, and application tasks might run on the same MCU core.
- *A4 - Device-specific frameworks*: To implement real-time systems with Wi-Fi capabilities on IoT devices, device-specific frameworks are used.

The design of our experiment setup is deviated from these assumptions.

III. Methodology

A certain impact of networking tasks and interrupts on Microcontroller Unit (MCU) utilization and timing predictability in real-time embedded systems seems inherent. With the experimental methodology and setup we aim to analyze this impact both qualitatively and quantitatively on two current Commercial Off-The-Shelf (COTS) IoT devices.

A. Experiment Design

For the quantitative analysis we designed two sets of experiments observing timing metrics of a periodic task. The experiments are run under different network loads with two regarded interrupt setups (cf. Section III-D). The experiment permutations were additionally adapted to take into account the differences between priorities of preexisting network tasks.

Observed Metrics: We defined two sets of experiments observing different metrics under changing network loads.

1) Lateness: The critical task we observe is periodically called by a timer with period p and a deadline d. We define the lateness l as the time the task takes longer to finish than its deadline allows, hence $l = t_{end} - d$ where t_{end} is the time at which one process cycle of the task has finished as depicted in Figure 1. Once we reach a load at which the task misses its deadline, the lateness will start to accumulate over iterations. We present the accumulated lateness per second.

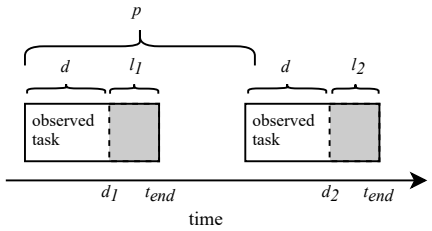


Fig. 1: Lateness measurements.

2) Relative MPU utilization: In the second set of experiments, the observed task runs in a closed loop for a duration of Δt in an interval i. By measuring the number of cycles the critical task finishes in one Δt under different network loads we can make out the resource utilization of networking operations depending on the number of received packets.

B. Test Environment

The experiments were performed on two widely used IoT development boards in similar performance and price ranges. One equipped with a dual-core ESP32 chip at 160 MHz, the other with a Particle Photon (P0) single-core ARM Cortex M3 at 120 MHz. While both devices have ports of the FreeRTOS operating system, the vendor-provided programming frameworks differ significantly in terms of programmability and port specific implementations. Both devices are programmed using their respective frameworks. While the frameworks themselves and operating systems are open source, some low-level software such as Wi-Fi drivers are only available as binary objects. While the differences between frameworks and used processing chips limit the degree of direct comparability, they allow us to evaluate the possibilities of development inside the frameworks' constraints. This way, all results are realistic for the systems tested under their established workflows.

C. Task Priorities

To keep the connections alive, driver and networking tasks have to be kept running on the devices. These are called by NIC-triggered ISRs but can be preempted by a higher prioritized task on the same core.To evaluate the different shares of delays caused by ISRs and networking tasks, experiments are repeated with observed task priorities chosen above, equal to, and below the driver. The default networking task and driver priorities are specified by the frameworks as depicted in Table I. Wi-Fi driver priorities cannot be changed.

TABLE I: Networking task priorities module in firmwares

prio	ESP32	P0	prio
24	-	Wi-Fi driver	*9*
23	Wi-Fi driver	network (high)	*8*
22-19	-	network (low)	*7*
18	network	-	*6-1*

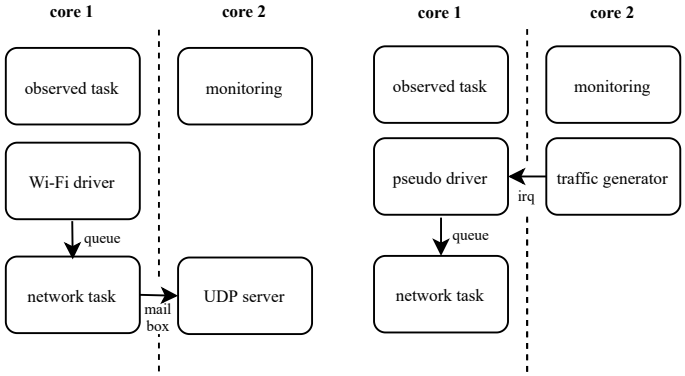
D. Interrupt Workload

To analyze the impact on lateness and computing resource utilization, the devices under test are put under different network loads.

Generation: One of the main difficulties when investigating the process flow a received packet triggers is the unavailability of large parts of the driver source code. Quantitatively analyzing the number of times an ISR is effectively called is therefore difficult to realize. To be able to compare the relative impact of ISRs and low-level packet processing, two interrupt setups are run on the devices.

1) Real Network Packets: In the first setup, network packets are sent to the devices over a Wi-Fi connection and are handled by the framework-supplied driver and networking tasks (Figure 2a). We use the second core of the ESP32 to run a minimally configured UDP server to measure its impact. As the P0 does not have a dual-core processor, no UDP server was run and interrupts were generated externally.

2) Simulated Packets: Secondly, we implemented an analogous task flow to perform Wi-Fi driver independent experiments (Figure 2b). A networking task simulation largely performs the same actions a packet received over a network triggers: Upon registering the interrupt which in this case is triggered via an input pin, a short ISR is called that preempts the currently running process to copy a packet descriptor to a FreeRTOS queue. A pseudo driver task waits for packets in this queue and unblocks when it is not empty to process the entries.



(a) Wi-Fi driver (b) network simulator

Fig. 2: Experiment setups on ESP32.

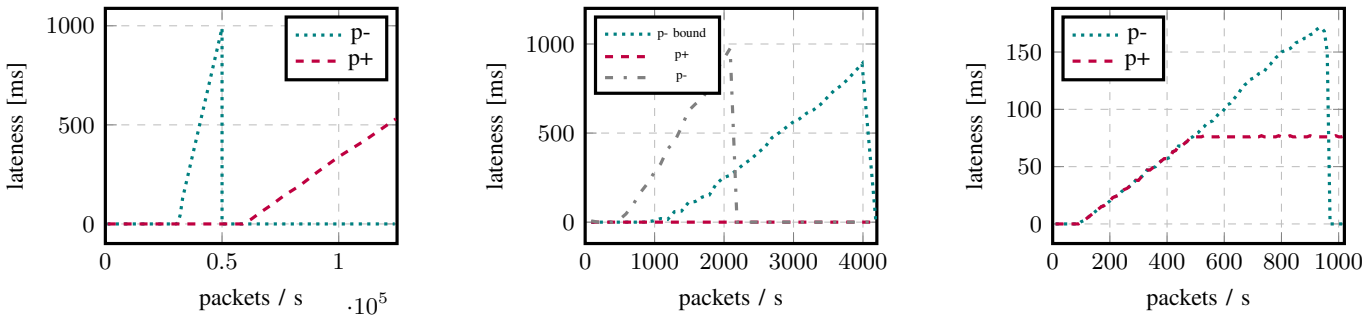
Traffic Loads: Changes in network load directly affect the number of ISR, driver, and network stack calls in the operating system. Depending on the system and interrupt type, network loads were increased in steps of 10, 100, or 1,000 packets per second and held to take measurements. Additionally, experiments were conducted with traffic bursts of 120,000 packets per second for one second.

IV. Experimental Results

This section presents our preliminary empirical results of how interrupt loads impact the lateness of critical tasks and utilization of CPUs.

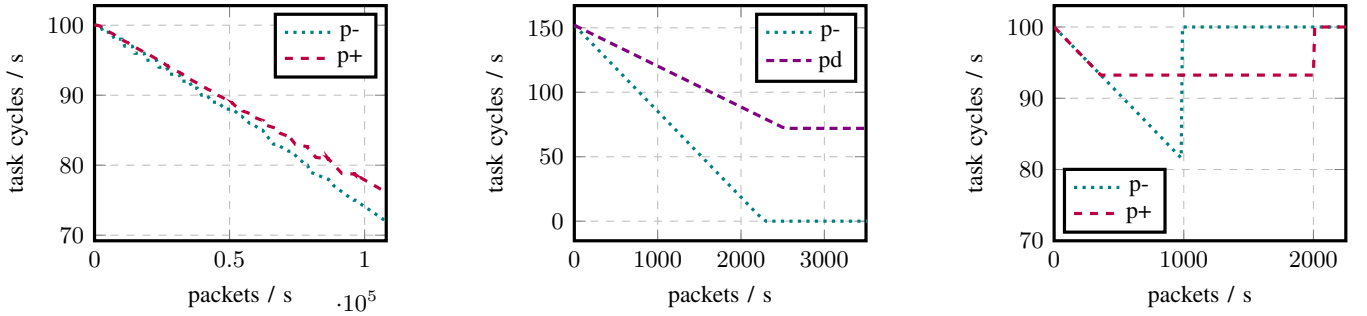
A. Lateness Experiments

ESP32: The first group of experiments was conducted to measure the lateness of the observed task under rising packet loads. Figures 3a and 3b contain the lateness results under simulated and real network traffics on the ESP32. Both show a linear increase in lateness with rising packet load once lateness occurs for priorities chosen below the driver priory with real IP packet impact reaching 50% lateness increase per packet per second. The differences between priorities correspond to



(a) ESP32, simulated network traffic (b) ESP32, real IP packets (c) P0, real IP packets

Fig. 3: Lateness experiment results with observed task priorities under networking tasks (p-) and critical (p+).



(a) P0, simulated network packets (b) ESP32, real IP packets, unbound port (c) P0, real IP packets

Fig. 4: Task cycle counter experiment results with observed task priorities under networking tasks (p-), critical (p+), and equal with wi-fi driver (pd).

the preemption of the fixed networking tasks once the MCUs are fully utilized. When the observed task has a lower priority than the networking tasks, it starves once too many packets arrive. When the observed task has a critical priority, it in turn starves the networking tasks. As can be seen this results in no impact, suggesting that no ISRs are triggered by the NIC in this case.

Figure 3b also shows that the impact of incoming packets is a lot higher when addressed to a port that is not listened on. This might be due to the partial deactivation of networking tasks when UDP buffers in the network stack are full.

Lateness results under traffic bursts do not differ from continuous loads and no other influence to the systems could be seen. Further burst results are therefor omitted from this work as this was true for all burst experiments.

P0: Figure 3c contains the analogous results of the experiments on the P0. The results of the simulated approach are very similar to the ESP32 equivalent, with the difference in slope explainable by platform specific pseudo driver implementations. The results from the experiments using real network packets show an impact slope of 2.2%. The results also show issue of the network driver residing on the highest priority level. Once the packet load is high enough that the network driver needs half of the computing resources for itself, the operating system's scheduler distributes the processing resources equally between the driver and the observed (critical) task. In contrast to ESP32 task, the observed task here does not starve when of lower priority. Before this can happen, the Wi-Fi task crashes at 980 packets per second.

B. Utilization Experiments

Figure 4 contains the parallel results for real network packets and software. Task priorities of the presented results were chosen equal to and lower than the Wi-Fi driver. Task performance decreases by 2.15% and 3.8% respectively. When receiving packets on an unused port the impact is again higher with performance decreases of 3.17% and 6.67% respectively.

Figure 4a contains the results for the network driver simulation on the P0, which are comparable to the lateness results. With real network packets the system crashes at 980 packets per second when giving the observed task a lower priority than the Wi-Fi driver and at 2,000 packets per second with the same priority. Task performance decreases by 2% per packet per second until equal resource utilization with the driver (same priority) or throughout gradual extrusion by it (lower priority).

V. Discussion

This section discusses the findings from the experiments and the feasibility of IP networking on MCUs.

A. Insights from the Experiment Results

The evaluation results show that the performance and real-timeness, is directly dependent on incoming IP network load.

A high overhead generated by the receiving of packets can be seen in continuous floods as well as short transmission bursts. The Wi-Fi driver and network stack utilize any compute resources they need to handle packets unless preempted. Yet, the results also show that NIC-triggered ISRs have no impact on the observed systems. This observation is made when incoming packets are not handled due to the driver being preempted by a task of critical priority.

To mitigate the observed breaking of real-time guarantees on the tested devices, developers still have some options. The ESP IDF provides one priority level above the Wi-Fi task's priority. Running a critical task here will preempt the driver. For critical code sections it is also possible to disable all or only the Wi-Fi interrupt. This is the only option under Particle's DeviceOS.

B. Feasibility of IP Networking

The main take away is that it is necessary to fit all mission-critical operations into a critical task that is independent of any signals received over the network. Hence, performing command and control operations over the network cannot be an option for real-time systems. This however invalidates most IoT use-cases.

Network driver tasks, which are responsible for a large share of the impact on timing predictability and hence lateness, are very highly prioritized in the tested systems' frameworks. This highly limits the margin critical tasks have. While short and independent tasks and code blocks can be executed in a safe manner, this is also where it ends.

Network driver tasks are currently given a priority level in RTOSs that is not suitable for critical real-time systems. Using network connectivity only for monitoring timing independent tasks might still be an option when one is ready to provide the resources and prioritize the necessary tasks appropriately.

C. Networking Software Robustness

Current IoT device programming frameworks seem to be designed for certain connectivity guarantees rather than real-timeness of critical applications. However, this also does not hold for the evaluated systems as can be seen from the results. Under the conditions of the test setup, even low traffic loads lead to Wi-Fi driver crashes and, depending on the framework configuration, to system halts. This could be reliably reproduced on the P0 (cf. Figure 4c).

D. Summary

Following points can be taken from our work:

- If no explicit care is taken, networking has a huge impact on the real-timeness of embedded systems, rendering real-time guarantees invalid.
- The empirically observed problems are largely introduced by IoT programming frameworks rather than the general architecture.
- Transmitting command and control signals over an IP network is not feasible for tasks that need hard real-time guarantees.
- The inflexibility of current network stack implementations and the high networking overhead suggest the reservation of a separate core for networking software. This might not be worth the cost since the previous point holds true for dual-core modules.
- Limited access to network related tasks prevent a holistic real-time system design and cooperative scheduling between them and programmed tasks.

VI. Related Work

Addressing the unwanted impact of interrupts is being researched since decades. So, to understand today's common interrupt handling mechanisms, it is helpful to understand the evolution of those mechanisms. The first part of this section will hence briefly outline implementations that pioneered concepts which are relevant up until today. Thereafter, we present scientific works which address the impact of IRQs on embedded real-time systems and applications.

A. Basic Interrupt Handling Mechanisms

The *LynxOS* RTOS has introduced the concept of running re-entrant network protocol stacks in the priority spectrum of a receiver's process. By de-multiplexing network traffic at the NIC level, low latency for processing real-time network traffic can be guaranteed. LynxOS claims to be the first system built on an architecture that separates ISRs and Interrupt Service Tasks (ISTs) (i.e., light weight kernel tasks) [11].

The *SUN* (now *Oracle*) *Solaris* OS [12] is another prominent example for implementing interrupt handling within the priority spectrum of regular process and thread scheduling. Solaris distinguishes real-time, system, and time-sharing scheduling classes. With these priority spectrums, Solaris was able to effectively isolate real-time processing from networking.

Windows Embedded Compact (*Windows CE*) [13], is an OS family developed for handheld consumer electronics (CE) devices. Interrupt handling is divided among the kernel and a process running all device drivers. By running network handlers with a priority lower than the one used for real-time tasks, Windows CE effectively is able to implement isolation and call admission for incoming network activities.

These OSs served millions of industrial applications in practice and the influences of their concepts can still be found today. Nevertheless, applications, their surrounding environments, and their underlying hardware have changed, where even entry-level hardware has complex performance features.All together, these developments complicate the timing predictability and existing knowledge needs to be confirmed or refined with up-to-date numbers.

B. Advanced Interrupt Handling

Several works have identified the duality in priority spaces as still being a challenge for RTOSs and applications.

A class of approaches tries to mitigate the unwanted effects with additional hardware. This hardware usually intercepts the IRQs between their source and the CPU. Gomes et al. propose

to extend the interrupt controller with a task-aware priority controller [14]. This controller compares the priority of an incoming IRQ with the priority of the currently running task and avoids that lower-priority tasks preempt higher-priority tasks. In a proposal by Leyva-del-Foyo et al. a custom interrupt controller [15] is used to realize dynamic priorities for the IRQ lines. Additionally, all ISRs become ISTs, which enables the Field-Programmable Gate Array (FPGA) to unify their synchronization and scheduling. While these solutions promise to have little overhead, the requirements of additional interrupt hardware can not be met with COTS microcontrollers.

Other works try to mitigate the unwanted effects using software-only. In [16], Leyva-del-Foyo et al. enhance the applicability of their concept presented in [15] with an implementation for standard PC interrupt hardware. Ober et al. patented a solution where the decision whether to wake an IST or not is based on a unique priority per task and a global interrupt priority value [17].

Prominent work regarding the unification of priority spaces is the approach implemented in the *Sloth* OS [18], [19]. Instead of using a software scheduler for threads, every control flow is designed as a thread-related system call using the hardware interrupt system. By letting the hardware manage all control flows, ISRs and (other) threads preempt each other in accordance with their priorities.

Although sophisticated mitigation approaches are being developed, only a few works quantify the impact of IRQ loads on real-time tasks. In [20], the authors explicitly specify the IRQ load caused with network packets (20 Hz) while measuring ISR latencies. Furthermore, Regehr et al. present a collection of approximate IRQ frequencies [21].

The research described shows that there are promising solutions for an improved latency or efficiency of interrupt handling. However, none of the examined papers focus on the impact of IRQs on real-time tasks and do not quantify the effects directly.

VII. Conclusion

The IoT introduces network interfaces and full IP stacks to microcontrollers running real-time applications. These open up the systems to interrupts network-triggered IRQs. We therefore analyzed the impact of network packet floods to the lateness and performance of real-time tasks on two state-of-the-art IoT MCUs. Our results show that the execution of network stack tasks on IoT devices can pose a significant threat to real-time guarantees and that the ISR executions themselves have a similar, yet less severe impact on critically prioritized tasks in comparison to the entire packet handling.

The results have shown that a more comprehensive study with a broader range of IoT devices and test scenarios is in order. Furthermore, we will investigate mitigation techniques for NIC-generated ISR delays in real-time systems.

Acknowledgments

We thank Martin Haug and Laurenz Mädje for their support during experiment implementations.

References

[1] A. Malinowski and H. Yu, "Comparison of embedded system design for industrial applications," *IEEE Transactions on Industrial Informatics*, vol. 7, no. 2, pp. 244–254, 2011.

[2] Chen Youdong, Wei Hongxing, and Wang Tianmiao, "Embedded control system for industrial robots," in *2010 International Conference on Computer, Mechatronics, Control and Electronic Engineering*, vol. 5, 2010, pp. 122–125.

[3] Bin Li and Jinhui Lei, "Design of industrial temperature monitoring system based on single chip microcontroller," in *2011 International Conference on Computer Science and Service System (CSSS)*, 2011, pp. 342–344.

[4] T. N. B. Anh and S. Tan, "Real-time operating systems for small microcontrollers," *IEEE Micro*, vol. 29, no. 5, pp. 30–45, 2009.

[5] S. Nuratch, "The iiot devices to cloud gateway design and implementation based on microcontroller for real-time monitoring and control in automation systems," in *2017 12th IEEE Conference on Industrial Electronics and Applications (ICIEA)*, 2017, pp. 919–923.

[6] C. Zhang and R. Green, "Communication security in internet of thing: preventive measure and avoid ddos attack over iot network," in *Proceedings of the 18th Symposium on Communications & Networking*, pp. 8–15.

[7] C. Dovrolis, B. Thayer, and P. Ramanathan, "Hip: hybrid interrupt-polling for the network interface," *ACM SIGOPS Operating Systems Review*, vol. 35, no. 4, pp. 50–60, 2001.

[8] H. Petersen, M. Lenders, M. Wählisch, O. Hahm, and E. Baccelli, "Old wine in new skins? revisiting the software architecture for ip network stacks on constrained iot devices," in *Proceedings of the 2015 Workshop on IoT Challenges in Mobile and Industrial Systems*, ser. IoT-Sys '15. ACM, 2015, p. 31–35.

[9] X. Bellekens, A. Seeam, K. Nieradzinska, C. Tachtatzis, A. Cleary, R. Atkinson, and I. Andonovic, "Cyber-physical-security model for safety-critical iot infrastructures," in *Wireless World Research Forum Meeting*, vol. 35, 2015, p. 18.

[10] N. Klingensmith and S. Banerjee, "Hermes: A real time hypervisor for mobile and iot systems," in *Proceedings of the 19th International Workshop on Mobile Computing Systems & Applications*, 2018, pp. 101–106.

[11] M. Bunnell, "Operating system architecture using multiple priority light weight kernel task based interrupt handling," uS Patent 5,469,571.

[12] Oracle, "Oracle solaris 11.4 programming interfaces guide." [Online]. Available: https://docs.oracle.com/cd/E37838_01/html/E61059/chap7rt-19493.html

[13] Microsoft, "History of windows embedded compact 7," 2013. [Online]. Available: https://web.archive.org/web/20130412030437/http://www.microsoft.com/windowsembedded/en-us/evaluate/history-of-windows-embedded-compact-7.aspx

[14] T. Gomes, P. Garcia, F. Salgado, J. Monteiro, M. Ekpanyapong, and A. Tavares, "Task-aware interrupt controller: Priority space unification in real-time systems," *IEEE Embedded Systems Letters*, vol. 7, no. 1, pp. 27–30, 2015.

[15] L. E. Leyva-del Foyo and P. Mejia-Alvarez, "Custom interrupt management for real-time and embedded system kernels," in *Proceedings of the Embedded Real-Time Systems Implementation Workshop at the 25th IEEE International Real-Time Systems Symposium (RTSS'04)*. IEEE.

[16] L. E. Leyva-del Foyo, P. Mejia-Alvarez, and D. de Niz, "Predictable interrupt management for real time kernels over conventional pc hardware," in *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*. IEEE, pp. 14–23.

[17] R. E. Ober, R. D. Arnold, D. F. Martin, and E. K. Norden, "Interrupt and trap handling in an embedded multi-thread processor to avoid priority inversion and maintain real-time operation," Patent US Patent 7,774,585.

[18] W. Hofer, D. Lohmann, F. Scheler, and W. Schröder-Preikschat, "Sloth: Threads as interrupts," in *2009 30th IEEE Real-Time Systems Symposium*. IEEE, 2009, pp. 204–213.

[19] W. Hofer, D. Lohmann, and W. Schröder-Preikschat, "Sleepy sloth: Threads as interrupts as threads," in *2011 IEEE 32nd Real-Time Systems Symposium*. IEEE, 2011, pp. 67–77.

[20] P. Regnier, G. Lima, and L. Barreto, "Evaluation of interrupt handling timeliness in real-time linux operating systems," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 6, p. 52–63, 2008.

[21] J. Regehr and U. Duongsaa, "Preventing interrupt overload," *SIGPLAN Not.*, vol. 40, no. 7, pp. 50–58, 2005.