Departamento de Electrónica, Sistemas e Informática
Especialidad en Sistemas Embebidos

Improving firmware development through the Rust Programming Language

TRABAJO DE OBTENCIÓN DE GRADO que para obtener el **GRADO** de
ESPECIALISTA EN SISTEMAS EMBEBIDOS

Presenta: **MIGUEL PÉREZ GARCÍA**

Asesor **DIEGO ANTONIO MEJÍA SÁNCHEZ**

Tlaquepaque, Jalisco. agosto de 2021.

Improving firmware development through the Rust Programming Language

MIGUEL PÉREZ GARCÍA and DIEGO ANTONIO MEJÍA SÁNCHEZ, Instituto Tecnológico y de Estudios Superiores de Occidente, México

ANSI-C and C++ have historically been the go-to option when developing embedded software but both languages lack memory security features which have been proven to be problematic by security standards. The Rust Programming Language has been pushed as a progressive alternative when security is a concern in firmware development and it has shown its viability by the community and the industry.

CCS Concepts: • **Computer systems organization** → *Embedded software.*

Additional Key Words and Phrases: embedded software, firmware, architecture, programming languages

ACM Reference Format:
Miguel Pérez García and Diego Antonio Mejía Sánchez. 2021. Improving firmware development through the Rust Programming Language. 1, 1 (August 2021), 10 pages.

1 INTRODUCTION

Security for mission-critical embedded software is challenging due to emerging threats such as software level attacks. Moreover, software complexity is also increasing, which makes software development more prone to bugs, even in professional scenarios where code is peer reviewed and quality assurance or certifications are required [16]. Historically, this type of systems have been developed using ANSI-C or C++, even though other programming languages have emerged as alternatives, such as Lua, Python or Java, but trying to take advantage of their benefits, other drawbacks have to be considered, and thus they have not been adopted in the embedded systems realm [14].

Another aspect of embedded software development is that the adoption of other languages or platforms needs to be aligned with the characteristics required by the embedded system, such as real-time response, small memory footprint, and power consumption, among others [20]. The C and C++ languages provide great benefits for this purpose since they are both strongly typed, memory is manually managed, and they are statically checked [15]. However, the manual managed memory has proven to be the source of at least 70% of all security vulnerabilities as reported by Microsoft [17]. Furthermore, in the Common Vulnerabilities and Exposures (CVE) database, 3826 records were related to embedded systems and security vulnerabilities, which were caused by programming errors that lead to control flow attacks, such as buffer overflows or dangling pointers [19]. Thus, a new system programming language is proposed as an alternative to what is already being used by developer communities worldwide.

Authors' address: Miguel Pérez García, miguel.perezg@iteso.mx; Diego Antonio Mejía Sánchez, diegomejia@iteso.mx, Instituto Tecnológico y de Estudios Superiores de Occidente, Periférico Sur Manuel Gómez Morín 8585, Guadalajara, Jalisco, México, 45604.

1.1 The Rust Programming Language as a safe alternative

Rust is a new programming language for systems development originally designed by Graydon Hoare and then sponsored by Mozilla [10]. From the design of the language, it was thought as a progressive replacement for C and C++ [15]. This language has two features needed for safe software development: memory management and memory safety [10]. Languages like Python or Java have automatic memory management strategies often implemented in a runtime environment or language interpreter. These strategies are rarely implemented in embedded systems where time constrains are important like in medical or automotive applications [11]. Rust avoids this problem with a somewhat hybrid approach where memory is manually managed but its rules are hidden behind two concepts present in the compiler: *ownership* [1] and *lifetimes* [2].

1.1.1 Ownership, references and borrowing in Rust. *Ownership* consists of three rules: each variable is the *owner* of its value; each value can only have a unique owner and when the *owner* goes out of scope, the value will be dropped [1]. These rules are enforced at a compiler level so they act as a memory management system that requires no extra logic to work in runtime. These rules are complemented by the concept of *references* and *borrowing* [3]. *References* work just like C pointers, but with the caveat that two kinds of *references* exists: mutable and immutable. All variables are immutable unless they are specified as mutable furthermore during the lifetime of a variable it can have any number of immutable *references* or only one mutable reference. As a side effect, dangling pointers are prevented [4]. The following Rust code shows these concepts.

```
// Immutable variable, A owns value 5.
let A = 5;
// Immutable reference to A.
let ref_A = &A;
// A is not mutable, so mutable references are not allowed.
// The following line throws a compiler error.
let ref_mut_A = &mut A;

// Mutable variable, B owns value 7.
let mut B = 7;
// Since B is mutable, mutable references are allowed.
let mut ref_mut_B = &mut B;
// But no more than one mutable reference is allowed.
// The following line throws a compiler error.
let mut_ref_C = &mut B;
```

The rules stated above are complemented with the *borrowing* concept; if a function receives a *reference* parameter, the *owner* variable is *borrowing* its value to the function; thus it cannot be modified unless a *mutable reference* has been borrowed, however, when a parameter is not a *reference* then *ownership* over the value is passed down to the receiving function rendering the original *owner* invalid when the function returns as shown in this next code section.

```
fn main() {
    let text = String::from("Hello world");
    // This function is borrowing a reference to variable text.
    borrow_func(&text);
    // Using the variable is still valid, it has not been dropped.
    let text_length = text.len();
    // This function is receiving ownership of the vairable text.
    onwership_transfer(text);
    // The variable text is now invalid, the value was dropped
    // at the end of the last function.
    let text_length = text.len();
}
```

1.1.2 Lifetimes. With the introduction of *lifetimes*, dangling pointers are prevented; this problem is caused by freeing a resource and then trying to access it. In Rust, all *references* have their own lifetime, for example; it is not possible to return a reference to a variable that will be dropped at the end of a function without telling the compiler not to drop it; this is done through *lifetime notation* as shown in the following code [2].

```
// Valid lifetime
// Both references share the same lifetime, so both must be valid at
// the end of the execution of the function and can be safely returned
fn from_byte<'a>(bytes_a: &'a [u8], bytes_b: &'a [u8], byte: u8) -> Option<&'a [u8]> {
    for (i, data) in bytes_a.iter().enumerate() {
        if byte == *data {
            return Some(&bytes_a[i..]);
        } else if byte == bytes_b[i]{
            return Some(&bytes_b[i..]);
        }
    }
    return None;
}
// Invalid lifetime:
// Reference bytes_a and bytes_b are not granted to be valid for the duration
// of the execution, each parameter has its own lifetime
fn from_byte(bytes_a: &[u8], bytes_b: &[u8], byte: u8) -> Option<&[u8]> {
    for (i, data) in bytes_a.iter().enumerate() {
        if byte == *data {
            return Some(&bytes_a[i..]);
        } else if byte == bytes_b[i]{
            return Some(&bytes_b[i..]);
        }
    }
    return None;
}
```

In practice, most of the time the compiler is able to figure out which lifetimes have to be used, and when it is not inferred, the developer must state its intention in the code.

2 EMBEDDED HARDWARE ABSTRACTION LAYER (HAL) IMPLEMENTATION

For the demo, FRDM-K64F [5] and STM32F4Discovery (STM32F407G-DISC1) [6] boards were used considering they were available to the team at the beginning of the project. The STM32 board already has a HAL implemented and it can be found as `stm32f4xx-hal` at the https://crates.io repository [7]. However, the FRDM-K64F board does not have a HAL available so it had to be developed for the project.

2.1 FRDM-K64F HAL Architecture

For the basic demo only a single `UART` is required, so all the focus of the project was aimed towards the implementation of the hardware abstraction layer for this peripheral. To make the HAL standard for the Rust ecosystem, a couple of predefined traits had to be implemented: the `Write` and `Read` traits from the `embeded_hal::serial` module can be found in the `embedded_hal` [8] crate in the https://crates.io repository. The traits mentioned before define the minimum methods required for communication through a serial channel. A word size can also be configured for situations where the byte is not the word size for the peripheral, and in the case of the FRMD-K64F board, the word size can be either 8-bit or 9-bit; however, for this implementation only the 8-bit situation was considered. By implementing these traits, as long as libraries that are used in our programs use the same abstractions, they work as *plug-n-play* with our HAL implementation or with another board's HAL. An overview of the full architecture is shown in Figure 1.

The following code is a minimal representation of the usage of the `UART` HAL.

```
let p = k64::Peripherals::take().unwrap();
let port_b = p.PORTB.split();
let pins = cortex_m::interrupt::free(move |cs| {
    (
        port_b.pb17.into_alternate_af3(cs),
        port_b.pb16.into_alternate_af3(cs)
    )
});

let config = Config::new(
    115200.into(),
    Parity::None,
    WordLength::DataBits8 ,
    StopBits::Stop1
);
```

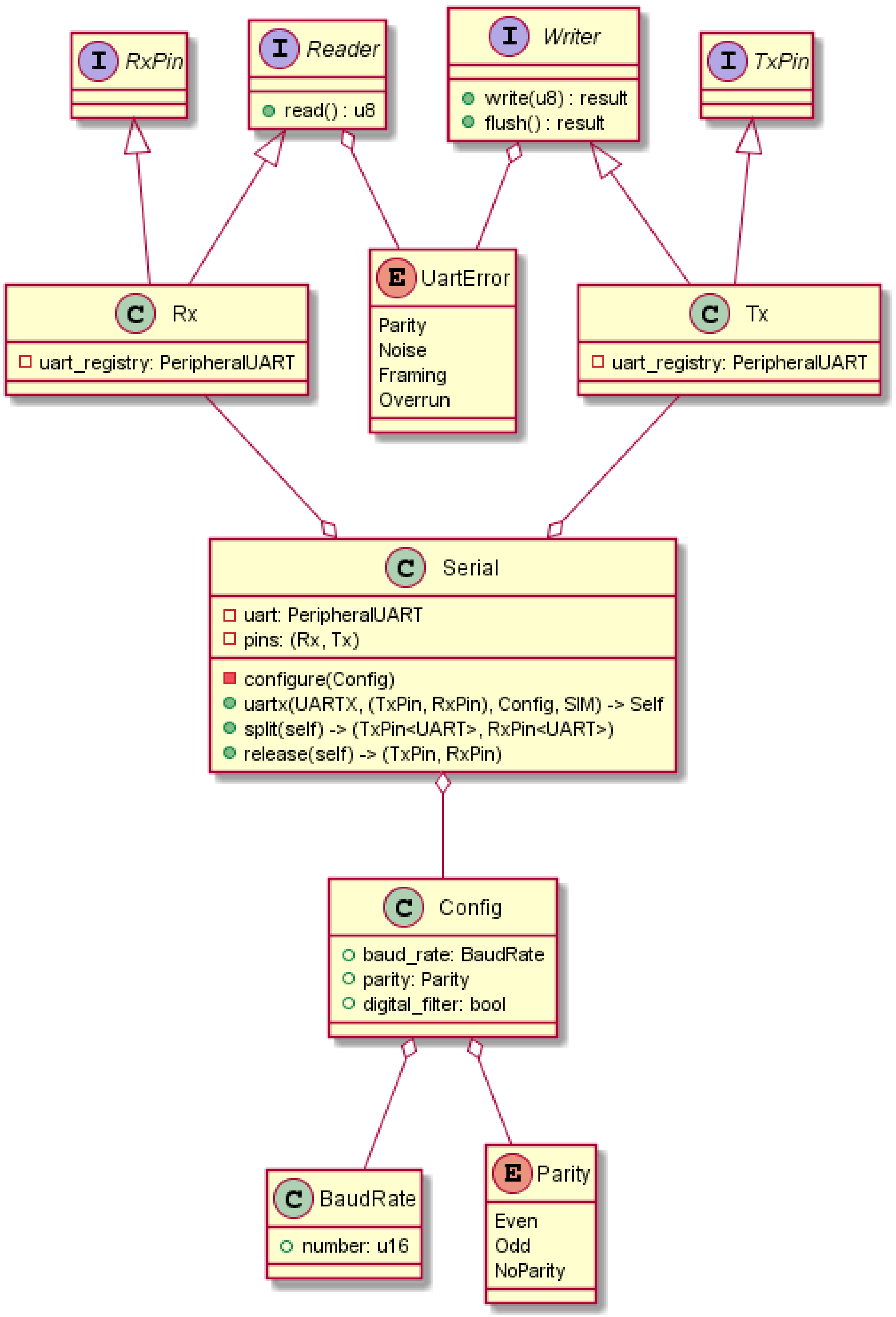


Fig. 1. FRDM-K64F HAL Architecture

```
let mut serial = Serial::uart0(p.UART0, pins, &config, &p.SIM);
```

After configuring the pins inside a critical section, the peripheral is configured and at the end the `UART0` instance is created. Given that the method signature for `Serial::uart0` which is `pub fn Serial::uart0(uart: UART0, pins: (TxPin, RxPin), config: &Config, sim: &SIM)` where `TxPin` and `RxPin` is only implemented for the pins corresponding to the `UART0` instance; the `serial` variable is now the owner of the `UART0` peripheral, and `PB17` and `PB16` pins; no other method can use this `UART` instance nor any of the pins configured during the lifetime of the `serial` variable, therefore the situation in which a pin is configured twice by error is not possible; also it is forbidden to use any other pin which is not related to the `UART` instance since its constructor only defines the pins that are meant to be used. The `Serial`, `TxPin` and `RxPin` structures implement the traits that `embedded_hal` defines at the `serial` module, which are `Read` and `Write`; with the implementation of these traits separating the use of each pin for its purpose is possible; and therefore, its easier to target only the functionality required.

2.2 Hidden implementation details

As previously addressed, when implementing the standard traits defined at `embedded-hal`, it is possible to abstract away the implementation details of the peripheral. The following method definition shows an example.

```
fn write_uart(uart: &mut impl Write<u8, Error>, data: u8) {
    uart.write(data);
}
```

The method does not have any knowledge from the board where it will be executed because every register access is abstracted away by the `Write` and `Read` traits.

3 MODBUS LIBRARY ARCHITECTURE

As a library implementation example, the Modbus communication protocol [12] was used to demonstrate the usefulness of Rust as an alternative for embedded systems programming. The architecture uses marker-like structures to avoid misuse of the protocol. To avoid any kind of heap allocation, a `RingBuffer` [13] is used to receive and store the data at the `ModbusClient` structure. A `new` method is declared which is used to initialize all the operations. This method receives a mutable reference to an array of bytes with the maximum size of a Modbus RTU package to make the reference valid throughout the execution of the program. The array has a lifetime parameter marked as `'a`, which is used by the `RingBuffer` to ensure that the compiler does not mark any of the memory used by this array as safe to clear during any method call.

All other structures have a method called `send` responsible of placing the bytes in the register where it is required by the `UART` interface, but this method does not have knowledge of any specific microcontroller implementation; instead it receives a trait to write data and another to read from the peripheral, which are the `reader` and `writer` parameters, respectively. To be able to use the `UART` from any microcontroller, these traits need to be implemented, which are part of the `embedded-hal` crate in the Rust ecosystem [8]. Only the sections where microcontroller-specific code is required are not sharable between the boards.

3.1 RingBuffer implementation

The `RingBuffer` has two methods that deserve to be explained further, `new` and `get_written` methods; when the `RingBuffer` is created by the `new` method, the mutable reference to the array is then owned by an `enum` that provides exclusive access to the array where data will be written. The `get_written` method returns a reference to the written section of the array inside the `RingBuffer`, but ownership of the array is not transferred down to the caller. Therefore the lifetime of the reference to the `RingBuffer` instance cannot be `dropped` until the reference to is out of scope where the variable has borrowed the reference. As the `RingBuffer` is tide to the `ModbusClient`, each client will have its own buffer and sharing the array is not possible since the reference is mutable.

4 BASIC DEMO

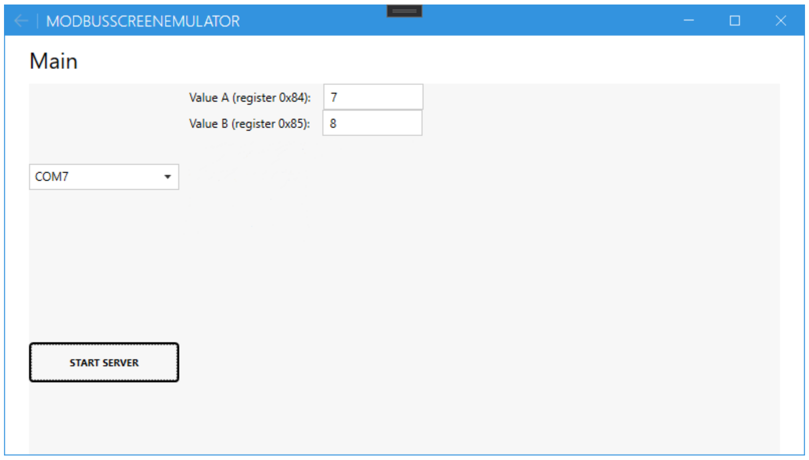
A basic demo can be found at https://github.com/miker1423/article-demo, it sends a single Modbus RTU package through the `UART` to a server written in C# using the `AMWD.Modbus.Serial` library [18]. The firmware running on both the FRDM-K64F and STM32F4Discovery boards is mostly shared. First, each board configures each of its peripherals but only one version of the function is selected based on which compiler flag is active, then the array which will be used in the `RingBuffer` is created with `256` elements. Inside the infinite loop, a `ModbusClient` is created and used to produce the request and at the end the request is sent through the `UART` and the response is then returned by the `send` method, this code piece shows the code explained previously.

```
#[entry]
fn main() -> ! {
    let (mut tx, mut rx) = get_uart();
    let mut buffer = [0u8; 256];
    loop {
        let client = ModbusClient::new(&mut buffer, 0x05.into());
        let result =
            client.read_coil_from(0x02.into())
                .with_quantity(0x25)
                .send(&mut tx, &mut rx);

        delay(10_000);
    }
}

fn get_uart() -> (impl Write<u8>, impl Read<u8>) {
    #[cfg(feature = "k64f")]
    return k64::get_uart();
    #[cfg(feature = "stm32f4")]
    return stm32f4::get_uart();
}
```

Figure 2 show the software emulating a Modbus device where values for registers can be set and in figure 3 the FRDM-K64F is shown with an OLED display showing the values received by the board through the Modbus protocol.



Value A (register 0x84): 7
Value B (register 0x85): 8

Fig. 2. Windows Software emulating Modbus device

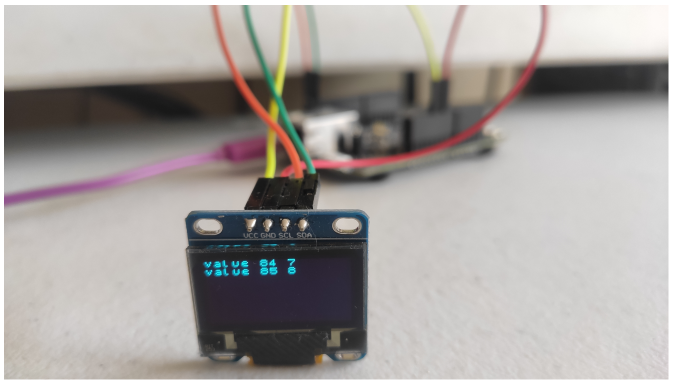


Fig. 3. I2C Screen showing values received through Modbus protocol

5 FURTHER WORK

For the Modbus library to be feature complete it should also implement a client for ethernet interface and a TCP/IP library like smoltcp [9]. It must also implement a server for both a UART and ethernet interface with special features like direct memory access (DMA) or interruptions.

6 CONCLUSIONS

Rust is a great alternative for C and C++. It provides a safety features which are not present in other languages and that improve the security of the program implemented. The borrow checker is a great tool to avoid mistakes such as dangling pointers and memory leaks when programming as close to the metal as it is done in embedded systems for improving the safety of the firmware. When starting a new project it is important to consider the availability of drivers and libraries and given that Rust is mostly an open community effort it has been improving through the years.

7 ACKNOWLEDGE

This work was partially supported by the Mexican federal government through the CONACyT agency.

REFERENCES

[1] [n.d.]. https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html

[2] [n.d.]. https://doc.rust-lang.org/book/ch10-03-lifetime-syntax.html#validating-references-with-lifetimes

[3] [n.d.]. https://doc.rust-lang.org/stable/book/ch04-00-understanding-ownership.html

[4] [n.d.]. https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html#references-and-borrowing

[5] [n.d.]. https://www.nxp.com/design/development-boards/freedom-development-boards/mcu-boards/freedom-development-platform-for-kinetis-k64-k63-and-k24-mcus:FRDM-K64F

[6] [n.d.]. https://www.st.com/en/evaluation-tools/stm32f4discovery.html

[7] [n.d.]. https://crates.io/crates/stm32f4xx-hal

[8] [n.d.]. https://docs.rs/embedded-hal/0.2.4/embedded_hal/

[9] [n.d.]. https://crates.io/crates/smoltcp

[10] 2020. https://en.wikipedia.org/w/index.php?title=Rust_(programming_language)&oldid=994925985 Page Version ID: 994925985.

[11] 2021. https://en.wikipedia.org/w/index.php?title=Garbage_collection_(computer_science)&oldid=1025118711 Page Version ID: 1025118711.

[12] 2021. https://en.wikipedia.org/w/index.php?title=Modbus&oldid=1028761462 Page Version ID: 1028761462.

[13] 2021. https://en.wikipedia.org/w/index.php?title=Circular_buffer&oldid=1031546277 Page Version ID: 1031546277.

[14] Jacob Beningo. 2017. *Reusable Firmware Development A Practical Approach to APIs, HALs and Drivers* (1 ed.). Apress.

[15] evrone. [n.d.]. Why Rust is meant to replace C. https://evrone.com/rust-vs-c

[16] Christoph M. Kirsch and Reinhard Wilhelm. 2007. Grand Challenges in Embedded Software. In *Proceedings of the 7th ACM & IEEE International Conference on Embedded Software* (Salzburg, Austria) *(EMSOFT '07)*. Association for Computing Machinery, New York, NY, USA, 2–6. https://doi.org/10.1145/1289927.1289930

[17] Ryan Levick and Sebastian Fernandez. 2019. We need a safer systems programming language – Microsoft Security Response Center. https://msrc-blog.microsoft.com/2019/07/18/we-need-a-safer-systems-programming-language/

[18] Andreas Müller. 2021. *AndreasAmMueller/Modbus*. https://github.com/AndreasAmMueller/Modbus

[19] Dorottya Papp, Zhendong Ma, and Levente Buttyan. 2015. Embedded systems security: Threats, vulnerabilities, and attack taxonomy. In *2015 13th Annual Conference on Privacy, Security and Trust (PST)*. IEEE, 145–152. https://doi.org/10.1109/PST.2015.7232966

[20] Elecia White. 2011. *Making Embedded Systems* (1st ed.). O'Reilly.