Analytical Instruments Empowered by Rust

Deepak Hegde

Vishal Keshava Murthy

A case study on the use of the Rust programming language in Analytical Instrument development

Presented at Pittcon 2021

Tismo Technology Solutions (P) Ltd.
22/2, Palmgrove Road
Bangalore 560047, INDIA

ABSTRACT

The objective of this paper is to explore the value of the Rust programming language in the development of an analytical instrument. The study was carried out by developing a part of an Analytical Instrument software using Rust. Experimental results and anecdotal evidence suggest that Rust is a viable option to tackle typical challenges associated with developing analytical instruments. Rust's emphasis on safety, its support for modern programming paradigms and a thriving Eco system aids the developer greatly in rapidly developing robust and reliable, high quality code without incurring any performance penalties.

1. INTRODUCTION

A typical analytical system in the embedded space is a combination of a multitude of hardware and software subsystems. The development of these high stakes, complex systems which demand high reliability, computational accuracy and repeatability often poses unique technical challenges to the developer. Long lifetimes, strict regulatory requirements and critical time to market windows further add to the complexity of development from a product management perspective.

Rust is a statically typed, systems level programming language designed for safety, concurrency and performance. The language promises to alleviate the design challenges that plague the world of embedded software development. The study focuses on exploring these issues in detail and the language as a whole, diving into the ecosystem, language constructs, design paradigms and other facets of Rust in the context of software development for an analytical instrument, which in this case is an elemental analyzer.

Background

Analytical systems encompass various hardware interfaces, sensor frontends, external hardware and software systems, communication mediums, computations and HMIs.
An elemental analyzer for example could contain sensors, values, actuators, PWMs, communication interfaces that make up the hardware interfaces of the system. It would also need data acquisition subsystems for measuring temperature, pressure, flow etc. In addition to these hardware interfaces the system would need software subsystems for instrument configuration management, various control processes for all relevant parameters, coupled with sensor data calibration processing and analyzing capabilities.
Processing and control aside, a system of this nature also mandates the need for various forms of communication interfaces and corresponding protocol support. Command interface and UI is also a part of this growing list of requirements.

Software reliability, repeatability of measurements and computational accuracy are critical to products of this nature. The environments in which these devices are deployed demand adherence to multiple safety and regulatory compliance. Time to market is also paramount to the commercial success of such an undertaking.

Owing to the scale of features of a product like this, abstraction of hardware and functionality is a major hurdle in

architecting software. The control and measurement algorithms deployed are arduous to develop, implement and test. The system is inherently asynchronous which makes concurrency and safety cardinal for reliable operation of the software. A wide variety of standard communication interfaces such as CAN, Modbus, OPC etc. that need to be supported adds to the complexity. Features such as diagnostics, firmware upgrade support, intuitive GUI and portability, while not a part of the core features of the system are still imperative from a usability and convenience perceptive.

The key therefore, for successful development of instrument software are robust implementation architecture, support for inherently safe and reliable programming paradigms and the ability to manage and abstract the complexities and real time observance of the system.

C and C++ are the established programming languages currently in use for embedded software development. Something seemingly as innocuous as the use of an uninitialized variable can wreak havoc in an otherwise perfect program. Once pointers, memory management and concurrency are involved, null pointer dereferencing, dangling pointers, memory leaks, stack corruption and a whole host of hard to track problems that affect code in insidious ways are introduced to the code base. The lack of ready to use modules and associated package managers, like npm or NuGet offered by other modern languages also result in slower development process hindered by their inability to leverage mature prewritten modules.

Rust, a relatively new systems programming language, offers an important set of features that promises to address most of the needs established above.

Objectives

The main objective of this study was to evaluate Rust as a language for the development of embedded analytical systems. Quantifiable Code metrics such as code size, binary size were collected and presented. In addition to these, soft metrics like ease of development, debug support, hardware architecture support etc. were also evaluated.

Evaluation of RTIC, a real time framework developed in Rust that employs an interrupt driven approach to writing concurrent code was also an important goal. The tool chain, package management system, external library support etc. were also put under scrutiny since the development ecosystem is an inseparable part of the software development experience.

Since Rust is still evolving, it was also a part of the research effort to recognize shortcomings of the language and determine future areas of study.

Scope of the research

1. Evaluation of Rust language features, development ecosystem and library support

2. Evaluation of RTIC framework

3. Collection of code metrics for Rust code compared against equivalent C code

Research methods

A development environment was set up with the Rust toolchain including the Rust stable and nightly compiler, package manager cargo, Rust formatter, Rust language server RLS and debugger codeLLDB.

A TCD controller application, one of the subsystems of the elemental analyzer involving temperature control and measurement, configuration management, timed tasks, a simple CLI, status indication and a PID loop were developed in both C and Rust. The code bases were developed iteratively using established best development practices and were subject to multiple reviews and refactoring to ensure code quality and performance.

Code metrics were collected for both applications under various optimization settings. Other non-quantifiable aspects of development such as development environment, toolchain, ease of development, testing support etc. were also evaluated and presented.

The structure of the document

The following sections will be covered in the report:

1. Overview of Rust
2. Embedded Rust
3. Embedded Rust ecosystem
4. Embedded Rust Crates
5. Developed application
6. Useful Language features
7. Observations

2. DETAILED ANALYSIS

2.1. Overview of Rust

Rust[1] is a multi-paradigm systems programming language developed by the Mozilla foundation with heavy emphasis on safety, performance and support for concurrency. This systems programming language converts memory and concurrency related run time problems into errors that get caught early during the compilation process. The compiler accomplishes to do so by adhering to the following fundamental principles.

1. Immutability: Every entity is immutable by default
2. Ownership: Every entity has a clear owner at all times which determines its scope and lifetime. This enables memory safety at compile time without relying on garbage collection

2.2. Embedded Rust

Bare-metal hardware environments are first class citizens of Rust[2]. The language itself is built in layers with a core layer libcore which implements platform agnostic operations, provides API for language primitives and APIs for processor features. All platform integration APIs and the run time environment are implemented in the stdlib. Development for embedded applications take place in a no_std environment meaning that only the core layer is used. Memory management schemes, run time environments[12], stack protection schemes and data structures[15] such as vectors and maps are added as required externally on top of the libcore using crates[11].

3. Move semantics: Assignment, copy and move operations are clearly distinct and operate by different sets of rules.

While the core features described above help with code correctness, a variety of concepts and paradigms inspired heavily from other languages such as pattern matching, anonymous functions and algebraic data types from functional programming languages, traits and structures from object-oriented programming and features such as macros and template programming from metaprogramming languages makes Rust a very powerful and expressive language.

Rust also features the concept of unsafe code, sections of the code can be marked unsafe, inside of which potentially dangerous operations such as raw pointer manipulation can be made under the discretion of the programmer foregoing the compiler checks for extra flexibility. Other prominent features include built in support for testing, code formatting and profiling.

This distinction between the core and the standard library ensures that the features required for the application can be customized as required based on the target hardware and application requirements. Rust also extends the powerful Ownership mechanism to manage the peripherals by treating them as variables. One can enforce singleton pattern on peripherals, and these can be shared as read-only, read-write types using the borrow checker paradigm

Since concurrency is one of the core focus of rust, depending on the target platform, there is native support for atomic operations. Synchronization mechanisms such as critical sections, mutexes, interior mutability refcells are also provided. Real

Time Interrupt Driven Concurrency framework[6], RTIC is another option if RTOS like functionality is required.

Support for interoperability with C is another attractive feature of rust. This is useful when validated C code needs to be integrated with Rust. Rust provides Foreign Function interface (FFI) for this purpose. Tools like bindgen aid in generation of mapping metadata between languages.[1][8]

2.3. Embedded Rust ecosystem

A good development ecosystem is critical for the adoption of the language into production quality software work. The Rust ecosystem while still in its infancy, provides excellent tools and reasonable coverage for most target platforms. The toolchain set up process is fairly straight forward, and is explored in detail in the later sections of the paper.

There is extensive support for the ARM cortex M series of processors, with Hardware abstractions available for MCUs from major chip vendors such as STM, Microchip, NXP, TI and Nordic. Xtensa and AVR are some of the notable architectures that yet aren't supported.[11]

The conventional layered approach for development is followed, with the lowest layer being the Peripheral Access Crate. The PAC defines memory mapped registers for the particular target hardware. This layer is generally used in conjunction with a micro architecture crate that encompasses routines and functions common to the target architecture. SVD2Rust is a tool that can be used for the generation of these target specific Peripheral Access crates.[14]

HAL crates are built atop PACs, by providing definitions to traits defined in embedded-hal[12]. This layer provides peripheral specific hardware APIs using which applications can be built. It should be noted however that these HALs are community driven and often do not support all features provided by the hardware due to which Hardware registers of the device may still have to be manipulated directly through services provided by the device specific PAC.

RustC[3], the Rust compiler is pivotal to the traction that Rust has been gaining lately. Compilation times can be pretty long in contrast with that of C or C++ but the output is fairly well optimized for the target platform. The error messages are very explicit and provide relevant and helpful suggestions. Rust also ships with package and build manager cargo that helps with the build process as well with dependency management and integration of external libraries crates. RustC, and cargo coupled with the debugger GDB can be setup inside text editors such as VScode for a complete building, flashing and debugging platform. The Rust ecosystem also features inbuilt support for testing and code profiling. Formatter rustfmt and static code checker rust-Clippy are other utilities packaged into the system.

2.4. Embedded Rust crates

Rust crates are a large part of the appeal of embedded rust. The libraries managed by Cargo[4] helps the developer in leveraging the Rust community and integrate tested and verified features into the code. Crates are an indispensable part of the Rust ecosystem.

Since bare-metal embedded Rust development takes place in the embedded environment it is up to the developer to select the required memory management scheme utilizing the respective crate[11].
PACs and HALs introduced earlier, if available are also developed in the form of crates that can be used to build up applications. Several crates that are designed to be used in no_std environments provide a wide variety of functionality ranging from atomic access, bit field manipulation methods, circular buffers and heapless data structures. CRCs, linear algebra libraries, FFT support are some the prominent math libraries. Device drivers[17] for a large class of devices ranging from simple peripheral based sensors, memory modules, to displays and cellular modules are also pervasive. Stacks developed in Rust for Bluetooth, Ethernet are some of the protocol crates that are gaining traction. PID control[16], CLI[15] were some of the application specific crates evaluated and used in this project.

2.5. Application overview

The TCD subsystem of an elemental analyzer for the detection of nitrogen content in protein samples was developed for the sake of the experiment. The overall system works on the Dumas principle in which a sample is purged and heated in a high temperature combustion furnace in the presence of pure oxygen such that the sample decomposes into carbon dioxide, water, nitrogen dioxide in addition with nitrogen as several oxides.

The combustion products are collected and the gas mixture is passed over hot copper to remove any oxygen and convert nitrogen dioxides into molecular nitrogen. The sample is passed through traps that remove water and carbon dioxide. The measured signal from the thermal conductivity detector for the sample is then converted into total nitrogen content.

The TCD detector subsystem was implemented in both C and Rust on TM4C123XX[10], an ARM cortex M4 based board. Initial prototyping was carried out using the Tiva launch pad with the same MCU and an STM32F407 discovery board on which the RTIC framework[6] was evaluated.

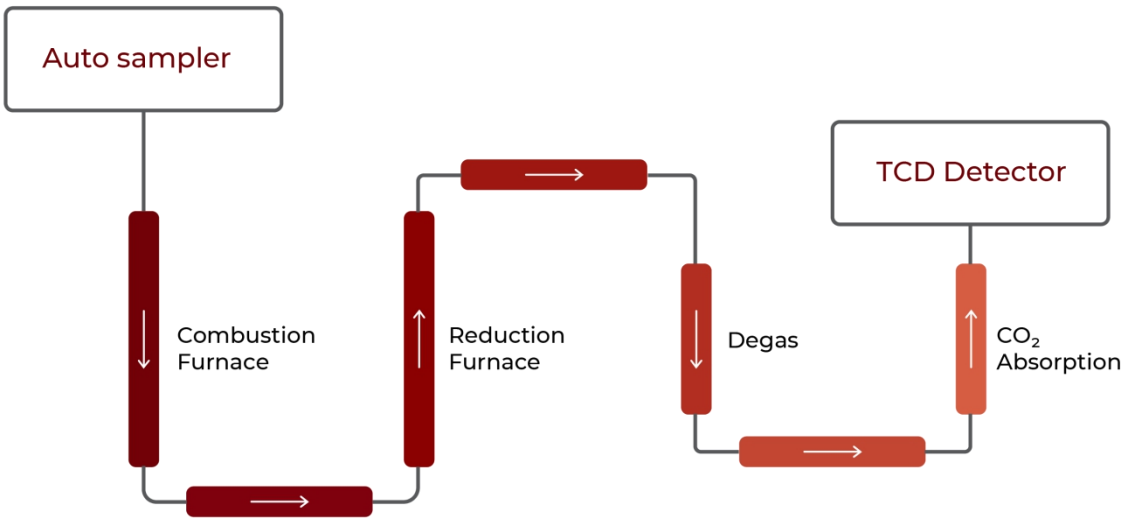

Figure 1

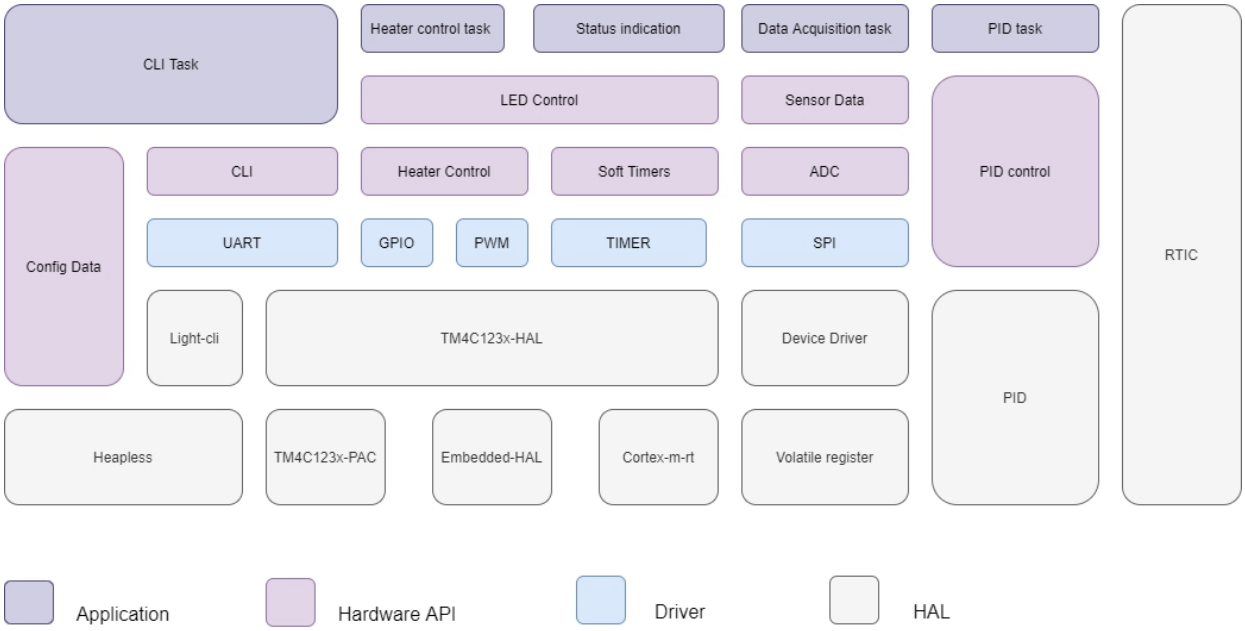

Figure: 2

A traditional layered architecture was utilized for the code structure with the MCU HAL making up most of the lower layers through the tm4c123x-hal[13] crate for dealing with various onboard peripherals such as SPI, UART etc. Additional crates were used for runtime[11], memory management[14], and low-level register access[17] functions. Other purely software constructs such as PID control[16] and CLI[15] were also built around crates for rapid prototyping and development.

The hardware API layer was built atop the previously described HAL. Object based design approach was utilized here, grouping all similar functions into modules that exposed to the application layers above only the functionality required.

The application layer is comprised of independent tasks for PID control, Heater control, Data acquisition and a command Line Interface task. These were tied together using the RTIC framework.[7]

2.6. Useful language features

The following language features were heavily utilized for the development of the project.

Struct and traits

Structures in Rust are similar to C struct allowing heterogeneous data types. Unlike a traditional Object Oriented programming language Rust has no support for classes. Object oriented behavior is implemented using struct and traits instead. Traits are implemented for structures giving the structure traditional class like methods. These methods however need to take an explicit self parameter if they need to act on themselves. Since immutability is a core aspect of the language all elements and traits are private and immutable by default. Consider the heater struct defined below:

```
pub struct Heater
{
  status : bool,
  pwm_pin : pwm::EvenPWM<tm4c123x_hal::tm4c123x::TIMER2>,
  pwm_period_us:u32 ,
  pwm_duty_cycle:u32,
}
impl Heater
{
  pub fn disable(&mut self)
  {
    self.status = false;
    self.pwm_pin.disable(());
  }

  fn dutycycle_to_period(&mut self,duty_percent:u32) -> u32
  {
    let res = duty_percent*(self.pwm_period_us)/100;
    return res ;
  }
}
```

Code snippet: 1

Here, the disable trait is defined for structure heater. The elements on the structure are private by default and therefore are not accessible to outside modules. The disable function takes a mutable reference to itself and acts on the elements of the structure Heater. This API, marked as public is exposed to other modules so that they can interact with the heater structure. The function dutycycle_to_period is a private method that is inaccessible to outside modules.

Option type

Null types are a common source of bugs in most traditional programming languages. Rust provides Option type and Result type that can represent variables that can be uninitialized. Internally they are implemented using templates.

Consider the example below:

```
pub fn do_pid_control_task(softtimer_block:&mut SoftTimers, task_frequency:u32, pid_bloc
k : &mut PidControl, prev_output : f32) ->Option<f32>
{
  if softtimer_block.check_timer_ms(ESoftTimers::LedSoftTimer)
  {
    let res:Option<f32>;
    let control_output = pid_block.run_loop(prev_output);
    res = Some(control_output);
    softtimer_block.set_timer_ms(ESoftTimers::PidSofttimer,task_frequency);
```

```
        return res;
    }
    else
    {
        return Option::None;
    }
}
```

Code snippet: 2

The temperature control task runs only periodically and returns *res* which is an option type. The *res* type contains the result of the operation if the timer has expired or returns the None type

```
heater_control::do_temperature_control_task(&mut my_mcu.heater_controller , pid_output)
```

Code snippet: 3

This result is captured in option variable pid_output and passed down to the do_temperature_control_task. The function can then act on this parameter depending on whether it has a value or not.

Pattern Matching

The match statement is used extensively in the program, particularity when the option type is involved. The do_temperature_control_task from the previous example utilizes match effectively as shown.

```
pub fn do_temperature_control_task(pwm_block : &mut Heater, control_input_option : Option<f32>)
{
    match control_input_option
    {
        Some(val) => {pwm_block.set_dutycycle(val as u32);},
        None => {},
    }
}
```

Code snippet: 4

Here, if the option type does not have a value then, no action is taken.

Closures

Closures or anonymous functions as known in C++ are used extensively in the PACs. Register manipulations are done utilizing closures

```
timer.icr.write(|w| w.tatocint().set_bit());
```

Code snippet: 5

In this example, |W| is the closure that performs the operation for setting a respective bit.

Modules

The Rust file system employs modules for structuring code. Files are treated as modules each with their own namespaces, with access specifier as explained above determining the visibility of functions inside. The scope operator as used in C++ is also available in Rust to keep the code structured and modular. The syntax results in highly modular yet readable code without having to deal with separate implementation and interface files.

Module heater control is imported into the file here using the keyword mod

```
mod heater_control;
heater_control::do_temperature_control_task(&mut my_mcu.heater_controller , pid_output)
;
```

Code snippet: 6

Public functions exposed by the heater module are accessed using the scope operator in this file.

Unsafe blocks

unsafe keyword is necessary in places where the compiler cannot guarantee safety. While the language discourages the use of *unsafe*, it is useful when dealing with raw pointers or global values when it is determined to be safe to do so by the developer as demonstrated in the code snippet below.[9]

```
unsafe {
    tm4c123x_hal::tm4c123x::NVIC::unmask(tm4c123x_hal::tm4c123x::Interrupt::TIMER0A)
  }
```

Code snippet: 7

Templates

Templates are powerful features from the metaprogramming paradigm. It is useful for generic programming and is used by Rust to accomplish monomorphization. Templates result in the increase in compile types but incur no performance penalties. Templated types are also used in the HAL libraries for typestate checking[3]. The hardware therefore can be characterized as having only a limited set of valid states with a clear sequence of possible operations. A clear example of this is the GPIO pin which is modeled such that the validity of operations on it is determined by the state of the pin. This results in much safer code that prevents wrong initialization at compile time

```
gpioc::PC5< AlternateFunction< AF1,PushPull>>
```

Code snippet: 8

PC5 here is defined as a pin of type Alternate type AF1 with push pull configuration. The methods that can be applied on this pin are dictated by the

typestate. It would result in a compiler error for an invalid method not defined for pins of this type.

Concurrency

Rust provides several features to aid concurrency.

The common paradigm of Foreground background process of using superloops with interrupts is well supported, with the use of attributes to define interrupt handlers. On a few platforms atomic access is supported for simple and effective means of guaranteeing safe concurrent access.

While the usage of global mutable data is discouraged to prevent data race conditions, they can help still be used inside unsafe blocks under the programmers discretion.[3]

The example below demonstrates the use of both the interrupt attribute for defining the timer0 interrupt as well as the utilization of global mutable variable RUNNING_COUNTER.

```
static mut RUNNING_COUNTER:u32 = 0;
#[interrupt]
fn TIMER0A()
{
      unsafe {
        let periph_temp = tm4c123x_hal::Peripherals::steal();
        let timer = periph_temp.TIMER0;
        timer.icr.write(|w| w.tatocint().set_bit());

        if RUNNING_COUNTER > MAX_TICK {RUNNING_COUNTER = 0;}
        else {RUNNING_COUNTER+= 1 ;}
                }
 }
```

Code snippet: 9

Critical sections are another crude but effective concurrency mechanism supported by Rust. The functionality is provided by runtime environment crates such as cortex_m[12]. These are often used with closures, whose operations take place within the critical section. The runtime environment crate cortex_m also provides standard resource management mechanisms such as mutexes to ensure safe concurrent access.

In the example below, the value of LED_MUTEX is set inside a critical section.

```
static LED_MUTEX: Mutex<Cell<u8>> = Mutex::new(Cell::new(0));
cortex_m::interrupt::free(|cs|  LED_MUTEX.borrow(cs).set(2) );
```

Code snippet: 10

Real-Time Interrupt-driven Concurrency (RTIC)[7] is a framework that can be used for the development of larger more intricate real time systems. Features such as tasks, message passing, queues and scheduling provided by this framework makes this an excellent middle ground between developing concurrent programs from scratch using the concurrency mechanisms explained above and using full blown schedulers such as FreeRtos.

2.7. Development experience

Learning

The initial learning curve for Rust is steep especially for programmers with only C/C++ background. Programming concepts novel to Rust such as the borrow checker and move semantics are challenging and take time to get accustomed to. HAL interface construction requires careful thought and planning, established imperative or object oriented approach for the same are hard to express in Rust. There are various official and community driven high quality documentation for embedded systems development in Rust. The crates however are purely community driven and therefore vary wildly in quality.

Setup and tools

Setting up the development ecosystem is not a very involved process. There are several tools that help the developer get productive with the tool chain fairly early. Rustup is the installer which aids in setting up the compiler and package manager for development in Windows, Linux and Mac environment. Package management is handled by the Rust package manger Cargo, which also serves as the build system with which project creation, library creation and management are handled. CodeLLDB in conjunction with JlinkGDB was used for flashing and debugging binaries.

Since there were no IDEs for Rust development at the time, Microsoft Visual Studio Code was used with the official Rust extension. Tasks were setup for building and flashing the project within the IDE. Debugging within the IDE was setup using cortex-debug extension used in conjunction with Jlink GDB server.

Productivity

With the initial set up out of the way, community driven libraries hosted on crates were a good starting point for the project. Embedded HAL implementation for TI and STM boards were readily available, although some features were unavailable or under the process of development requiring register level manipulation of the hardware using the respective Peripheral Access crates. This is in stark contrast with C code since chip manufacturer provide near complete HAL which cuts down on development time of lower layers.

Application development on the other hand was made much simpler due to external Crates. Suitable implementations for circular buffers, PID algorithms, CLI implementations etc. hosted on crates.io were easily integrated into the code base. Inbuilt support for testing and a built-in formatter also contributes to writing better quality code faster.

Ease of development

Rust provides strong abstraction with no performance tradeoffs which makes for highly expressive code. The compile times are quite large especially if templates are used heavily, due to all the static checks performed. The error messages provided by

the compiler are helpful and generally very accurate. Use of modules help with code architecture and help develop highly structured easily maintainable code. Rusts statically typed nature feel prohibitive at times since it disallows implicit conversions, or unhandled cases thereby making it hard to use for prototyping. The use of raw pointers is discouraged and the language does not fully support conventional OOP paradigms.

2.8. Observations

The developed C and Rust code were used for the capture of key code metrics. The development process was kept identical for both languages. An effort was made to use idiomatic conventions to result in not only high performance but also high quality maintainable code. Memory footprint data were collected from release versions of both code bases with link time optimization enabled. The results are tabulated below:

Size	No Optimization		Size Optimized	
	ROM	RAM	ROM	RAM
C	20956	1348	18970	1243
Rust	100780	24	27936	24
The code footprint in Rust is 50% larger				

Table: 1

Lines of code were also captured for both metrics as they are indicative of the development effort.

Lines of code	
C	1750
Rust	1500
Rust code-base is 15% smaller	

Table: 2

Check Appendix section for metrics collected in other optimization profiles.

SUMMARY AND CONCLUSIONS

The Rust programming language offers a rich set of features emphasizing safety which results in the development of reliable software. Safe programming paradigm helps in meeting safety requirements whereas built-in support for testability helps with reliability and verifiability of critical applications. Development process aided by modern programming constructs and portability across hardware, thanks to highly modular architecture, helps in improving time to market. The open source community driven development helps in quick adaptability in both developer communities and organization looking for a safe, sustainable and stable ecosystem.

Code metrics collected suggest that the code developed in Rust while about fifty percent larger than code developed with its counterpart C, is smaller in terms code base by close to fifteen percent. These penalties however can be overlooked considering improvements in code quality. While the ecosystem for embedded Rust is satisfactory but pales in comparison to the maturity that C/C++ provides. However the ecosystem is continuously evolving, and as the language gains traction this aspect will get better. Overall, this modern language that is not weighed down by legacy like C++ is, offers a refreshing new option which, regardless of its future demands strong consideration

References

1. S. Klabnik, C. Nichols. (2020). The Rust Programming Language.
<https://doc.rust-lang.org/book/title-page.html>

2. Embedded Working Group. (2020). The Embedded Rust Book.
<https://docs.rust-embedded.org/>

3. Embedded Working Group. (2020). The rustc book
<https://doc.rust-lang.org/rustc/what-is-rustc.html>

4. Embedded Working Group. (2020). The Cargo book
<https://doc.rust-lang.org/cargo/index.html>

5. Embedded Working Group. (2020). The Rustonomicon
<https://doc.rust-lang.org/nomicon/index.html>

6. P. Lindgren, Embedded Systems Group. (2020). Real-Time Interrupt-driven Concurrency.
<https://rtic.rs/0.5/book/en/preface.html>

7. Embedded Working Group. (2020). Discovery.
<https://docs.rust-embedded.org/discovery/>

8. Embedded Working Group. (2020). The Embedonomicon.
<https://docs.rust-embedded.org/embedonomicon/>

9. Texas Instruments. (2014). TM4C123GH6PM Microcontroller.
<shorturl.at/btxB4>

10. The resources team. (2021). rust-embedded
<https://github.com/rust-embedded/awesome-embedded-rust>

11. The Cortex-M Team. (2020). Cortex-m-rt
<https://docs.rs/cortex-m-rt/0.6.13/cortex_m_rt/>

12. J. Pallant, J. Aparicio. (2020). embedded-hal
<https://docs.rs/embedded-hal/0.2.4/embedded_HAL/>

13. D. Wood, J. Pallant, J.Aparicio, M. poulhies. (2020). tm4c123x-hal
<https://docs.rs/tm4c123x-hal/0.10.2/tm4c123x_hal/>

14. J. Aparicio, P. Lindgren. (2021). heapless
<https://docs.rs/heapless/0.6.1/heapless/>

15. R. Horn. (2019). light-cli
<https://rudihorn.github.io/light-cli/light_cli/>

16. K. Elkabany. (2020). PID Controller for Rust
<https://docs.rs/pid/3.0.0/pid/>

17. D. Dockter. (2021). device-driver
<https://docs.rs/device-driver/0.1.1/device_driver/

APPENDIX

Appendix A

Optimization results for Rust code base

Optimization levels	Text	Data	BSS
No Optimization	100780	8	16
Size optimized	27936	8	12
Size optimized	28480	8	16
Size optimized	38392	8	16

Table: 3