Hopter: Leveraging Augmented Rust Semantics and Soft-Locks for a Safe, Robust, and Responsive Embedded Operating System

Anonymous Author(s)

Abstract

This is an abstract.

1 Introduction

The number of embedded devices has been constantly growing in the past decade and has been predicted to continue to grow in the next decades.

Embedded microprocessors are not noticeably improving their performance in recent years. Yet with the growing market size, it is urging to bring important features like memory safety to embedded systems. Previous research has attempted to use managed programming languages like Java or WASM on microcontrollers, but the performance sucks.

The emerging programming language, Rust, offers an alternative solution to achieve memory safety while not sacrificing performance. Previous operating systems like Theseus attempts to express system invariants with Rust's type system and leverages the compiler to enforce them, called the intra-lingual approach. [More details into intra-lingual approach: System resource ownership is tracked by objects on the task's stack.]

However, Rust fails to guarantee complete memory safety because its semantics assume an infinite size of function call stacks, which is especially unrealistic for microcontrollers having usually tens to a few hundreds KiB of SRAM. Existing vulnerability reports have confirmed the occurrence of stack overflows in Rust libraries, including those suitable for embedded use [9–14]. Common approaches to detect stack overflows, like stack protectors (canaries) [2, 16] or periodic stack pointer inspection [3], are belated efforts. At the time of discovering an overflow, the system memory would have already been corrupted and the door to attacks would have been long opened.

Although Rust defines panic as a language exception mechanism to handle fatal errors and potentially recover the system from them, a full stack unwinder is, however, usually unavailable on deeply embedded systems. Without it, Rust panics lead to hang or reset of the application [17] or the whole system [7, 8]. But even with an unwinder, the current panic mechanism is incapable of handling stack overflows: A drop handler, a.k.a. object destructor function, can overflow the stack, but must not panic. Ironically, Rust's canonical programming paradigm exacerbates the problem by encouraging drop handlers to be the one that overflows the stack (§X.Y).

Language restrictions of Rust also hinders zero-latency interrupt handling, which is important for fast response to events on embedded systems. To achieve zero-latency, the kernel must never disable interrupts even in critical sections. Previous solutions [18–20, 23, 24] are infeasible because they require a global serialization queue holding closures. The closures are inserted by interrupt handlers interacting with kernel objects, and are executed in sequence to avoid race conditions. However, Rust forbids such code pattern because the compiler cannot statically verify the lifetime of the references contained within the closures pointing to kernel objects.

In this paper, we seek to answer the following question: Can we bring stack memory safety, system robustness, and interrupt responsiveness to a Rust-based embedded system, while being transparant to application programmers? We present a positive answer with the Hopter embedded operating system, following a co-design between the kernel and the compiler to complete memory safety and achieve panic resilience. We also for the first time brings zero-latency interrupt handling to a Rust-based system supporting threaded tasks, using a novel synchronization primitive called the soft-lock.

Our key to stack memory safety and overflow resilience is to augment Rust semantics with the awareness of finite-sized stacks (FS-semantics). The intuition is to turn each function call site as a potential panic site. A function call made with insufficient free stack space will raise a panic. In case of a drop handler or its callee overflows a stack, the stack will be temporarily allowed to extend for further execution, and the panic will be deferred to occur after the drop handler finishes.

FS-semantics require both runtime support in the kernel and compiler modification. The new semantics result in more CPU instructions and render some compiler optimizations infeasible but fortunately have small impact on the compiled code size and speed (§X.Y). To support stack extension on microcontrollers without virtual memory, we leverage an existing technique called the segmented stack. As a byproduct, application tasks may time-multiplex the memory for stacks, resulting in reduced system memory footprint (§X.Y).

With stack overflows being unified with other fatal errors as Rust panics, Hopter then leverages a customized stack unwinder to reclaim the resources upon panics and allows failed tasks to restart. When possible, Hopter performs concurrent restart of failed tasks, where the new restarted task

instance runs concurrently with the unwinding procedure for faster recovery speed.

Bringing zero-latency interrupt handling to Hopter requires new code patterns that satisfy the Rust compiler. We observe that, conceptually, we must break the global serialization queue into multiple local queues, each associated with a kernel object to record operations to be performed on that object. Yet in implementation, most kernel objects need not a queue but only an integer counter. Our novel synchronization primitive, the soft-lock, grants direct access to the protected object when not under contention, or otherwise forces the code to push intended operations to the associated queue, which will be committed after the contention is over.

We open-source Hopter as a Rust library crate. Application programmers using Hopter declare tasks or interrupt handlers as Rust threads, which transparently enables them to enjoy memory safety, system robustness, and interrupt responsiveness. Although Hopter requires a customized compiler, the language syntax remains the same and the semantics compatible. Hopter supports unmodified third-party hardware abstraction layer (HAL) crates, allowing application programmers to embrace the wide Rust ecosystem while being protected.

We summarize our contributions as follows:

- We introduce the finite-sized stack semantics (FS-semantics) of Rust and implement it through compiler modification and kernel runtime support. We show that the new semantics incur negligible performance loss and small binary size increase.
- We introduce soft-lock, a novel synchronization primitive to enable zero-latency interrupt handling on Rust-based systems with threaded tasks. Our benchmark shows that interrupt handling latency is a small constant with soft-locks.
- We design, implement, and open-source an embedded operating system called Hopter that incorporates the FS-semantics and soft-locks, achieving memory safety, system robustness, and interrupt responsiveness, completely with software.

2 Evaluation

We seek to answer the questions below with our evaluation:

(1) What is the overhead of binary size and CPU performance introduced by Hopter to ensure stack memory safety, fatal error resilience, and zero-latency interrupt handling?
(2) Can soft-lock deliver a low and constant interrupt response time regardless of system load?
(3) Does Hopter support performance demanding applications?
(4) Can Hopter recover a failed time-sensitive task in time in a mission critical scenario?

Using a set of micro-benchmarks with controlled workload, we compare Hopter with two well-known embedded OSes: FreeRTOS [1], the most widely used embedded OS, and Tock [17], the state of the art Rust-based embedded OS. Our results demonstrate that Hopter is suitable for performance demanding workload, and it is a safer and more robust alternative to FreeRTOS with moderate binary and performance overhead. When applications are benign, Hopter allows significantly smaller system size and faster response time than Tock. Additionally, Hopter performs semantic cleanup of failed tasks, allowing the system to recover from errors that are unrecoverable if developed with the other two OSes.

We further develop a flight control program using Hopter for the Crazyflie [4] miniature drone as a macro-benchmark. Hopter not only exhibits its capability of supporting cooperating tasks with frequent synchronization, but can also recover a failed time-sensitive task in time and maintain the drone's hovering when such error occurs. Its soft-lock supporting zero-latency interrupt handling is also essential to the correct functioning of the radio module.

2.1 Comparison Across OSes

We perform our micro-benchmarks with the STM32F412G-Discovery board equipped with an Arm Cortex-M4F CPU, 256 KiB SRAM, and 1 MiB Flash. We configure the CPU to run at 96 MHz and the compiler to optimize for speed in all experiments.

To demonstrate the overhead contributed by each of the features of Hopter, we break down its implementation as shown in Figure 1. Starting from the "bare" version that includes none of the features, we progressively add them towards the full-featured version. The "soft-lock" version enables zero-latency interrupt handling by substituting soft-locks for interrupt masking in critical sections. The "seg-stack" version ensures stack memory safety and allows stack extension by generating prologues for each compiled function and incorporating a stack extension runtime. The "unwind" version supports resource reclamation upon Rust panic by incorporating a customized stack unwinder. The unwinder also requires the compiler to generate unwind tables and landing pads. Finally, the full Hopter version enables system resilience against stack overflows by instrumenting drop handlers and disabling compiler optimizations based on the `nounwind` attribute.

2.1.1 Minimum System Size. We build a blinking LED application, the "hello world" program for the embedded world, using three OSes under comparison. We assume an application programmer's role, i.e., using the OS and hardware abstraction layer (HAL) library as-is through their public APIs,

	Minimum System		Latency			Feature		
	Flash (KiB)	†SRAM (KiB)	* Task-Task (µs)	** IRQ (µs)	** IRQ-Task (µs)	Responsive	Safe	Robust
FreeRTOS	12.98	1.69	8.0	2.20 (0.39)	12.76 (0.73)	✔	✖	✖
Tock	‡147.5 + 6.60	‡66.53 + 0.75	264.7	151.54 (26.03)	365.14 (26.71)	✖	✔	✔
Hopter	**29.19**	**1.02**	**19.5**	**1.46 (0.00)**	**32.34 (1.99)**	✔	✔	✔
Hopter-unwind	24.30	1.02	13.7	1.32 (0.00)	28.20 (1.54)	✔	✔	✖
Hopter-seg-stack	13.46	0.86	13.9	1.18 (0.01)	24.30 (1.90)	✔	✔	✖
Hopter-soft-lock	10.85	0.77	13.6	1.16 (0.00)	27.10 (1.75)	✔	✖	✖
Hopter-bare	9.11	0.50	15.1	5.66 (1.35)	25.80 (1.56)	✖	✖	✖

Table 1: Comparing Hopter to FreeRTOS and Tock. Hopter's requires more flash than FreeRTOS's mainly because it includes additional components to improve robustness, but it is significantly less than Tock that compiles its kernel separately. Hopter has the lowest and consistent interrupt response latency, benefiting from the soft-lock. Hopter's context switch performance and task notification latency is close to FreeRTOS's and an order of magnitude lower than Tock's. (†: Excluding function call stacks, whose sizes are configurable. ‡: Kernel plus application size. *: Average from 10,000 trials. **: Maximum and standard deviation from 10,000 trials.)

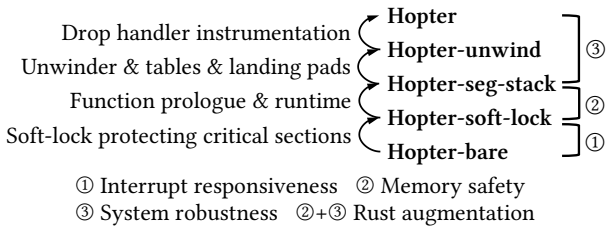

Figure 1: A breakdown of Hopter's unique features. The bottom "bare" version includes none of the components while the top includes all. Each version in the hierarchy has one more feature included compared to the one below it.

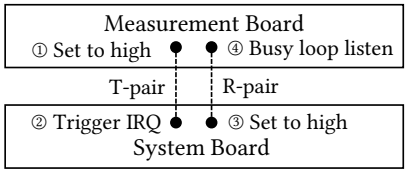

Figure 2: Experiment setup to measure interrupt acknowledgment latency. The measurement board raises the T-pair pin to high to trigger an interrupt on the system board, which acknowledges it by raising the R-pair pin to high. The latency is measured by the time elapsed between setting the T-pair to high and the R-pair becomes high.

without modifying their internal implementation. We use the HAL library from `stm32-rs` [15] and STM32CubeF4 [29] for Hopter and FreeRTOS, respectively. Tock comes with its own HAL.

The minimum system columns in Table 1 list the flash and SRAM size. The flash overhead of Hopter mainly comes from the components to enhance system robustness. The stack unwinder in the "unwind" version contributes 7.71 KiB of size increase, the landing pads contribute 1.63 KiB, and the unwind tables contribute 1.50 KiB. From the "unwind" version to the full version, instrumenting the drop handlers increases the code size by 3.54 KiB and the unwind table by 0.63 KiB. Disabling compiler optimizations based on the `nounwind` function attribute increases the code size and unwind table size by another 0.16 KiB and 0.55 KiB, respectively. The function prologue for stack memory safety and the stack extension runtime each incur a minor flash overhead of 0.58 KiB and 2.03 KiB, respectively. Underlined numbers are constant while others grow with the binary size. The constant overhead will be amortized away when the application is more meaningful as in §2.2.

On microcontrollers that have more than 100 KiB of flash, the increased flash overhead will be less of a burden, rendering it feasible to switch from FreeRTOS to Hopter for improved safety and robustness. On the other hand, the system developed with Tock requires significantly larger flash and SRAM because its kernel is compiled separately from the application. The compiler toolchain is unable to remove unused code by the application from the kernel at compile time, resulting in the size bloat.

2.1.2 System Responsiveness. To compare the system responsiveness, we measure the following performance metrics: the latency of a context switch between two tasks and the latency to respond to an interrupt from a handler or task.

Context switch latency reflects the overhead for inter-task cooperation. We compute it by running two tasks performing context switches to each other using the most efficient synchronization primitive available, i.e., mailbox on Hopter, task notification on FreeRTOS, and inter-process communication (IPC) on Tock. The Task-Task column in Table 1 lists the average latency by measuring 10,000 context switches. Hopter's latency is higher than FreeRTOS's mainly due to

the use of safe Rust in implementing task lists. The optimized task list structure and operations found in FreeRTOS are incompatible with Rust's ownership model. Nevertheless, Hopter's latency is on the same order of magnitude with FreeRTOS's, allowing it to support multiple cooperating tasks with frequencies up to 1,000 Hz (§2.2). Tock on the other hand is substantially slower, because each context switch from a task to another requires four context switches between the kernel and user space, rendering Tock not suitable for performance demanding workloads. We discuss Hopter's slowdown caused by drop handler instrumentation in §2.2.1.

We further measure interrupt response latency, which determines if the system may miss an ephemeral event. We use two microcontroller boards for the measurement: an STM32F412G-Discovery board to run the system under test (system board) and an STM32F411E-Discovery board as the measurement instrument (measurement board). As shown in Figure 2, the boards are connected via two pairs of general-purpose input/output (GPIO) pins, referred to as the trigger pair (T-pair) and the response pair (R-pair). On the system board, we register an interrupt handler to be invoked upon rising edges on the T-pair pin. The system board then acknowledges the interrupt by setting the R-pair pin high. The measurement board measures the acknowledgment latency by timing the interval between setting the T-pair pin high and detecting the R-pair pin going high, using a busy loop. The measurement is precise to 0.02 µs.

The IRQ column in Table 1 reports the maximum latency observed in 10,000 tests where the interrupt handler directly acknowledges the interrupt[1], while the IRQ-task column lists the numbers when a task acknowledges the interrupt notified by the handler. These numbers are obtained when two other tasks are context switching to each other as the system background workload. Hopter has the lowest and consistent latency when acknowledging in the interrupt handler, benefiting from the soft-lock mechanism. When acknowledging in a task, Hopter becomes slower than FreeRTOS due to its slower context switch speed. In both cases, Hopter is orders of magnitude faster than Tock.

Hopter's soft-lock is effective in maintaining a consistent interrupt response latency regardless of the background workload, as shown in Figure 3. The slight increase in the number under load is due to the eviction of cached instructions and literals in the ART accelerator for flash [25]. In contrast, FreeRTOS's latency is sensitive to the background workload. Masking interrupts in its critical sections causes delay in interrupt response, and such delay will be exacerbated if the CPU runs at a lower frequency or the kernel is under heavier load.

[1]For Tock, we register the handler as a capsule in the kernel space.

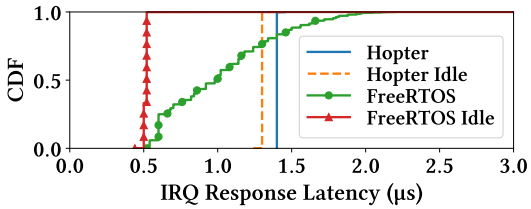

Figure 3: Hopter's interrupt response latency is almost constant using soft-lock. Background workload causes slight increase in latency due to the stress on instruction and literal cache. FreeRTOS is sensitive to background workload, because it masks interrupts inside its critical sections.

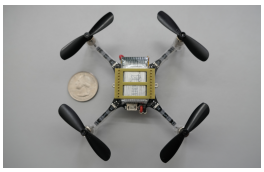

Figure 4: Crazyflie 2.1, a commercial off-the-shelf miniature drone. We implement a flight control program with Hopter to evaluate its capability of supporting real-time applications.

2.1.3 Semantic Cleanup. Hopter allows a faulty application task to perform arbitrary cleanup logic during resource reclamation, which enables the system to recover from more complex scenarios and overcome the limitations of a simple task restart. We implement an application as a proof of concept consisting of three tasks: A serial server task that transmits data received from other tasks over the serial interface, and two client tasks that periodically send data to the server task. To prevent undesirable data interleaving between sending tasks, a client task first acquires a lock on the server before sending data and releases it when finishes.

We deliberately trigger an out-of-boundary array write in one client task while it is holding the lock. On Hopter, the fault reveals as a Rust panic, causing the task to be restarted. During stack unwinding, the drop handler of the lock guard object is invoked to release the lock. After the restart, both client tasks proceed as normal. In contrast, on Tock, if the written address falls in the task's allowed address range, it becomes a silent data corruption within the task. Otherwise, the kernel detects the fault and restart the task, but the lock will not be released, subsequently causing a deadlock. Since FreeRTOS has no safety protection, the out-of-boundary write either causes a system wide data corruption, or trigger a hardware fault that hangs or resets the system.

2.2 Flying a Drone

As a macro-benchmark, we develop a flight control program with Hopter using the commercially available Crazyflie 2.1 hardware platform [4] as shown in Figure 4. The drone is

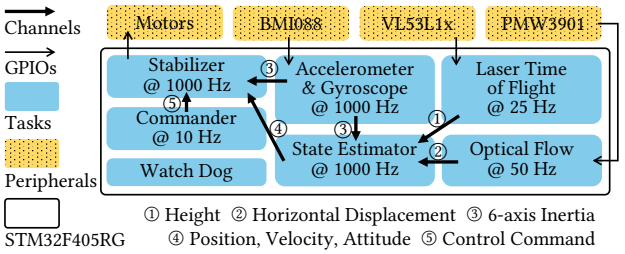

Figure 5: The flight control application consists of cooperating periodical tasks running at high frequency. Tasks synchronize and exchange data through the data channels provided by Hopter. Peripherals are connected via GPIO pins.

	Flash (KiB)	SRAM (KiB)	CPU
Hopter	**220.07**	**7.51**	**56.2%**
Hopter-unwind	177.22	7.51	48.1%
Hopter-seg-stack	151.00	7.35	47.8%
Hopter-soft-lock	144.50	7.00	46.1%
Hopter-bare	148.58	6.52	48.3%

Table 2: Binary size, SRAM usage, and CPU load of the flight control program. The SRAM usage excludes function call stacks, whose sizes are configurable. The CPU load is the average of 100 measurements when the drone is hovering.

powered by an STM32F405RG microcontroller, featuring a Cortex-M4F CPU with a maximum clock speed of 168 MHz, 192 KiB of SRAM, and 1 MiB of flash storage.

We implement the flight control program in Rust consisting of around 10,000 lines of code, in which around 6,700 lines are peripheral driver code manually translated from open-source libraries [6, 21, 22]. We use the HAL library from `stm32-rs` [15] to access perihperals. The flight control program contains 22 unsafe Rust statements, mostly for interrupt configuration.

The flight control program includes seven periodical tasks that collectively manage flight control as shown in Figure 5. Three tasks are responsible for reading data from the inertial sensors, the height sensor, and the optical flow sensor, respectively. Subsequently, the estimator task obtains the collected data, through the data channel synchronization primitive provided by Hopter, and computes the the drone's position and attitude with the Kalman filter algorithm. Finally, the stabilizer task utilizes the output from the estimator to modulate the power distribution across the four propeller motors, aiming to maintain the drone's stability and follow the commands sent from the commander task. For operational safety, the watchdog task oversees other flight control tasks, and if any of them is unresponsive for more than one second, the watchdog task will shut off all motors.

2.2.1 System Size and CPU Load. Table 2 shows the flash and SRAM usage of the flight control program and the CPU load when the drone is hovering. As a real-world application, the binary overhead introduced by the recovery mechanism for panic and stack overflow is amortized, reducing it from 173% as in the minimum system (§2.1.1) to 46% in the flight control program.

Drop handler instrumentation (§X) introduces the largest overhead in both flash size and CPU load, which is counter intuitive since the instrumented code appears to be short and applies only to drop handler functions. However, we observe a cascading effect in the generated code that leads to the size bloat and higher CPU load.

(1) ARMv7, as a RISC instruction set, must load the address of global variables into registers before accessing them, resulting in longer instruction sequences.
(2) There is more register spilling, further increasing the instruction sequence length.
(3) Larger function body causes the compiler not to inline the drop handler function, resulting in more function calls and returns.
(4) An outlined function in turn includes additional segmented stack prologue instructions.

Executing instructions in-place from flash (XIP) exacerbates the CPU load increase after the drop handler instrumentation. Since Cortex-M4F is not equipped with a branch predictor, the ART accelerator cannot prefetch instructions at the jump target. Thus, any jump, including function calls and returns, will incur a 5 cycle stall [26] if the target instruction is not cached. Since the instruction cache is 1 KiB [27], cache miss is also very likely.

However, drop handler instrumentation is indispensable for correct recovery from stack overflow. We perform a static stack depth analysis for the seven flight control tasks, ignoring indirect function calls[2]. Five of the tasks reach their respective maximum stack size inside a drop function. Without the instrumentation, initiating stack unwinding inside a drop handler will cause a system reset.

2.2.2 In-flight Fatal Error Tolerance. We deliberately introduce fatal errors into the flight control program while the drone is hovering to assess Hopter's ability to recover the system from them. The drone has no visible disturbance when we introduce a panic into an interrupt handler or to the flight control tasks except the stabilizer task. The drone rotates for around 20 degrees along the yaw axis and drops a few centimeters when the stabilizer task panics, but can still maintain hovering following the task restart. The reason for the stabilizer task being the most sensitive one to fatal errors is that it directly modulates the power of motors. When other

[2]E.g., calls through function pointers or trait objects. 0.1% of all calls.

	Panic	OF-Recur-4	OF-Recur-10
Unoptimized (µs)	197	268	402
Ctxt. Reuse (µs)	134	207	335
Concur. Restart (µs)	189	192	190
# Stack Frames	6	7	13
# Landing Pads	2	6	12

Table 3: Stabilizer task recovery latency upon fatal errors and stack unwinder workload. The error is either a deliberate `panic!()` or a stack overflow after some function recursions. Both the task context reuse and concurrent restart optimization can speed up the recovery upon panic and stack overflow. Concurrent restart maintains a constant recovery latency regardless of the stack depth.

tasks fail, the stabilizer task can temporarily use stale data to control the propellers, masking the error from observation.

We further measure the recovery latency of the stabilizer task upon different fatal errors. For the panic test, we place a `panic!()` statement in the code that updates the stabilizer states, simulating an out-of-boundary array access or a failed assertion. For the stack overflow test, we insert a call to a recursive function that defines a local object with a drop handler, which will overflow the stack after 4 or 10 recursions. Table 3 lists the number of stack frames to unwind and landing pads to invoke for each test case. The recovery latency is measured as the time elapsed between the occurrence of the error and the execution of the entry function of the restarted task instance.

The measurement result demonstrates that both the task context reuse and concurrent restart optimization can reduce the recovery latency. Task context reuse outperforms concurrent restart when the stack is shallow, but concurrent restart enables faster recovery by maintaining a constant latency when the stack is deep.

In all three cases, the stabilizer task can recover before the next execution interval. The observed disturbance to the drone is due to the new stabilizer instance not having any command to follow. The motors are kept at zero power until receiving the next command.

2.2.3 In-flight Interrupt Response Latency. We measure the interrupt response latency while setting the drone to hover, following the same experimental setup as in §2.1.2. With soft-lock, the maximum interrupt response latency over 10,000 measurements is 1.00 µs with a standard deviation of 0.01 µs. However, without soft-lock, the maximum latency rises to 14.00 µs with a standard deviation of 0.72 µs, due to the critical sections preventing the system to respond to the interrupt for an extensive period of time.

Prompt interrupt response is essential for the proper functioning of the drone's radio module. The drone employs an nRF51822 chip that forwards received radio packets to the main microcontroller via a universal asynchronous receiver-transmitter (UART) serial interface at baud rates up to 2 Mbps. In the Syslink protocol [5] used by the drone, the packet length is determined by the first four bytes, so direct memory access (DMA) can only be initiated after these bytes are received. Without DMA, the system must promptly respond to the interrupt after the UART receives a byte. Considering that each data byte carries an overhead of one start bit and one stop bit, an interrupt response delay over 5 µs will cause an overrun of the receive shift register [28], resulting in data loss. Only with soft-lock can the radio module function correctly.

References

[1] Amazon.com, Inc. 2020. FreeRTOS: Real-time operating system for microcontrollers and small microprocessors. https://www.freertos.org. Version 10.3.1.

[2] Arm Ltd. 2023. *Arm Compiler Version 6.6 armclang Reference Guide.* Section 2.28.

[3] AWS open source. 2024. *FreeRTOS Documentation.* https://www.freertos.org/Documentation/00-Overview Chapter: FreeRTOS Stack Usage and Stack Overflow Checking.

[4] Bitcraze. 2020. Crazyflie: A flying open development platform. https://www.bitcraze.io/.

[5] Bitcraze. 2024. Syslink protocol version 2024.10. https://www.bitcraze.io/documentation/repository/crazyflie2-nrf-firmware/2024.10/protocols/syslink/.

[6] Bosch. 2023. BMI088. https://github.com/BoschSensortec/BMI08x-Sensor-API.

[7] Embassy Project Contributors. 2023. Embassy: The next-generation framework for embedded applications. https://embassy.dev.

[8] RTIC Contributors. 2023. RTIC: The hardware accelerated Rust RTOS. https://rtic.rs/2/book/en/.

[9] MITRE Corporation. 2019. CVE-2019-25001: Flaw in CBOR deserializer allows stack overflow. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-25001. Accessed: 2024-10-02.

[10] MITRE Corporation. 2020. CVE-2020-35858: Parsing a specially crafted message can result in a stack overflow. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-35858. Accessed: 2024-10-02.

[11] MITRE Corporation. 2024. CVE-2024-36760: A stack overflow vulnerability found in version 1.18.0 of rhai. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2024-36760. Accessed: 2024-10-02.

[12] The Rust Security Advisory Database. 2018. RUSTSEC-2018-0005: Uncontrolled recursion leads to abort in deserialization. https://rustsec.org/advisories/RUSTSEC-2018-0005.html. Accessed: 2024-10-02.

[13] The Rust Security Advisory Database. 2023. RUSTSEC-2023-0080: Buffer overflow due to integer overflow in transpose. https://rustsec.org/advisories/RUSTSEC-2023-0080.html. Accessed: 2024-10-02.

[14] The Rust Security Advisory Database. 2024. RUSTSEC-2024-0012: Stack overflow during recursive JSON parsing. https://rustsec.org/advisories/RUSTSEC-2024-0012.html. Accessed: 2024-10-02.

[15] Daniel Egger. 2024. Peripheral access API for STM32F4 series microcontrollers. https://crates.io/crates/stm32f4xx-hal. Version 0.22.0.

[16] Free Software Foundation, Inc. 2023. *GCC 15.0.0 Manual.* Free Software Foundation. https://gcc.gnu.org/onlinedocs/gcc/ Section 3.12, Program Instrumentation Options.

[17] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. Multiprogramming a

64kb computer safely and efficiently. In *Proc. ACM SOSP*.

[18] Anthony J Massa. 2002. *Embedded software development with eCos*. Prentice Hall Professional.

[19] Ralph Moore. 2005. Link Service Routines. *Micro Digital* (2005).

[20] John Pagonis. 2004. Overview of Symbian OS hardware interrupt handling. *Symbian, www.symbian.com* (2004).

[21] Pimoroni. 2023. PMW3901. https://github.com/pimoroni/pmw3901-python/tree/master.

[22] Pololu. 2022. Vl53l1x. https://github.com/pololu/vl53l1x-arduino.

[23] Friedrich Schon, Wolfgang Schroder-Preikschat, Olaf Spinczyk, and Ute Spinczyk. 2000. On interrupt-transparent synchronization in an embedded object-oriented operating system. In *Proceedings Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2000)(Cat. No. PR00607)*. IEEE, 270–277.

[24] Wolfgang Schröder-Preikschat. 1994. *The logical design of parallel operating systems*. Prentice-Hall, Inc.

[25] STMicroelectronics. 2020. RM0402 (Rev.6) Reference manual for STM32F412 advanced Arm-based 32-bit MCUs. Chapter 3.4.2: Adaptive real-time memory accelerator (ART Accelerator).

[26] STMicroelectronics. 2024. RM0090 (Rev.21) Reference manual for STM32F405/415, STM32F407/417, STM32F427/437 and STM32F429/439 advanced Arm-based 32-bit MCUs. Table 11. Number of wait states according to CPU clock (HCLK) frequency (STM32F405xx/07xx and STM32F415xx/17xx).

[27] STMicroelectronics. 2024. RM0090 (Rev.21) Reference manual for STM32F405/415, STM32F407/417, STM32F427/437 and STM32F429/439 advanced Arm-based 32-bit MCUs. Chapter 3.5.2: Adaptive real-time memory accelerator (ART Accelerator).

[28] STMicroelectronics. 2024. RM0090 (Rev.21) Reference manual for STM32F405/415, STM32F407/417, STM32F427/437 and STM32F429/439 advanced Arm-based 32-bit MCUs. Chapter 30.3.3: Receiver - Overrun Error.

[29] STMicroelectronics. 2024. STM32CubeF4 HAL driver MCU component. https://github.com/STMicroelectronics/stm32f4xx_hal_driver/releases/tag/v1.8.3. Version 1.8.3.