Hopter: a Safe, Robust, and Responsive Embedded Operating System

Submission #397 Anonymous Author(s)

Abstract

Microcontroller-based embedded systems are vulnerable to memory safety errors and must be robust and responsive because they are often used in unmanned and mission-critical scenarios. The Rust programming language offers an appealing compile-time solution for memory safety but leaves stack overflows unresolved and foils zero-latency interrupt handling. We present Hopter, a Rust-based embedded operating system (OS) that transparently provides memory safety, system robustness, and interrupt responsiveness to embedded systems. Hopter executes Rust code under a novel finite-stack semantics that converts stack overflows into Rust panics, enabling recovery from fatal errors through stack unwinding and restart. Hopter also employs a novel mechanism called soft-locks so that the OS never disables interrupts. We compare Hopter with other well-known embedded OSes using controlled workloads and report our experience using Hopter to develop a flight control program for a miniature drone. We demonstrate that Hopter is well-suited for resource-constrained microcontrollers and supports error recovery for real-time workloads.

1 Introduction

Embedded systems have seen increasing adoption in the past decade, and their number is predicted to increase even more in the coming decades [34–36]. However, microcontrollers, the heart of many embedded systems, are not noticeably improving in their performance or available resources. With embedded systems increasingly used in mission-critical applications, it is imperative to bring memory safety and system robustness without demanding more resources or sacrificing system responsiveness.

The Rust programming language offers an attractive solution to achieve memory safety without sacrificing performance. It promises memory safety primarily through compile-time checks, incurring little runtime overhead, and has seen adoption in operating system (OS) development [2–4, 12, 17, 47] and embedded systems [1, 38, 60].

However, challenges persist when programming microcontrollers with Rust. First, memory safety errors linger on even with safe Rust, because its semantics assume an infinite size of function call stacks, which is especially problematic for microcontroller-based embedded systems where there is no virtual memory and only 10s to 100s KiB of SRAM are usually available. Vulnerability reports have shown the existence of stack overflows in Rust libraries, including those intended for embedded use [18–20, 23–25]. Second, Rust raises panics upon fatal errors, but the stack unwinder is usually unavailable on microcontrollers. Without it, panics lead to hang or reset of the application [38] or the whole system [15, 16]. Finally, language restrictions of Rust make it difficult to implement zero-latency interrupt handling, where the OS never disables interrupts to ensure timely response to events. Known solutions [44, 45, 49, 54, 55] are infeasible with safe Rust because they struggle to pass the compile-time check. §2 elaborates on the challenges.

In this paper, we seek to answer the following question: Can we bring memory safety, system robustness, and interrupt responsiveness to a Rust-based embedded system while remaining transparent to application logic? We present a positive answer with the Hopter embedded OS, following a co-design between the OS and the compiler to complete memory safety and achieve fatal error resilience. Hopter also brings zero-latency interrupt handling to a Rust-based system running threaded tasks, using a novel mechanism called soft-locks.

Hopter's key to stack memory safety and overflow resilience is to augment Rust with the finite-stack semantics (FS-semantics) relying on compile-time code instrumentation and runtime OS support. FS-semantics treat each function call site as a potential panic site. Functions check the remaining stack space before executing their body, and will raise a panic if insufficient. In case of a drop handler or its callee overflows a stack, Hopter will temporarily extend the stack for further execution inside the drop handler, using the technique called segmented stacks [32, 42, 64], and raise the panic after the drop logic finishes.

With stack overflows being unified with other fatal errors as Rust panics, Hopter then reclaims the resources upon panics utilizing a customized stack unwinder, which allows to automatically restart failed tasks. When possible, Hopter performs concurrent restart to expedite recovery, where the restarted task instance runs concurrently with the unwinding procedure of the failed one.

Soft-locks enable zero-latency interrupt handling by serializing mutations individually for each OS kernel object, avoiding the global serialization queue to allow the Rust compiler to statically verify memory safety. When not under contention, soft-locks grant direct access to the protected objects. Otherwise, soft-locks only allows the code to record intended operations and will commit them at a later time when the contention is over.

We implement and open-source Hopter as a Rust library crate. Hopter requires a customized compiler to compile the application, but the Rust syntax remains the same and the semantics compatible. Hopter also supports unmodified third-party hardware abstraction layer (HAL) crates, allowing application programmers to embrace the wide Rust ecosystem. We observe a 56% increase in flash usage and proportionally 15% higher CPU load incurred together by FS-semantics, the unwinding mechanism, and soft-locks.

We summarize our contributions as follows:

- We present the finite-stack semantics (FS-semantics) of Rust to achieve stack memory safety and overflow resilience, and we implement it through compile-time instrumentation and runtime OS support.
- We introduce soft-locks, a novel locking mechanism allowing zero-latency interrupt handling on Rust-based systems with threaded tasks.
- We design, implement, and open-source the Hopter embedded OS that incorporates FS-semantics and soft-locks, achieving memory safety, system robustness, and responsiveness while being transparent to application logic.
- We evaluate the code size and performance overhead incurred by FS-semantics, the unwinding mechanism, and soft-locks, and we show Hopter's suitability for resource constrained microcontrollers and real-time workloads.

2 Background

Microcontroller. Microcontrollers are widely used in embedded systems, from simple home appliances [31, 46, 62] to complex automotive control systems [26, 28, 37] and industrial automation [14, 43]. Many of these systems are unmanned or mission-critical, making robustness and the ability to recover from fatal errors essential. Due to cost and energy constraints, the hardware resources available on microcontrollers are usually limited and have not increased significantly over the past decades [38]. Microcontrollers typically provide hundreds of kilobytes to a few megabytes of flash for code and read-only data, and tens to hundreds of kilobytes of SRAM for runtime data. Consequently, operating systems designed for microcontrollers must be lightweight.

Microcontrollers operate without virtual memory. All code runs in a single physical address space where flash storage, SRAM, and peripheral registers are mapped. The CPU typically executes instructions directly from the byte-addressable flash, called execute in place (XIP), while function call stacks and mutable data reside in SRAM. The code on microcontrollers usually has unrestricted access to the entire address space, making the system prone to memory safety errors like buffer overflows and invalid pointer accesses, which can lead to system instability or security vulnerabilities.

Hardware-based memory protection, e.g., memory protection unit (MPU) [6] on Arm and physical memory protection (PMP) [53] on RISC-V, is available on high-end microcontrollers and has been actively exploited by some embedded OSes, for example, Tock [38]. Unfortunately, hardware-based protection introduce overheads in code size, memory usage, and runtime performance. MPU/PMP allows developers to define up to 16 memory regions with selective read, write, or execute permissions. However, each memory region must be aligned and sized to the nearest power of two, leading to wasted flash or SRAM due to internal fragmentation. Moreover, employing an MPU or PMP requires system calls to switch privilege modes via software interrupts, and arguments need to undergo marshaling and be verified by the kernel, adding performance overhead. In contrast, in systems without such protection mechanisms, system calls can be efficiently implemented as simple function calls. As a result, popular embedded OSes such as FreeRTOS [5] consider hardware-based protection optional, even if the system is written in an unsafe language like C.

Rust Programming Language. Rust is a modern systems programming language that provides memory and concurrency safety while offering direct control over hardware. It presents a promising alternative to hardware-based protection without the significant overhead incurred by managed languages. Rust has seen increasing adoption in OS development [2–4, 12, 17, 47] and embedded systems [1, 38, 60].

Rust manages resources through its ownership model without using a garbage collector. Each resource is owned by a variable, and when the variable goes out of scope, its drop handler, also called the destructor function in other languages, runs to release the resource. By default, function calls in Rust move the ownership of the argument variables to the callee. The callee function is in turn responsible for further calling the drop handlers of the arguments. Thus, call stacks are likely to reach their maximum depth while calling drop handlers, a phenomenon confirmed by our flight control program (§6.3).

Rust statically verifies the ownership model and performs lifetime analysis at compile time to ensure memory safety, unless opted out with the `unsafe` keyword. However, the static check prohibits some common code patterns from compiling. For instance, the lifetime analysis disallows a data structure from storing references unless their lifetimes in syntactical scope outlive the data structure's. Such restrictions prevent a Rust-based system from adopting prior implementations [44, 45, 49, 54, 55] to achieve zero-latency interrupt handling, where they leverage a global serialization queue to store pending operations containing references to various kernel objects. Hopter instead introduces a novel mechanism called soft-locks (§4.3) to achieve zero-latency interrupt handling.

Rust complements its compile-time check with a language exception mechanism called panic to prevent memory errors that can only be detected at runtime, such as out-of-bounds array access. A panic usually initiates a stack unwinding procedure, where an unwinder iterates through the function frames in the call stack and invokes the drop handlers of live objects to reclaim resources. This allows subsequent recovery from the error without restarting the entire system. However, embedded Rust systems usually lack a stack unwinder [15, 16, 38], and thus a panic will hang or reset the system. Hopter overcomes this limit by incorporating a stack unwinder to microcontrollers to enhance system robustness (§4.2).

However, Rust cannot guarantee stack memory safety by preventing function call stack overflows. Its semantics assume an infinite stack space for each running thread, which is particularly unrealistic on microcontrollers. Common approaches to detecting stack overflows, including stack protectors (canaries) [7, 29] and periodic stack pointer inspection [8], are belated efforts. The system memory is already corrupted by the time of detection. Prior Rust-based embedded systems either use MPU/PMP to forestall stack overflows [38], accepting the associated overhead, or employ only a single down-growing stack placed at the lower boundary of the SRAM region [15, 16], which restricts scheduling patterns. Hopter addresses this problem by running Rust code with finite-stack semantics (§4.1), which not only prevents stack overflows but also converts such errors into Rust panics to allow a unified recovery procedure through unwinding and restarting.

3 System Development with Hopter

Before we present the design (§4) and implementation (§5) of Hopter, we describe how a system developer may use it to develop an embedded system with application code.

3.1 Development Model

Hopter is open-source and a system developer receives it as a Rust library crate. Since Hopter supports threaded tasks and forsakes hardware-based protection, it can interoperate with third-party HAL libraries [27, 59], which significantly lowers the development effort. The developer must write application code in Rust and use a customized Rust compiler supplied by Hopter to build their system that uses Hopter as a dependency. Downloading and registering the customized compiler with `cargo` requires only three commands. The developer compiles the system with `cargo build` as usual and the Rust syntax remains the same.

Hopter expects application code to be benign, because application code may use `unsafe` Rust that potentially introduces memory errors. Hopter guarantees memory safety of the system as long as all `unsafe` code used by the application

```
#[main]
fn main() {
    task::build()
        .set_entry(|| another_task_main())
        .set_priority(8)
        .set_stack_limit(1024)
        .spawn_restartable()
        .unwrap();
}
```

Listing 1: Application code starts the execution from its main function marked by the `#[main]` attribute. Other tasks can be started dynamically through the task builder pattern. The `spawn()` method is available if the entry closure does not implement the `Clone` trait.

is sound. This expectation is similar to that of the popular FreeRTOS [5]; we consider it reasonable since all source code to be compiled is usually available to the system developer of a microcontroller-based embedded systems. Hopter's threat model is weaker than that of Tock [38] where application code can be malicious and the kernel must isolate itself from it using hardware-based protection (See §2) and suffers from its overhead.

3.2 Using Hopter Abstractions

Hopter provides three important abstractions for system developers to achieve memory safety, fault tolerance, and interrupt handling promptness. To implement application logic, a developer uses either the *restartable task* or the *fallible interrupt handler* context abstraction, which guarantees memory safety and allows recovery from fatal errors. Fallible interrupt handlers have zero-latency in response time to hardware events and can coordinate with tasks through provided *synchronization primitives*. Here we elaborate each of the three key abstractions: its benefits and how to use it. Their design and implement details will be presented in §4 and §5, respectively.

Restartable Tasks: The tasks running on Hopter are thread-based and scheduled preemptively. They improve applications' resilience against fatal errors through automatic restarting. Applications on Hopter declare restartable tasks with the builder pattern as shown in Listing 1, providing the entry closure and specifying the attributes. The main task is an exception that starts execution with the function marked with the provided `#[main]` attribute, which usually initializes the system and spawns other tasks. To enable automatic restarting, the application task needs only ensure that its entry closure implements the `Clone` trait and is started with the `spawn_restartable()` method. The compiler automatically implements the `Clone` trait for a closure if all of the enclosed variables are `Clone`. Upon fatal errors like stack

```
// Initialized to `Some(_)` after booting.
static TIMER: Spin<Option<CounterUs<TIM2>>>
    = Spin::new(None);

static MAILBOX: Mailbox = Mailbox::new();

// Invoked periodically by the timer IRQ.
#[handler(TIM2)]
fn tim2_handler() {
    TIMER.lock()
         .as_mut().unwrap() // Unwrap `Option`
         .wait().unwrap();  // Acknowledge IRQ
    // Resume the task.
    MAILBOX.notify_allow_isr();
}

// Spawned as a task.
fn another_task_main() {
    loop {
        MAILBOX.wait(); // Block and yield CPU
        // do something ...
    }
}
```

Listing 2: Synchronization between a handler and a task using the mailbox. The interrupt handler is declared through the `#[handler(...)]` macro. The handler has zero-latency response time while still being able to invoke synchronization primitives. Peripheral access is provided through a third-party HAL crate.

overflows, out-of-boundary array accesses, and failed assertions, restartable tasks terminate, release their resources, and automatically restart execution from beginning. If the entry closure is not `Clone`, the `spawn()` method is still available to run the task, but the task will only terminate with resources released upon fatal errors rather than restart.

Fallible Interrupt Handlers: The interrupt handlers on Hopter are functions that run to respond to hardware interrupts, sharing a single interrupt stack and always preempting tasks or lower-priority handlers. They improve system robustness by tolerating fatal errors during interrupt handling. Applications declare interrupt handler functions using the `#handler[IRQ_NAME]` attribute macro as shown in Listing 2. Handlers terminate execution upon fatal errors with resources released, and will rerun if the interrupt is still pending. Overflows of the interrupt stack is an exception that Hopter currently cannot recover from, due to the restrictions of dynamic memory allocation within interrupt context.

Synchronization Primitive: The synchronization primitives on Hopter are Rust `struct` types provided to applications to facilitate their coordination across contexts. They allow synchronization between interrupt handlers and tasks without compromising the zero-latency response time to interrupts. Application code interacts with synchronization primitives by calling defined methods. Listing 2 shows an example of the synchronization between a timer interrupt handler and a task using a mailbox. The hardware timer is accessed through a third-party HAL crate [27].

4 System Design

We next present the key design aspects of Hopter to realize memory safety, fault tolerance, and interrupt responsiveness. Hopter employs a compiler-based mechanism to guarantee memory safety for restartable tasks and fallible interrupt handlers. Notably, for stack memory safety, Hopter executes Rust code with finite-stack semantics (FS-semantics), facilitated by compile-time instrumented code (§4.1). FS-semantics unify runtime memory errors and other fatal errors as Rust panics, allowing Hopter to apply a universal recovery mechanism based on unwinding and restarting (§4.2). Moreover, Hopter overcomes expressiveness challenges associated with Rust, achieving zero-latency interrupt handling by adopting a novel mechanism called soft-locks (§4.3) in the kernel to support the implementation of synchronization primitives. Application logic remains agnostic to both the compile-time instrumentation and kernel internal implementations, allowing existing Rust libraries to be used unmodified.

4.1 Rust with Finite-stack Semantics

Hopter executes Rust code with *finite-stack* semantics (FS-semantics) to ensure stack memory safety and allow system recovery from stack overflows. FS-semantics associate each Rust thread of execution with an implicit stack size. Hopter forestalls any function call made with insufficient free stack space by raising a panic. In the exceptional case where an overflow occurs during the execution of a drop handler, Hopter temporarily extends the stack to finish the drop handler and raises the panic afterwards. Hopter takes three steps to achieve FS-semantics:

- Detect and forestall an imminent stack overflow. (§4.1.1)
- Raise a panic upon forestalled overflows in non-drop functions or after drop logic in drop handlers. (§4.1.2).
- Enable stack extension on physical memory. (§4.1.3)

4.1.1 Forestalling Stack Overflows. Hopter detects impending stack overflows by examining the free stack space before executing a function body. This is achieved by a prologue of instructions emitted by the compiler before allocating a stack frame inside the function body. The prologue computes the free stack size as the difference between the current stack top indicated by the stack pointer and the stack region boundary stored in a task-local variable `BOUNDARY`. Insufficient free space causes a trap to the kernel for further diagnosis. The execution of the function prologue requires at most 4 bytes of reserved stack space on Arm.

4.1.2 Initiating Stack Unwinding. Hopter raises panics to the tasks experiencing stack overflows. Panics initiate stack unwinding that reclaims the resources from the failed tasks to allow restarting them without leaked resources or deadlock.

In the common case, Hopter raises a panic immediately upon detecting an imminent overflow by the prologue (§4.1.1). After trapping, the kernel sets the program counter of the task to the stack unwinder entry to start the unwinding.

In the rare case where the stack overflows during the execution of a drop handler, Hopter extends the stack to finish running the drop logic before raising a panic. This is because stack unwinding must not start inside a drop handler, as doing so would skip some code responsible for releasing resources. More precisely, Hopter extends the stack and defers the panic if the function overflowing the stack meets either of the following two criteria: (1) Is a drop handler, or (2) Is called directly or indirectly by a drop handler.

Two separate mechanisms are required to check these criteria. The first criterion is addressed by the function prologue (§4.1.1), which passes to the kernel whether the current overflowing function is a drop handler. Checking for the second criterion requires drop handler instrumentation. Whenever a drop function starts executing, it sets the `IN_DROP` task-local variable to true. Nested drop handler invocations typically occur with nested struct types, so only the outermost drop function clears the flag before returning. Hopter can determine if the second criterion is met by observing the value of the `IN_DROP` flag. Note that the function prologue is still applied on top of the instrumentation for drop handlers.

The kernel defers raising the panic if either of the criteria holds upon overflow. It sets the `OVERFLOWED` task-local variable to true and extends the stack, rather than raising a panic immediately to the task. After finishing the drop logic, the outermost drop function will check the `OVERFLOWED` flag and raise the deferred panic if necessary.

Since any function can overflow the stack and initiate stack unwinding under FS-semantics, this conflicts with existing compiler optimizations. The LLVM compiler backend infers the `nounwind` attribute for functions and simplifies generated code based on it. LLVM marks a function as `nounwind` if it satisfies the following two conditions [39, 40]: (1) The function body contains no side-effect instruction. (2) The function makes calls to only `nounwind` functions or is a leaf function. LLVM then simplifies the code by omitting the landing pads and unwind table entries if a call is made to a `nounwind` function. However, if such a function call overflows the stack, the stack unwinding will fail, causing a hang or system reset. Therefore, Hopter disables compiler optimizations based on the inferred `nounwind` attribute for correctness.

4.1.3 Extending Stacks. Hopter leverages segmented stacks [32, 42, 64] to extend stacks on systems without virtual memory. A segmented stack is a linked list of non-contiguous memory chunks called stacklets. It dynamically allocates a new stacklet upon function calls if the remaining stack space is insufficient, and frees it upon function return.

The function prologue (§4.1.1) facilitates stack extension by providing the kernel with the requested stack frame size and stack-passed argument size. These two numbers are constants known to the compiler during compilation and are embedded in the instruction sequence as literals to avoid using extra registers. The kernel subsequently fetches the values by following the program counter prior to the trap and then allocates a stacklet with a size no less than the sum of the two numbers. If stack-passed arguments exist, the kernel will also copy them to the new stacklet so that they are adjacent to the callee's stack frame. Finally, the kernel resumes the task execution with the new stacklet.

It is important to note that the stack extension mechanism requires software-based overflow detection so that the new stacklet allocation always occurs at a predetermined place, allowing the code to pass parameters to the kernel. In contrast, hardware-based protection for physical memory typically detects the overflow after the function body has started execution and accessed an invalid memory address. Allocating a new stacklet and transferring execution to the new stacklet midway is very difficult, if possible at all.

4.2 Recovery from Panics

Hopter supports recovery from Rust panics in both tasks and interrupt handlers. A panic occurs due to an out-of-bounds array access, a failed assertion, or a stack overflow under FS-semantics. When a task or an interrupt handler panics, Hopter terminates its execution and reclaims allocated resources. The panic is isolated within the context where it is raised and does not prevent the system from continuing execution. If a task's entry closure implements the `Clone` trait, Hopter can automatically restart it upon a panic. Thus, interrupt handlers on Hopter are *fallible* and tasks *restartable*.

4.2.1 Unwinding Stacks. Hopter integrates a stack unwinder to reclaim resources upon panics, enabling graceful termination of tasks or interrupt handlers and allowing subsequent recovery. The unwinding procedure runs in the context of the panicked task or interrupt handler, during which the scheduler can continue to perform context switches and higher-priority interrupts can nest atop. Logically, the unwinder forces the immediate return of active functions when a panic occurs, starting from the top of the call stack and proceeding until the entry function of a task or interrupt handler returns. Local objects are dropped during this procedure. Mechanically, the unwinder refers to the unwind table generated by the compiler to restore registers and invoke small pieces of code called landing pads. Landing pads are categorized as either `cleanup` pads, which execute drop handlers, or `catch` pads, which terminate the unwinding procedure.

Hopter's stack unwinder differs from existing ones [33, 52, 61] in two ways to support segmented stacks. First, Hopter's unwinder avoids making divergent function calls that never return. Since the stack unwinder may be invoked to clean up a task experiencing a stack overflow, it must extend the stack to execute its own logic. If the unwinder made divergent calls, the stacklets used would not be freed, leading to potential memory leaks. In contrast, prior implementations typically make divergent calls to landing pads. Second, Hopter's unwinder recognizes stacklet boundaries and frees stacklets during unwinding to avoid memory leak. This is important because a task's stack may contain multiple stacklets, e.g., when an overflowing drop handler raises a panic while the code is running on an extended stacklet.

4.2.2 Restarting Applications. With resources reclaimed by the unwinder, Hopter can recover a panicked task by restarting it from its entry closure. To enable restarting, Hopter requires the task's entry closure to implement the `Clone` trait so that the kernel can safely duplicate the closure's enclosed environment. For interrupt handlers, Hopter performs an exception return instead of re-executing the handler after catching the panic. If the interrupt request is still pending, the handler will automatically be invoked again.

To speed up recovery from task panics, Hopter implements the *concurrent task restart* optimization proposed by [41]. This technique allows for the immediate execution of a new instance of the panicked task, running concurrently with the unwinding procedure of the previous instance, thereby ensuring minimal unresponsive time. Hopter reduces the task priority of the panicked instance to the lowest, so that unwinding uses otherwise idle CPU time. Rust's ownership model eliminates race conditions between the restarted instance and the one being unwound. Application programmers may opt out of concurrent restart to prevent the restarted task from observing logically inconsistent data. In this case, Hopter performs a *context reuse* optimization, reusing the task structure of the unwound task for the restarted instance, thereby saving the allocation and initialization of a new task structure.

4.3 Zero-latency Interrupt Handling

Hopter supports zero-latency interrupt handling, wherein the system always invokes the associated handler function immediately upon receiving an interrupt. This is achieved by never disabling interrupts within the kernel. To prevent race conditions between the interrupt handler and the preempted context, Hopter adopts a novel mechanism called *soft-locks* which are amenable to Rust's compile-time check. These locks protect kernel data structures by serializing mutations from concurrent contexts, allowing all interrupt handlers in Hopter to interact with kernel objects, such as notifying tasks through synchronization primitives.

4.3.1 Algorithm Description. Soft-locks eliminate race conditions caused by concurrent access from interrupt handlers. Acquiring a soft-lock yields either *full access* or *partial access* to the protected data structure. Code gets full access if the soft-lock is not already acquired or otherwise partial access. Full access allows mutations to all data structure fields, while partial access permits modifications to only a subset of fields.

An interrupt handler receives partial access to record its intended operations when a preempted context holds the full access. These operations are deferred to be executed after the handler returns and once the code with full access completes its own operation. To further prevent race conditions arising from concurrent task execution, the scheduler is always suspended before acquiring a soft-lock, thus code running within task contexts always receives full access. Since interrupt handlers always preempt tasks and are strictly prioritized when nesting, code with full access resumes execution after the code with partial access finishes, therefore allowing deferred operations to execute immediately before releasing the full access.

Soft-locks encapsulate the protected data structure and maintain two atomic boolean flags to track contention states and whether any deferred operations are pending. Listing 3 shows the pseudo-code for acquiring and releasing soft-locks. The `locked` flag indicates whether the data structure is under contention, and the `pend` flag indicates if there are deferred operations. The protected data structure must implement the `AllowDeferredOp` trait to specify which fields are accessible through the full or partial accessor and the code for running deferred operations. Upon releasing a soft-lock, pending deferred operations are checked and executed if necessary using the scoped full accessor, which makes it amenable to compile-time checks. In the actual implementation, soft-locks are released automatically through the drop functions of access guards instead of manually.

4.3.2 Use Cases in Kernel. Soft-locks protect four data structure types in Hopter's kernel. Table 1 lists these data structures, along with the operations under full or partial access and the deferred operations. The wait queue provides the foundation to implement synchronization primitives like mutexes, condition variables, semaphores, and channels. The ready queue maintains ready tasks for the scheduler. The mailbox is a lightweight synchronization primitive that allows tasks to wait for a notification, optionally with a timeout. The sleep queue stores sleeping tasks waiting for virtual timers to expire. An interrupt handler may notify a task by removing it from the wait queue or mailbox, adding it to the

Table 1: Kernel data structures protected by soft-locks. Code gets full access to the data structure to perform the listed operations if the soft-lock is not under contention. Code running in task context always gets full access. An interrupt handler gets partial access if the data structure is being accessed in the preempted context, where it can access only the underlined fields to record its intended operation. Code that is granted full access will run the deferred operation before releasing the soft-lock.

Data Structure	Fields	Full Access	Partial Access	Deferred Operation
Wait Queue	linked list L counter x	Add a task to L Remove the highest priority task from L	Increment x	Remove x high-priority tasks from L and clear x
Ready Queue	linked list L lock-free buffer I	Add a task to L Remove the highest priority task from L	Insert a task to I	Add tasks in I to L and clear I
Mailbox	task placeholder H counter x	Set H to hold a task or decrement x Take the task from H or increment x	Increment x	Clear H and decrement x if H is not empty
Sleep Queue	linked list L lock-free buffer D	Add a task to L Remove expired tasks from L	Insert a task to D	Remove expired tasks and tasks in D from L, then clear D

```
struct SoftLock<T> where T: AllowDeferredOp {
    locked: AtomicBool, pend: AtomicBool, ...
}

fn acquire(sl: &SoftLock<T>) -> Access {
    let prev_locked = sl.locked.swap(true);
    if prev_locked { return Access::Partial; }
    else { return Access::Full; }
}

fn release(sl: &SoftLock<T>, acc: Access) {
    match acc {
        Access::Partial => { sl.pend = true; }
        Access::Full => { loop {
            let prev_pend = sl.pend.swap(false);
            if prev_pend { acc.deferred_op(); }
            sl.locked = false;
            if !sl.pend { break; }
            else { sl.locked = true; }
        }}
    }
}
```

Listing 3: Pseudo-code for the `acquire` and `release` operation on soft-locks. Protected data structures must implement the `AllowDeferredOp` trait, which specifies what fields are accessible through the `Full` or `Partial` accessor and what code to execute when running `deferred_op()`. Soft-locks return the `Full` accessor under no contention or otherwise `Partial`. Deferred operations are executed when releasing with the `Full` accessor. The loop aims to avoid missing deferred operations that come after the first check of the `pend` flag.

ready queue, and optionally deleting it from the sleep queue. When any data structure is under contention, the handler uses partial access to record its intended operations, which are deferred to be executed later.

5 Implementation

We implemented Hopter for Arm Cortex-M4-based microcontrollers. Supporting a new board requires only defining an interrupt vector array, as Hopter leverages existing HALs for peripheral access. The implementation consists of two parts: compiler modifications and Rust code for the library crate. Additionally, we implement Hopter's features in a hierarchy to allow application programmers to trade-off between the benefits and associated overheads.

Compiler Modification. We modified both the rustc compiler frontend and the LLVM backend to support FS-semantics, and we have open-sourced the patches. To generate the function prologue (§4.1.1), we added the `split-stack` attribute to all functions during the intermediate representation (IR) emission stage in rustc. This attribute triggers a prologue adjustment step in LLVM's frame lowering stage, which we further modified to produce our desired prologue. To instrument drop handlers (§4.1.2), we modified the drop elaboration step in the mid-level IR (MIR) stage of rustc. In total, we added 260 lines of code to the frontend and 170 lines to the backend. We also removed 30 and 110 lines from the frontend and backend, respectively, to prevent optimizations based on the `nounwind` attribute (§4.1). Finally, we modified 50 lines in Rust's `core` library to strip away unused debug information fields in `PanicInfo`, which can reduce the compiled binary size by a few kilobytes.

Library Crate. We implemented Hopter as a Rust library crate consisting of approximately 12,200 lines of Rust code, of which fewer than 1,000 are unsafe Rust. Hopter resorts to unsafe Rust code only when certain functionalities cannot be expressed in safe Rust, such as stack extension and dynamic memory management. The unsafe code also includes inline assembly for direct register access, bootstrap and interrupt handler entry points, and primitive memory operations like `memset`. We also implemented two auxiliary library crates: one to provide Hopter with the `#[main]` and `#[handler]` attribute macros, and the other to allow applications to configure numeric kernel parameters. These consist of approximately 1,000 and 40 lines of safe Rust, respectively.

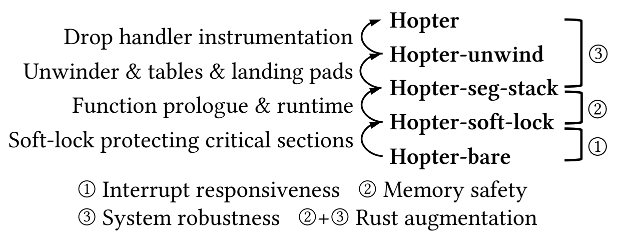
Drop handler instrumentation
Unwinder & tables & landing pads
Function prologue & runtime
Soft-lock protecting critical sections
Hopter
Hopter-unwind
Hopter-seg-stack
Hopter-soft-lock
Hopter-bare
③
②
①
① Interrupt responsiveness ② Memory safety
③ System robustness ②+③ Rust augmentation

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.

In addition, Hopter depends on open-source Rust library crates for implementing spin locks [63], intrusive data structures [22], lock-free data structures [21], and accessing CPU core peripherals [30]. Although these crates contain unsafe code within their provided safe abstractions, they have been widely used and scrutinized by the Rust community with at least millions of downloads of each.

Feature Hierarchy. We made each Hopter's feature optional and built a hierarchy. The hierarchy starts from features with low overhead but essential for system responsiveness and memory safety, extending towards features for enhancing system robustness with higher overhead, as shown in Figure 1. The hierarchy starts from the "bare" version that includes none of the features. Continuing up, 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 the 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.

6 Evaluation

Hopter aims to provide system memory safety, robustness, and responsiveness, while remaining transparent to application logic. Because memory safety is achieved by construction, this section quantifies robustness and responsiveness, as well as the overhead brought by the unique design choices of Hopter. Specifically, we seek to answer the questions below:

- (Q1 Overhead): What is the overhead introduced by Hopter in order to achieve its objectives?
- (Q2 Size): Is Hopter light enough to support microcontrollers with small flash and SRAM?
- (Q3 Responsiveness): Can Hopter support time-sensitive tasks and interrupt handling?
- (Q4 Robustness): Can Hopter recover from more fatal errors than other OSes and fast enough for mission-critical applications?

6.1 Methods

We employ three orthogonal methods to answer the above questions.

First, to quantify the impact by each of the features of Hopter, we follow the hierarchy in Figure 1 and measure the statistics for each listed variant.

Second, we compare Hopter with two well-known embedded OSes: FreeRTOS [5], the most widely used embedded OS, and Tock [38], the state-of-the-art Rust-based embedded OS. We develop a set of micro-benchmarks with controlled workload for all three OSes. FreeRTOS employs no mechanisms to provide memory safety or robustness to application errors, and is designed to support high-frequency workloads and to be light-weight. Tock assumes application code can be malicious and employs hardware-based protection.

Finally, in order to assess Hopter's performance and overhead under a realistic application and demonstrate its ability to recover from errors under mission critical scenarios, we develop a flight control program using Hopter for the Crazyflie [10] miniature drone as a macro-benchmark.

6.2 Controlled Workload

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. First, we build an LED blinking application to show the minimum resource required when developing with an OS. Next, we quantify system responsiveness by measuring the latency of the task context switch and response to interrupts. Finally, we demonstrate that Hopter can recover from more complex scenarios than other OSes with a serial server task.

6.2.1 Minimum System (Q1/Q2). To measure the minimum hardware resources required to run a system developed using the three OSes under comparison, we build a blinking LED application, the "hello world" program for the embedded world. We assume an application programmer's role, i.e., using the OS and hardware abstraction layer (HAL) library as-is through their public APIs, without modifying their internal implementation. We use the HAL library from `stm32-rs` [27] and STM32CubeF4 [58] for Hopter and FreeRTOS, respectively. Tock comes with its own HAL.

The minimum system columns in Table 2 list the flash and SRAM size. The flash overhead of Hopter mainly comes

Table 2: 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.

	Minimum System		Latency (µs)			Feature		
	Flash (KiB)	†SRAM (KiB)	* Task	** Interrupt	** Interrupt-Task	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	**27.68**	**1.00**	**16.0**	**1.36 (0.01)**	**31.88 (2.44)**	✓	✓	✓
Hopter-unwind	22.71	1.00	12.9	1.32 (0.03)	26.44 (1.96)	✓	✓	✗
Hopter-seg-stack	12.60	0.84	13.0	1.20 (0.01)	27.24 (1.90)	✓	✓	✗
Hopter-soft-lock	10.05	0.75	12.6	1.16 (0.02)	25.48 (1.84)	✓	✗	✗
Hopter-bare	9.12	0.49	15.5	5.26 (1.21)	28.30 (1.83)	✗	✗	✗

†: 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.

from the stack unwinder logic, the unwind table, the landing pads, and the drop handler instrumentation. A minor overhead comes from the function prologue and stack extension runtime. The flash overhead can be amortized with a more sophisticated application, like the flight control program (§6.3), reducing the overhead to 56%. We also provide a more detailed breakdown of the flash overhead for the flight control program.

The minimum hardware resources required to run Hopter without robustness features, i.e., the "bare", "soft-lock", or "seg-stack" variants, are similar to those of FreeRTOS. The support for stack unwinding and the drop handler instrumentation incur noticeable overhead particularly in flash usage, but the numbers are still within the lower tens of KiB. On the other hand, systems developed with Tock requires significantly larger flash and SRAM as shown in Table 2, because Tock's kernel is compiled separately from the application. The compiler toolchain is unable to remove unused kernel code at compile time, resulting in the size bloat.

6.2.2 Task Scheduling and Interrupt Handling (Q1/Q3). 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. Hopter's latency is on the same order of magnitude as FreeRTOS's, allowing it to support multiple cooperating tasks with frequencies up to 1,000 Hz (§6.3). 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 compute context switch latency 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 column in Table 2 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 and bookkeeping the current running task. The optimized organization and operations of task list found in FreeRTOS are incompatible with Rust's ownership model (See §2). However, an 8 µs overhead in context switch results in less than 1% CPU load increase for a task running at 1,000 Hz.

We also measure the interrupt response latency, which determines if the system may miss an ephemeral event. The board under measurement registers an interrupt handler triggered by rising edges on a general purpose input/output (GPIO) pin. The board responds by setting another GPIO pin to high. We use another board to trigger the rising edge and time the response latency precise to 0.02 µs.

The interrupt column in Table 2 reports the maximum latency observed in 10,000 tests where the interrupt handler directly responds to the interrupt, while the interrupt-task column lists the numbers when a high-priority task responds to the interrupt notified by the handler. Since Tock does not support declaring an application interrupt handler, we instead register the handler as a capsule in the kernel space. The latency 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 responding in the interrupt handler, benefiting from the soft-lock mechanism. When responding 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 2. The slight increase in the

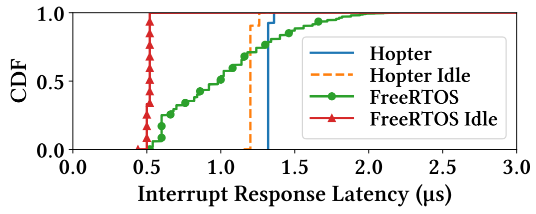

Figure 2: Hopter's interrupt response latency is almost constant using soft-lock. Background workload causes slight increase in latency due to the stress on instruction and literal cache. FreeRTOS is sensitive to background workload, because it masks interrupts inside its critical sections.

number under load is due to the eviction of cached instructions and literals in the ART accelerator for flash [56]. 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.

6.2.3 Fatal Error Recovery (Q4). 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 manifests 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.

6.3 Flying a Drone

As a macro-benchmark, we develop a flight control program with Hopter to measure its overhead in resource usage and CPU load with a realistic application, and to quantify the latency of interrupt response and task recovery demonstrating its suitability for real-time workload. The program runs

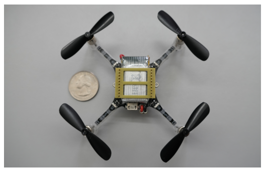

Figure 3: Crazyflie 2.1, a commercial off-the-shelf miniature drone. We implement a flight control program with Hopter to evaluate its capability of supporting real-time applications.

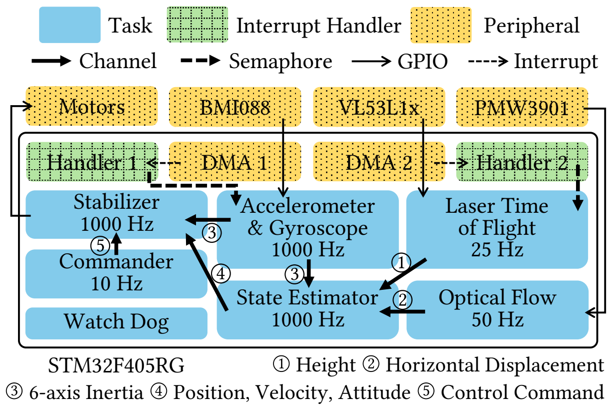

Figure 4: The flight control program consists of interrupt handlers and periodical tasks running at high frequency. Tasks and handlers synchronize through channels and semaphores. Peripherals are connected via GPIO pins.

with the commercially available Crazyflie 2.1 hardware platform [10] as shown in Figure 3. The drone is 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. About 6,700 lines are peripheral driver code manually translated from open-source libraries [13, 50, 51]. We use the HAL library from `stm32-rs` [27] 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 4. Three tasks are responsible for reading data from the inertial sensors, the height sensor, and the optical flow sensor, respectively. Two of them read through direct memory access (DMA). 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

Table 3: 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 set to hover.

	Flash (KiB)	SRAM (KiB)	CPU
FreeRTOS	133.56	29.10	36.5%
Hopter	**213.53**	**7.51**	**52.3%**
Hopter-unwind	173.43	7.51	48.8%
Hopter-seg-stack	147.80	7.35	47.5%
Hopter-soft-lock	141.90	7.00	45.5%
Hopter-bare	136.95	6.52	48.4%

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.

6.3.1 System Size and CPU Load (Q1/Q2/Q3). Table 3 shows the flash and SRAM usage of the flight control program and the CPU load when the drone is set to hover. We break down Hopter's flash overhead as follows, where underlined numbers are constant while others grow with the binary size. The function prologue for stack memory safety and the stack extension runtime each incur a flash overhead of 3.14 KiB and 2.75 KiB, respectively. To enable resource reclamation through unwinding, the stack unwinder adds 9.59 KiB, the landing pads add 7.50 KiB, and the unwind table adds 8.53 KiB. To allow recovery from stack overflow, instrumenting the drop handlers increases the code size by 27.80 KiB and the unwind table by 4.70 KiB. Disabling compiler optimizations based on the `nounwind` function attribute increases the code size and unwind table size by another 2.97 KiB and 4.64 KiB, respectively.

We also compare against a FreeRTOS-based implementation with similar program structure. FreeRTOS's version requires similar flash storage as with Hopter's "bare" variant. Its higher SRAM usage is due to the use of larger static buffers, and its lower CPU load is due to the availability of highly optimized math library and the DMA driver implementation for the SPI bus to access the optical flow sensor.

Hopter's drop handler instrumentation (§4.1.2) incurs the largest overhead in 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 CPU load increase. (1) If the type of any field inside a compound type implements `Drop`, the drop handler of both the inner and outer type will be instrumented. (2) ARMv7, as a RISC instruction set, must load the address of global variables into registers before accessing them, resulting in longer instruction sequences. (3) There is more register spilling, further increasing the instruction sequence length. (4) Larger function body causes the compiler not to inline some drop handler functions, resulting in more function calls and returns. (5) An outlined function in turn includes additional segmented stack prologue instructions.

Table 4: Recovery latency and stack unwinder workload for stabilizer task upon fatal errors. 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.

	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

However, drop handler instrumentation is indispensable for correct recovery from stack overflow despite the overhead. We perform a static stack depth analysis for the seven flight control tasks, ignoring indirect function calls through function pointers or trait objects (0.1% of all calls). 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.

6.3.2 In-flight Interrupt Response Latency (Q3). We measure the interrupt response latency while setting the drone to hover, following the same experimental setup as in §6.2.2. With soft-lock, the maximum interrupt response latency over 10,000 measurements is 1.04 µs with a standard deviation of 0.02 µs. However, without soft-lock, the maximum latency rises to 7.56 µs with a standard deviation of 0.57 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 to support the radio on the drone. 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 [11] used by the drone, the packet length is determined by the first four bytes, so DMA can only be initiated after these bytes are received. Without DMA, the system must respond immediately to the interrupt after the UART interface 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 [57], resulting in data loss. Only with soft-locks can the radio module function correctly.

6.3.3 In-flight Fatal Error Tolerance (Q4). We deliberately introduce panics 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 stabilizer task is the most sensitive one to fatal errors because it directly modulates the power of motors.

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 4 lists the latency as well as 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. Stack extension incurs an additional 20 µs delay.

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 task instance not having any command to follow. The motors are kept at zero power until receiving the next command.

7 Related Work

Memory Safety with Rust: Similar to Hopter's use of Rust, other OSes have leveraged Rust for memory safety across both the kernel and applications. Two recent works targeting PCs and servers, Theseus [12] and RedLeaf [48], utilize Rust to achieve memory safety, yet they still depend on the memory management unit (MMU) to place a guard page to prevent stack overflows. RIOT [9], Embassy [15], and RTIC [16] embedded OS, as well as the kernel space of Tock [38], employ Rust for memory safety on microcontrollers. RIOT supports threaded tasks written in Rust, but similarly uses MPU/PMP to set a guard region to prevent stack overflows. This approach will not allow recovery from stack overflows, as discussed in §4.1.3. Embassy, RTIC, and Tock's kernel space employs a single down-growing stack placed at the lower boundary of the SRAM region to prevent stack overflows from corrupting memory. However, they support only restricted scheduling patterns and cannot recover from stack overflows. In contrast, Hopter achieves memory safety solely via software, supports arbitrary scheduling patterns with threaded tasks, and can recover from stack overflows.

Fatal Error Recovery: Related to Hopter's capability of recovering from fatal errors, few other embedded OSes targeting microcontrollers provide recovery mechanisms. Tock [38] can automatically restart a failed task due to memory access violations, while Zephyr terminates a problematic task and requires applications to provide further recovery logic. Neither supports running application-supplied cleanup logic during task termination. In contrast, Hopter runs the application cleanup logic (drop handlers), allowing the system to avoid subsequent errors such as deadlocks.

Zero-latency Interrupt Handling: Hopter employs soft-locks to achieve zero-latency interrupt handling, but previous embedded OSes, such as eCos [44], PEASE [55], PURE [54], SMX [45], and Symbian [49], follow a different approach. These systems split interrupt handling into top and bottom halves to avoid masking interrupts in the kernel. Only the top half is directly invoked by hardware interrupts, while interactions to kernel objects are deferred to the bottom half executed when the kernel can guarantee an absence of race conditions. This split design requires unsafe Rust because it requires a global serialization queue to store the deferred operations (bottom halves) that contain references to kernel objects. The compiler may fail to statically verify the lifetime of the references. Also, Hopter's soft-lock can deliver better performance, because it defers operations only upon contention, which is relatively rare, thereby reducing CPU load by mostly taking the fast path.

8 Conclusion

This paper describes Hopter, a Rust-based embedded OS designed for microcontrollers. Hopter transparently provides applications with memory safety, system robustness, and interrupt responsiveness. Hopter forestalls stack overflows and converts such errors into Rust panics by executing Rust code under FS-semantics using compile-time code instrumentation and runtime OS support. By unifying all fatal errors as panics, Hopter performs stack unwinding to reclaim resources from failed application tasks or interrupt handlers, and can automatically restart failed tasks. To ensure timely response to interrupts, Hopter incorporates a novel mechanism called soft-locks to achieve zero-latency interrupt handling. Hopter is well-suited for resource-constrained microcontrollers and supports error recovery for real-time workloads.

References

[1] [n. d.]. The Embedded Rust Book. https://docs.rust-embedded.org/book/. Accessed: 2024-10-02.

[2] [n. d.]. Redox - Your Next(gen) OS. https://www.redox-os.org/. Accessed: 2024-10-02.

[3] [n. d.]. Rewriting Binder Driver in Rust. https://lore.kernel.org/rust-for-linux/20231101-rust-binder-v1-0-08ba9197f637@google.com/. Accessed: 2024-10-02.

[4] [n. d.]. Windows Drivers in Rust. https://github.com/microsoft/windows-drivers-rs. Accessed: 2024-10-02.

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

[6] Arm Limited. 2021. ARMv7-M Architecture Reference Manual (Issue E.e). https://developer.arm.com/documentation/ddi0403/ee/. Chapter B3.5: Protected Memory System Architecture, PMSAv7.

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

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

[9] Emmanuel Baccelli, Cenk Gündoğan, Oliver Hahm, Peter Kietzmann, Martine S Lenders, Hauke Petersen, Kaspar Schleiser, Thomas C Schmidt, and Matthias Wählisch. 2018. RIOT: An open source operating system for low-end embedded devices in the IoT. *IEEE Internet of Things Journal* 5, 6 (2018), 4428–4440.

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

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

[12] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. 2020. Theseus: an experiment in operating system structure and state management. In *Proc. USENIX OSDI.*

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

[14] Ching-Han Chen, Ming-Yi Lin, and Chung-Chi Liu. 2018. Edge computing gateway of the industrial internet of things using multiple collaborative microcontrollers. *IEEE Network* 32, 1 (2018), 24–32.

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

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

[17] Jonathan Corbet. 2022. A first look at Rust in the 6.1 kernel. https://lwn.net/Articles/910762/. *Linux Weekly News* (10 2022).

[18] 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.

[19] 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.

[20] 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.

[21] Alex Crichton, Jeehoon Kang, Aaron Turon, and Taiki Endo. 2024. Tools for concurrent programming. https://crates.io/crates/crossbeam. Version 0.8.4.

[22] Amanieu d'Antras. 2024. Intrusive collections for Rust (linked list and red-black tree). https://crates.io/crates/intrusive-collections. Version 0.9.7.

[23] 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.

[24] 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.

[25] 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.

[26] Guo Dong, Wang Hongpei, Gao Song, and Wang Jing. 2011. Study on adaptive front-lighting system of automobile based on microcontroller. In *Proceedings 2011 International Conference on Transportation, Mechanical, and Electrical Engineering (TMEE).* IEEE, 1281–1284.

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

[28] Bill Fleming. 2011. Microcontroller units in automobiles [automotive electronics]. *IEEE Vehicular Technology Magazine* 6, 3 (2011), 4–8.

[29] 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.

[30] Adam Greig. 2023. Low level access to Cortex-M processors. https://crates.io/crates/cortex-m. Version 0.7.7.

[31] Mehedi Hasan, Maruf Hossain Anik, and Sharnali Islam. 2018. Microcontroller based smart home system with enhanced appliance switching capacity. In *2018 Fifth HCT Information Technology Trends (ITT).* IEEE, 364–367.

[32] Robert Hieb, R Kent Dybvig, and Carl Bruggeman. 1990. Representing control in the presence of first-class continuations. *ACM SIGPLAN Notices* (1990).

[33] Free Software Foundation Inc. 2023. Exception handling and frame unwind runtime interface routines. https://github.com/gcc-mirror/gcc/blob/721cdcd1ddde0738deb08895e113a8db84187a14/libgcc/unwind.inc.

[34] Fortune Business Insights. 2024. Embedded Systems Market Size, Share & COVID-19 Impact Analysis, By Component, By Type, By Application, and Regional Forecast, 2023-2030. https://www.fortunebusinessinsights.com/embedded-systems-market-108767.

[35] Future Market Insights. 2024. Embedded System Market Analysis from 2024 to 2034. https://www.futuremarketinsights.com/reports/embedded-system-market.

[36] Global Market Insights. 2024. Embedded Systems Market Size - By Component, By Application, By Function & Forecast, 2024 - 2032. https://www.gminsights.com/industry-analysis/embedded-system-market.

[37] Prateek Khurana, Rajat Arora, and Manoj Kr Khurana. 2014. Microcontroller based implementation of Electronic Stability Control for automobiles. In *2014 International Conference on Advances in Engineering & Technology Research (ICAETR-2014).* IEEE, 1–5.

[38] 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.*

[39] LLVM. 2024. AttributorAttributes.cpp - Attributes for Attributor deduction. https://llvm.org/doxygen/AttributorAttributes_8cpp_source.html.

[40] LLVM. 2024. FunctionAttrs.cpp - Pass which marks functions attributes. https://llvm.org/doxygen/FunctionAttrs_8cpp_source.html.

[41] Zhiyao Ma, Guojun Chen, and Lin Zhong. 2023. Panic Recovery in Rust-based Embedded Systems. In *Proc. ACM PLOS.*

[42] Zhiyao Ma and Lin Zhong. [n. d.]. Bringing Segmented Stacks to Embedded Systems. In *Proc. ACM HotMobile.*

[43] Aleksander Malinowski and Hao Yu. 2011. Comparison of embedded system design for industrial applications. *IEEE transactions on*

industrial informatics 7, 2 (2011), 244–254.

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

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

[46] Mohamed Abd El-Latif Mowad, Ahmed Fathy, Ahmed Hafez, et al. 2014. Smart home automated control system using android application and microcontroller. *International Journal of Scientific & Engineering Research* 5, 5 (2014), 935–939.

[47] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. 2020. RedLeaf: Isolation and Communication in a Safe Operating System. In *Proc. USENIX OSDI*.

[48] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. 2020. RedLeaf: Isolation and communication in a safe operating system. In *Proc. USENIX OSDI*.

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

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

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

[52] LLVM Project. 2023. Implementation of C++ ABI Exception Handling Level 1. https://github.com/llvm/llvm-project/blob/f42482def236999b0f7896c09cd714b708861c8b/libunwind/src/UnwindLevel1.c.

[53] RISC-V. 2024. The RISC-V Instruction Set Manual: Volume II (Version 20240411). https://riscv.org/technical/specifications/. Chapter 3.7: Physical Memory Protection.

[54] 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.

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

[56] 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).

[57] 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.

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

[59] Andrew Straw and Richard Meadows. 2024. Hardware Abstraction Layer implementation for STM32H7 series microcontrollers. https://crates.io/crates/stm32h7xx-hal. Version 0.16.0.

[60] Rust Tools Team. 2021. svd2rust. https://github.com/rust-embedded/svd2rust

[61] Theseus OS. 2023. Support for unwinding the call stack and cleaning up stack frames. https://github.com/theseus-os/Theseus/blob/562a39cf6c662738f7718473f6bbc010970dce53/kernel/unwind/src/lib.rs.

[62] Vishakha D Vaidya and Pinki Vishwakarma. 2018. A comparative analysis on smart home system to control, monitor and secure home, based on technologies like gsm, iot, bluetooth and pic microcontroller with zigbee modulation. In *2018 International Conference on Smart City and Emerging Technology (ICSCET)*. IEEE, 1–4.

[63] Mathijs van de Nes and Joshua Barretto. 2023. Spin-based synchronization primitives. https://crates.io/crates/spin. Version 0.9.8.

[64] Rob Von Behren, Jeremy Condit, Feng Zhou, George C Necula, and Eric Brewer. 2003. Capriccio: scalable threads for internet services. *ACM SIGOPS Operating Systems Review* (2003).