WHY RUST

FOR EMBEDDDED DEVELOPMENT

Note: images generated with Adobe Firefly

SUMMARY

This whitepaper discusses the advantages of the Rust programming language, focusing on usage in embedded systems and the safety benefits it brings compared with classical programming languages.

This whitepaper will also present practical examples and comparisons with the default programming language for embedded systems - C.

Limitations: While this whitepaper does not engage in performance metrics - feel free to reach out to OxidOS Automotive for further information.

DISCLAIMER: While this paper presents differences from C/C++ that are viewed as downsides, one must understand that C and C++ are very powerful languages that made several design choices that were very powerful at the time they were written.

Rust fully benefits from the learnings of these two languages and from modern compiler optimizations that were not available at the time.

CHAPTERS

INTRODUCTION INTO RUST

Rust is a modern general-purpose programming language that prevents memory issues at compile-time - reducing memory access-related issues without needing heavy additional tooling. Rust is a memory-safe, type-safe, thread-safe programming language that provides execution speed and memory footprint similar to C.

Rust is now one of the fastest-adopted programming languages and the best alternative to C in embedded development (where modern high-level languages such as Python, Java, and other languages could not run). Rust is also recommended for secure software development by the NSA.

In terms of adoptions - Rust is getting traction: the first non-C programming language accepted in the Linux kernel (for safety reasons); Google, Amazon, Microsoft, and many others are using Rust to rewrite safety critical components.

ECO-SYSTEM OVERVIEW

Before diving into the technical details, let us first address the question – is anyone looking to use Rust in embedded applications?

The good news is that yes - rust is getting traction in the embedded eco-system, with players from all categories experimenting with Rust to replace (in part) C and C++. A few examples follow below – to the best of our knowledge *(date – April 2024).*

Ferrous Systems GmbH, AdaCore, and HighTec EDV-Systeme GmbH and possibly Green Hills - are working on a certified Rust compiler for the Automotive Industry.

AUTOSAR and SAE International have working groups on Rust for automotive - with SAE kicking off the SAfEr Rust Task Force.

Lauterbach GmbH offers essential Rust support and is expanding support for debugging embedded Rust code.

Infineon Technologies, STMicroelectronics, and at two major chip manufacturers that we can't disclose are working on supporting Rust on their automotive-grade microcontrollers.

In the automotive industry - CARIAD, Elektrobit, Ford, Renault, Toyota, and Volvo Cars have announced they are doing Rust based development.

MAIN IDEAS

The tagline of Rust is No Undefined Behavior.

Whenever there is code written, there is only one deterministic path of what that code can do.

To achieve this, Rust has:

1. no null reference; the Rust compiler explicitly asks developers to check this;
2. no implicit cast, even adding a u32 to a u8 must be casted;
3. safe access to shared data across threads verified at compile time;
4. uses type states to move runtime checks to compile time and force developers to check;
5. clearly defined data types, unlike i8 or u128;
6. safe unions, that provide a discriminant to prevent wrong interpretation of data;
7. clear code organization into crates and modules;
8. backward compatibility at crate level.

The following four chapters will go deeper into the above and add examples.

LANGUAGE FEATURES

No null

Safe Rust does not provide the concept of null reference. A variable is either a valid reference, or it does not exist. To represent the non-existing reference, Rust uses the Option enum.

```
let possible_null: Option<&R> = None;

let possible_null: Option<&R> = Some(a_valid_ref);
// Using the internal reference requires checking.

match possible_null {
    Some(reference) => {
        // use the reference
    },
    None => {
        // there is no reference to use
    }
}
```

In most cases, the Rust compiler is able to optimize this representation into a pointer length value, as it represents internally the None value as NULL.

As this can become very long, Rust provides some convenience functions and the ? operator.

```
let value = possible_reference.unwrap_or(value_if_none);

// return None, pass the value upwards the stack
let value = possible_reference?;

// return an Err(error) up the stack
let value = possible_reference.ok_err(error)?;
```

Rust makes it impossible to have a null reference and obliges the developer to ensure this never happens.

C/C++ do not have any of these features built-in the language and developers sometimes forget to check if a reference is != NULL. This leads to undefined behavior.

Safe data types

Data type sizes are a big issue in C/C++. Names like int or short do not have a clear meaning when it comes to data sizes, and as such depend on the compiler implementation.

Self-defining numeric data types

There is no confusion about the size of the numeric data types, they do not depend on the compiler.

Data Type	**Signed**	**Size**
`u8`	unsigned	1 byte
`i8`	signed	1 byte
`u16`	unsigned	2 bytes
`i16`	signed	2 bytes
`u32`	unsigned	4 bytes
`i32`	signed	4 bytes
`u64`	unsigned	8 bytes
`i64`	signed	8 bytes
`u128`	unsigned	16 bytes
`i128`	signed	816 bytes
`usize`	unsigned	*word* size
`isize`	signed	*word* size

Boolean

Rust distinguishes between boolean values and numbers. The value true is not equivalent to any number that is different from 0.

Control statements, such as if and while require an expression that evaluates to a boolean value.

Source code like if (x=5) generates a compilation error in Rust.

Safe Unions

Rust prevents misinterpretation of data structures by providing enums, which are like C/C++ unions with a discriminant value.
The Value type is represented similar to a C union, but has a discriminant value.

```
enum Value {
    U32(u32),
    I32(i32),
    Complex {
        real: f64,
        imaginary: f64
    }
}
```

The use of the internal stored values requires a match statement. This prevents undefined behavior, as the developer does not have to assume the data value type.

```
match value {
    U32(u32_value) => { /* use the u32_value */ },
    I32(i32_value) => { /* use the i32_value */ },
    Complex{real, imaginary} => { /* use real and imaginary
*/ },
}
```

While Rust does provide union for Foreign Function Interface (interfacing other languages, like for instance C and C++), these are marked as un safe.

Index Verification

Rust adds array indexing verification and prevents buffer overflow memory issues.

- Array indexes are always of type `usize`, and as such negative indexes are not possible.
- If an array index cannot be proven correct at compile time, a runtime check will have to be written inserted the code.
- Array slices are references and have as payload the size of the slice.

UTF-8 out of the box

Standard Rust strings and string slices are always UTF-8.

Safe Concurrency

The borrow checker requires two simple rules:

- a value can have any number of concurrent immutable borrows &value or
- a value can have one single mutable borrow &mut value.

Rust provides two special traits, Sync and Send that, together with lifetime annotations and the borrow checker rules, allow the compiler to reason about the correctness for the concurrent code.

This prevents concurrency issues at compile time. There is simply now way to borrow a mutable value to several threads, unless the developers use some kind of synchronization mechanism, like spinlocks or mutexes.

The spawn function creates a new thread. In the new thread it runs a closure that it receives as a parameter. By using lifetime annotations and the Send trait, it can impose compile time restrictions. In this example, the compiler enforces the following aspects:

The closure F can only:

- own values (which, if shared are not accessible by the owner) that implement the Send trait (which means they are safe to be sent across threads - for instance a MutexGuard is not safe to be send across threads);
- contain values that implement the Send trait;
- contain references that have a 'static lifetime, which means that never go out of scope; they will always be valid, and the caller of the spawn function cannot provide local variables references to a new thread.

The return type of the thread can only:

- own values (which, if shared are not accessible by the owner) that implement the Send trait (which means they are safe to be sent across threads - for instance a MutexGuard is not safe to be send across threads);
- contain values that implement the Send trait;
- contain references that have a 'static lifetime, which means that never go out of scope, they will always be valid, and the caller of the spawn function cannot provide local variables references to a new thread.

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
where
    F: FnOnce() -> T + Send + 'static,
    T: Send + 'static,
```

The 'static lifetime requirement is essential. In the usage example, a is a local variable that goes out of scope at the end of the current function. A reference to it, &a does not have a 'static lifetime. The new thread cannot use it though, as a might go out of scope before the thread uses it.

```
fn run_worker() {
    let mut a = vec![1, 2, 3];

    thread::spawn(|| {
        println!("hello from the first scoped thread");
        // `a` might not be valid here anymore
        dbg!(&a);
    });

    // This will be executed immediately after the `spawn
function, probably before the thread actually executes.
    // `a` goes out of scope here, making `&a` invalid.
}
```

Sending only 'static references is a limitation, so newer Rust versions offer an improved version of the spawn function that is related to a Scope structure and uses 'scope generic lifetime.

```
pub fn scope<'env, F, T>(f: F) -> T
where
    F: for<'scope> FnOnce(&'scope Scope<'scope, 'env>) ->
T;

impl Scope {
    pub fn spawn<F, T>(&'scope self, f: F) ->
ScopedJoinHandle<'scope, T>
    where
        F: FnOnce() -> T + Send + 'scope,
        T: Send + 'scope;
}
```

In the usage example, a is a local variable that goes out of scope at the end of the current function. A reference to it, &a does not have a 'static lifetime. The new thread can use it though, as the scope function assures that it will wait for the threads started from within the scope before returning.

```
fn run_worker() {
    let mut a = vec![1, 2, 3];

    thread::scope(|s| {
        // the thread is started from withing the `s`
scope.
        s.spawn(|| {
            println!("hello from the first scoped thread");
            // We can borrow `a` here.
            dbg!(&a);
        });
    }

    // This will be executed only when all the threads
started by `s.spawn` will finish execution.
    // `a` goes out of scope here.
}
```

Generic Bounds

Rust code that contains the equivalent of C++ generics is fully checked to be correct at definition time.

```rust
fn max<V> (v1: V, v2: V) -> V {
    if v1 > v2 {
        v1
    } else {
        v2
    }
}
```

Rust rejects generic code that has errors regardless of it being used or not. The code above has an error, as the compiler cannot verify that the type V is comparable.

```
2 |     if v1 > v2 {
  |        -- ^ -- V
  |        |
  |        V
  |
help: consider restricting type parameter `V`
  |
1 | fn max<V: std::cmp::PartialOrd>(v1: V, v2: V) -> V {
```

The code above will compile in C++ and will fail when used.
This requirement offers a very powerful guarantee for libraries. If a library compiles, there will not be any type compatibilities when using it.

```rust
fn max<V>(v1: V, v2: V) -> V
where
    V: PartialOrd + Copy,
{
    if v1 > v2 {
        v1
    } else {
        v2
    }
}
```

Zero Cost Abstractions

Developers don't have to pay the cost of dynamic dispatch if not necessary. The same source definition can be used with both method dispatch means, without any modifications.

```
trait Worker {
    fn run(&self) { }
}
```

Using any data types that implement the Worker trait with a generic type will generate static function dispatch. The data structure will have no vtable created and the compiler will statically dispatch the run function. This allows the compiler to fully optimize the code.

```
fn runner(worker: &impl Worker) {
    worker.run();
}
```

If any of the usages include a dyn Worker usage, the compiler will automatically generate the vtable and use dynamic dispatch.

Java uses always dynamic dispatch and pays the performance penalty. C++ requires the developer to annotate methods with the virtual keyword. failing to do so leads to code that runs incorrectly.

```
fn runner(worker: &dyn Worker) {
    worker.run();
}
```

Prevent trait objects

The Rust compiler decides whether to add vtable and dynamic dispatch if and only if a trait is used as a trait object (&dyn Trait). Whenever using trait objects, the compiler does not know the size of the actual data type that will be used at runtime. Requiring that all data structures that implement the trait to be Sized , prevents the compiler from using dynamic dispatch. This means that all these data structures need to have their size known at compile time.

```
trait Worker: Sized {
    fn run(&self) { }
}
```

The same trait bound can be used to prevent only a part of the methods to be dynamically dispatched. The run_optimized method cannot be used with dynamic dispatch, as it requires the implementing data type to have a known compile time size. Rust will not allow calling this method from &dyn Worker data types.

```
trait Worker {
    fn run(&self) { }
    fn run_optimized(&self) where Self: Sized {}
}
```

C++ allows calling statically dispatched methods from dynamic contexts. However, it calls the wrong methods, calling the methods defined on the defined type, instead of the actual subtype.

MEMORY MANAGEMENT

One important aspect that Rust brings at the table is the ability to track references. The compiler is able to track references across function calls only by looking at the function's declaration.

When working with references or pointers in C/C++, the two questions that pop up are:

- Can we free a pointer?
- Do we have to free the pointer?

```
char * without_first_word (char *s);

int main ()
{
    char *s = strdup ("some words");
    char *wfw = without_first_word (s);
    // 1. Can we `free(s)` ?
    // 2. Do we have to `free(s)` ?
    printf ("%s\n", wfw);
    // 1. Can we `free(wfw)` ?
    // 2. Do we have to `free(wfw)` ?

}
```

In C/C++, the compiler cannot answer these questions just by looking at the function declaration (which is what the compiler usually has access to).
Developers have to read the documentation and answer these questions. Depending on how the without_first_word function is written, answer may vary.

Version 1

```
char * without_first_word (char *s) {
    int pos = 0;
    for (unsigned int i=0; i < strlen (s); i++) {
        if (s[i] != ' ') pos = pos + 1;
        else break;
    }
    return &s[pos];
}

int main ()
{
    char *s = strdup ("some words");
    char *wfw = without_first_word (s);
    // 1. Can we `free(s)` ? NO - dangling pointer
otherwise
    // 2. Do we have to `free(s)` ? YES - memory leak
otherwise
    printf ("%s\n", wfw);
    // 1. Can we `free(wfw)` ? NO - undefined behavior
otherwise
    // 2. Do we have to `free(wfw)` ? NO - undefined
behavior otherwise
}
```

Version 2

```
char * without_first_word (char *s) {
    int pos = 0;
    for (unsigned int i=0; i < strlen (s); i++) {
        if (s[i] != ' ') pos = pos + 1;
        else break;
    }
    char *wfw = strdup(&s[pos]);
    free(s);
    return wfw;
}

int main ()
{
    char *s = strdup ("some words");
    char *wfw = without_first_word (s);
    // 1. Can we `free(s)` ? NO - undefined behavior
otherwise
    // 2. Do we have to `free(s)` ? NO - undefined behavior
otherwise
    printf ("%s\n", wfw);
    // 1. Can we `free(wfw)` ? YES - memory leak otherwise
    // 2. Do we have to `free(wfw)` ? YES - memory leak
otherwise
}
```

These are only two versions, but several others are valid if the code of without_first_word changes. Moreover, an update might change the without_first_word function and render undefined behavior in our software that is using the library.

Rust solves this issue by using an ownership system, a borrow checker, and lifetime annotations.

Lifetime annotation

The first question is answered by lifetime annotations. Each reference in Rust has to be annotated with a lifetime (similar to a label).

If a function is considered a black box, input and output references have to be annotated so that the compiler understands the link between them.

```
fn without_first_word<'a> (s: &'a str) -> &'a str {
    let mut pos = 0;
    for a in s.chars() {
        if a != ' ' { pos = pos + 1; }
        else { break; }
    }
    &s[pos..]
}
fn main() {
    let s = String::from("some words");
    let wfw = without_first_word (&s);
    // drop(s) - compiler error, as `wfw` depends on s
    println! ("{}", wfw);
}
```

In this example, s is an entry reference annotated with lifetime 'a while the output reference of the without_first_word function is annotated with the same 'a lifetime. The Rust compiler understands that as long as someone uses the output reference of the function, the input reference cannot be freed.

The following example shows the power of these annotations. In this case, the input reference s and the output reference are annotated with 'a, while the input reference n is annotated with 'b. The compiler understands that as soon as the function ends, the n reference can be freed, regardless of how the function's output is used, as the output does not depend on it.

```
fn append <'a, 'b> (s: &'a mut String, n: &'b str) -> &'a
str {
    s.push_str (n);
    s
}

fn main() {
    let mut s1 = String::from("some");
    let s2 = String::from(" words");
    let title = append (&mut s1, &s2);
    // drop(s1) - compiler error
    drop(s2); // works, as `title` does not depend on it
    println! ("{}", title);
}
```

Safe Memory Management

The second question is answered by the safe memory management.

The Rust compiler enforces automatic safe memory management at compile time using the ownership system and the borrow checker.

Enforced Ownership

In Rust, each value allocated into the memory, regardless of its position (global, stack or heap) is owned by exactly one single variable. As soon as that variable goes out of scope, the compiler inserts the drop statement automatically.

```
fn main() {
    let x = Box::new(5); // allocate a number on the heap
    // use `x`
    // ...
    // `x` owns the `Box` and as soon as x goes out of
scope,
    // the compiler inserts
    // drop(x);
}
```

When declaring a function like use_string below, the compiler understands that:

The use_string function takes ownership of the value s. In other words, the value s is moved out of main into the use_string function.

The use_string function is responsible for dropping (freeing) it.

```
fn use_string<'a> (s: String);

fn main() {
    let s = String::from("some words");
    // s is moved into the `use_string` function
    use_string(s);
    // s cannot be used here anymore, it is a compiler
error
}
```

Borrow checker

The borrow checker allows the usage of a value without taking ownership of it. It formally verifies at compile time that the value towards which the reference points to is valid as long as the references is used. This prevents the dangling pointer undefined behavior.

When declaring a function like use_string below, the compiler understands that:
The use_string function borrows the value s. In other words, the value s is a reference from main.
The use_string function is not responsible for dropping (freeing) it.

```
fn use_string<'a> (s: &String);

fn main() {
    let s = String::from("some words");
    // s is borrowed into the `use_string` function
    use_string(&s);
    // ...
    // `s` can be used here
    // ...
    // the value that `s` owns will be dropped by the
compiler when `s` goes out of scope
    // drop(s);
}
```

When declaring a function like use_string below, the compiler understands that:

- The use_string function borrows the value s. In other words, the value s is a referenced from main.
- The output reference of the function depends on the input reference s (due to the lifetime annotation).
- The use_string function is not responsible for dropping (freeing) it.

```
fn use_string<'a> (s: &String) -> &'a str;

fn main() {
    let s = String::from("some words");
    // s is borrowed into the `use_string` function
    let used = use_string(&s);
    // ...
    // `s` cannot be dropped here as the borrow did not
expire
    //     the borrow expires when `used` is not used
anymore.
    // ...
    println!("used is {}", used);
    // ...
    // the value that `s` owns will be dropped by the
compiler when `s` goes out of scope
    // drop(s);
}
```

Borrow Rules

The borrow checker requires two simple rules:

- a value can have any number of concurrent immutable borrows &valueOR;
- a value can have one single mutable borrow &mut value.

This prevents concurrency issues at compile time. There is simply now way to borrow a mutable value to several threads, unless the developers use some kind of synchronization mechanism, like spinlocks or mutexes .

PROJECT ORGANIZATION

Crates and modules

Rust provides a clear way of organizing code into crates and modules.

crates are similar to C/C++ binaries or libraries. A crate can contain a single library and multiple binaries. The crate is the main element that gets published to a crates repository.

module is a subdivision of a crate, that groups together data types, submodules and functions. It is used as the scope of visibility rules. Modules can be imported into scope and renamed at imports.

Dependencies in Rust are represented by crates. In contrast to C/C++ where the libraries and namespaces do not enforce any organization, Rust provides a clear definition of what a crate is.

Crates are versioned using sematic versioning.

Most versions of C/C++ do not provide any mean of code organization except folders and header (.h or .hpp) files. While C++ 20 does define the concept of module, it is experimental in most compilers and mixes headers, namespaces and modules, which makes code organization loose.

Prevent Duplicate Names

One important aspect of code organizing is preventing name clashes.
Rust does not allow importing into scope two items, modules or data types that have the same name. Importing two items with the same name results in a compiler error.
Rust allows the renaming of items at import time.

DataTypes

Renaming applies to data types.

```
use crate_name::module1::submodule2::DataType as
NewDataTypeName;
use crate_name::module1::submodule1::DataType as
NewDataTypeName2;
```

C does not provide any means of data type renaming, which makes using two data types with the same name inside one scope impossible. The only solution is to change the source code where the date type is defined and rename it. While C++ does provide type aliasing with namespaces, it gets difficult to read.

Modules
Renaming applies to modules.
Rust does not allow importing two items into one scope.
```
use crate_name::module::submodule as crate_name_submodule;
use another_crate_name::module::submodule as
crate_name_submodule;
```

C does not allow namespaces.

Scoped imports

Rust allows importing of items limited to a scope. Developers are not forced to import items in the global scope, they can import items locally.

```
fn f() {
    use crate_name::module::DataType;
    // DataType is valid up to the end of the function f.
}

fn main() {
    use another_crate_name::module::DataType;
    // DataType is valid up to the end of the function
main.
}
```

C does not allow any kind of scoped imports, it only uses the #include directive.

Scoped imports rename
```
use crate_name::module::DataType;

fn main() {
    // DataType has te be renamed in this scope as it would
conflict with
    // the global import
    use another_crate_name::module::DataType as
DataTypeMain;
}
```

C does not provide any means of data type renaming, which makes using two data types with the same name inside one scope impossible. The only solution is to change the source code where the date type is defined and rename it.

BACKWARD AND FORWARD COMPATIBILITY

One important aspect of programming languages is backward compatibility. C/C++ provide backward compatibility by only adding features to the compiler which results in the impossibility of enforcing new good practices rules without breaking code in the dependencies.
Rust editions allow exactly this. Every three years, a new Rust edition is coming out. With every edition, Rust drops a set of features that are considered deprecated or unsafe. Every crate specifies which edition of Rust it wants to use.

```
[package]
# ...
edition = '2021'
```

The Rust compiler supports all the editions that where released. Each crate can specify which editions it wants to use. Each create is compiled independently, regardless of whether it is a dependency or the main crate.

Main crate's cargo.toml.

```
[package]
# ...
edition = '2021'

[dependencies]
dependency = "x.y.z"
dependency crate's cargo.toml.
[package]
name = "dependency"
# ...
edition = '2024'
```

In this case, the main crate will be compiled using the 2024 Rust version, that does not support some of the older features anymore, while the dependency crate will be compiled with the 2021 version of Rust, and as such may use some older features that have been dropped in the newer edition.

This assures that the build is never broken, even if the main crate uses a newer, and possibly stricter, edition of Rust.

CONTACT & FURTHER INFORMATION

Feel free to reach out to OxidOS Automotive for further information, discussions on how OxidOS is using Rust to empower embedded software developers to develop secure, portable and reusable embedded code and collaboration opportunities.

https://www.oxidos.io/

hello@oxidos.io

https://linktr.ee/oxidos