

Enhancing Memory Safety by Integrating Rust in the μ 111 Real-time Operating System

M. I. Ben Salah, P. Liechti, D. Keller

As embedded systems are growing in complexity, maintaining their firmware correctness (memory-leak free, out of bounds access) and application performance requires appropriate development tools. The CSEM ultra-low power RTOS now supports Rust, which natively provides runtime mechanisms for ensuring memory safety and compilation time checks, with a minimal impact on in terms of memory consumption and runtime execution.

The Rust^[1] language can be used for developing software components with stricter memory management in combination with μ 111, benefiting from the properties of the Rust language in term of memory safety, while keeping the benefits of μ 111 in terms of ultra-low power and development flexibility. It has been developed and evaluated on the NUCLEO-L4R5ZI based on the STM32L4R5 (Cortex-M4, 2 MB of Flash & 640 Kbytes of RAM).

Rust aims at preventing, at compile time, most data races and memory-related bugs by annotating the source code to statically prove that memory is not accessed after it is no longer valid. This is done by using the concepts of 1) memory ownership where the compiler manages the lifetime and the access scope of variables, indicating an error whenever the code tries to access an unreachable, freed, or expired variable; 2) references which are read-only by default, but whenever a mutable reference is created, are automatically invalidated, thus avoiding race conditions. Rust also provides the concept of safe/unsafe code: safe code uses high-level abstractions, i.e., references, where Rust guarantees can be verified, unsafe code is used for memory operations which validity cannot be checked by the compiler such as direct memory access with pointers, C/C++ libraries calls (using Foreign Function Interface) and accessing Memory-Mapped I/O.

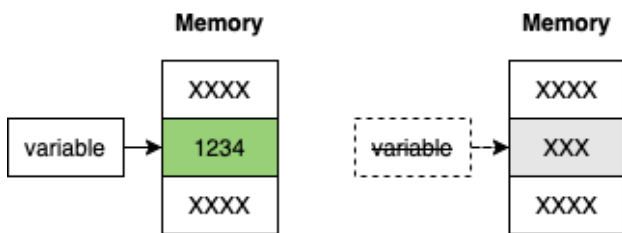


Figure 1: Rust scoping mechanism: when the variable is out of scope, the memory is automatically erased, and its location gets inaccessible.

```
fn safe_fill(array: &mut [i32]){
    for i in 0..10 {
        array[i] = i;
    }
}
unsafe fn unsafe_fill(pointer: *mut i32) {
    for i in 0..10 {
        unsafe { *pointer.add(i) = i };
    }
}
```

Figure 2: Safe and unsafe version of accessing an array in Rust: safe uses a reference, where the unsafe version uses pointers.

To compile Rust source code as easily as existing C code in μ 111, we created a Rust interface, which maps all the kernel calls in C, with the tooling needed to generate the correct compilation flags and integrate into existing makefiles.

The interfacing library exposes every μ 111 kernel function and the appropriate macros to Rust, which enable native multi-processing execution of Rust code, benefitting from both the kernel and the ecosystem of Rust packages. Rust also comes with many programming facilities such as pattern matching, iterators, etc. The Rust mechanisms introduce an overhead in terms of performance and memory consumption, which was evaluated using a reimplementation of a benchmark tool on μ 111, with fill array operation, X/Y projections and histogram computations. Figure 3 shows the impact in terms of runtime execution (max. 300% increase) and code size (max. 485% increase) caused by the additional runtime checks performed to ensure correct execution (memory ownership, out-of-bounds array access). However, unsafe Rust closely matches the original C performance. In addition, the optimized safe Rust implementation of the fill array operation is even faster than the original C code.

	Fill the array		X projection		Y projection		Histogram	
	μ s	bytes	μ s	bytes	μ s	bytes	μ s	Bytes
C	135	54	216	64	161	68	239	66
Safe Rust	100 - 539	316	212	288	165	208	301	158
Unsafe Rust	130	52	211	150	152	72	238	100

Figure 3: C/Rust comparison of μ 111 bench tool.

One of the foreseen use cases of Rust on embedded platforms is the implementation of cryptography algorithms, where correctness is an absolute necessity and correct usage of the memory (avoiding buffer overflows) can be facilitated by Rust. This use case was chosen as a validation vector. Running the ASCON (Lightweight Authenticated Encryption & Hashing algorithms) implementations in C and Rust on standard test vectors resulted in an overhead of execution time of 10%, with a slightly smaller code footprint (C: 4.81 KB, Rust: 3.05 KB).

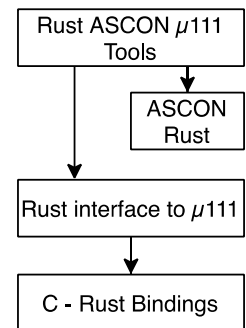


Figure 4: μ 111 Rust structure with ASCON tool.

The native Rust support provided by CSEM covers the complete μ 111 kernel API, which allows for its straightforward integration inside the μ 111 compilation workflow, as shown by the 32 μ 111 sample applications ported from C to Rust, reinforced by an in-depth tutorial for Rust usage in embedded systems and μ 111. It also enables the use of external Rust libraries. Rust shows a great potential in embedded systems by providing significant robustness improvements with acceptable levels of overheads. With the support of Rust, the development of memory-checked secure IP, such as a Virtual Secure Element isolated in a Trusted Execution Environment, is greatly facilitated.

[1] <https://www.rust-lang.org>