Scaling Embedded Innovation: Connecting Raptor to Rust

Xander Cesari - Pictorus

Introduction

Xander Cesari

~10 years in automotive

- FCA/Stellantis
- Rivian
- Karma
- Bollinger
- Pratt Miller Engineering

Calibration, controls development, and verification

Pictorus

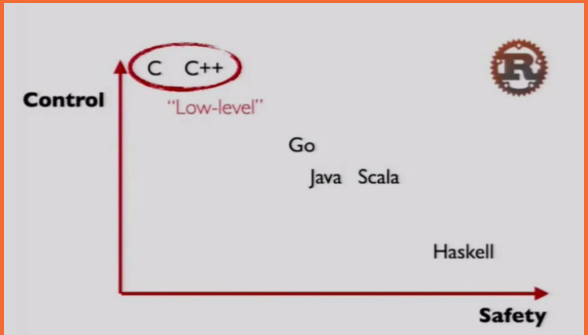
We’re a startup that makes future-proof model-based design software. We offer modern software practices to the controls industry, accelerating development velocity.

Target industries: automotive, aerospace, defense, robotics, energy

Why Pictorus Built on Rust

Promise of Correctness

What if we made a programming language designed control and performance...



...as well as safety and reliability?

Memory Safety

Memory ownership model for no memory leaks, no out-of-bounds memory access

Expressive Type System

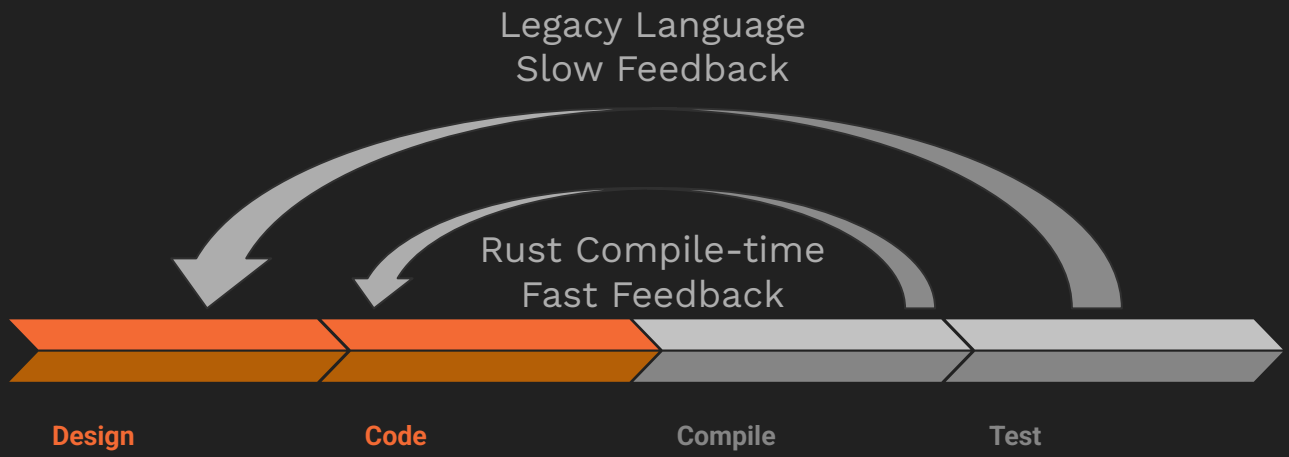
Modern algebraic types like enumerations and structs on top of a strict static type system

Modern Tooling

Integrated build system, dependency management, unit testing, and documentation generation

If It Compiles... it Runs

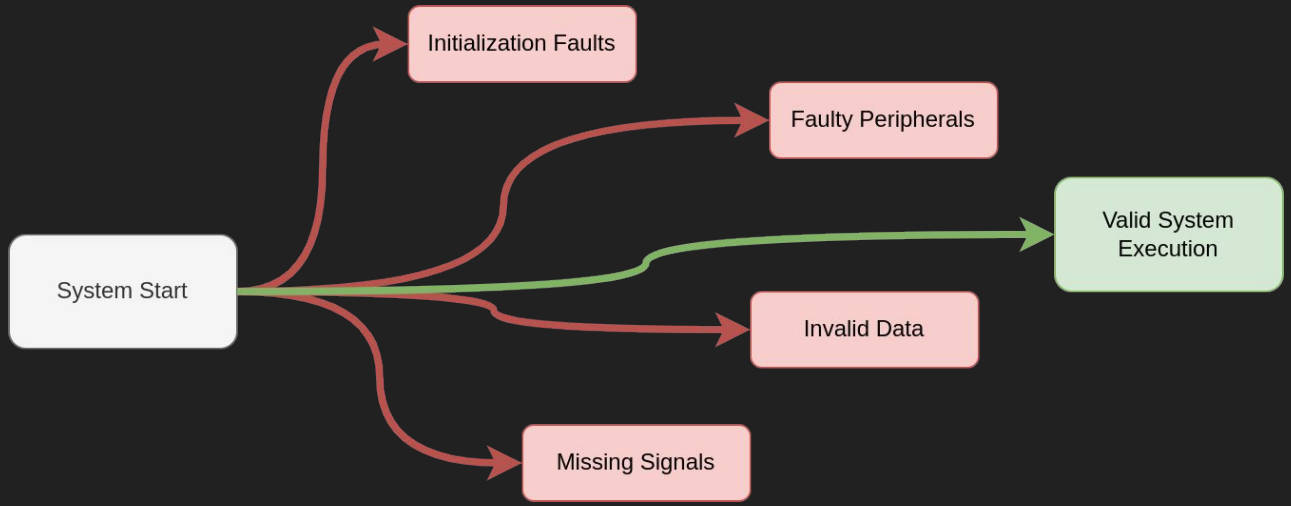
The Rust compiler errors when a program *could* fail at runtime



Make Invalid States Unrepresentable

Traditional languages offer expressive ‘happy path’ programming but no features for modeling error states

Rust’s rich type system and syntax handle the unhappy path as well: when I/O operations fail, data is absent, and other errors occur.



How Rust Solves Memory

Instead of a stop-the-world garbage collector with pauses (Python) or semi-manual memory management (C and C++), Rust uses an ownership system.

- Each piece of data is owned by a part of a program, transferring as it's moved
- Lifetimes infer when data is no longer needed and can be dropped
- Smart pointers and borrow checker enforce read and write control

Rust feature...	Borrow Checker		Lifetimes	
...solves for	Dangling Pointers	Out of Bounds Access	Use After Free	Memory Leaks

Memory Safety

“Memory safety bugs are responsible for the majority (~70%) of severe vulnerabilities in large C/C++ code bases”

- Secure by Design: Google’s Perspective on Memory Safety

70% Chrome and Android vulnerabilities

70% Microsoft vulnerabilities

68% Project Zero in-the-wild zero day exploits

Memory Unsafety is Deprecated

“Speaking of languages, it's time to halt starting any new projects in C/C++ and use Rust for those scenarios where a non-GC language is required. For the sake of security and reliability. the industry should declare those languages as deprecated.”

— Mark Russinovich (@markrussinovich) September 19, 2022

“Memory safety vulnerabilities affect software development across all industries,” said Neal Ziring, Technical Director of NSA Cybersecurity Directorate. “Working together to set clear goals and timelines in transition roadmaps to safer programming language is critical for mitigating these problems.”

You can find Rust in...

... the Linux kernel

... the Microsoft Windows kernel

... Google Chrome and Android

... every hyperscaler cloud provider’s hardware and software

... domestic robots

... cars

... space

The Rust Embedded Ecosystem

Cargo and crates.io

Cargo - the all-in-one package and dependency manager

- Pulls all dependencies, manages tooling, creates virtual environments
- Automated unit testing suites
- Generated docs from code comments
- Extensible and expandable!

crates.io - the library (“crate”) repository

- Central stabilized repository of crates
- Will only yank a crate under dire security circumstances, preventing leftpad incidents
- Can be replaced with an inter-organization repository

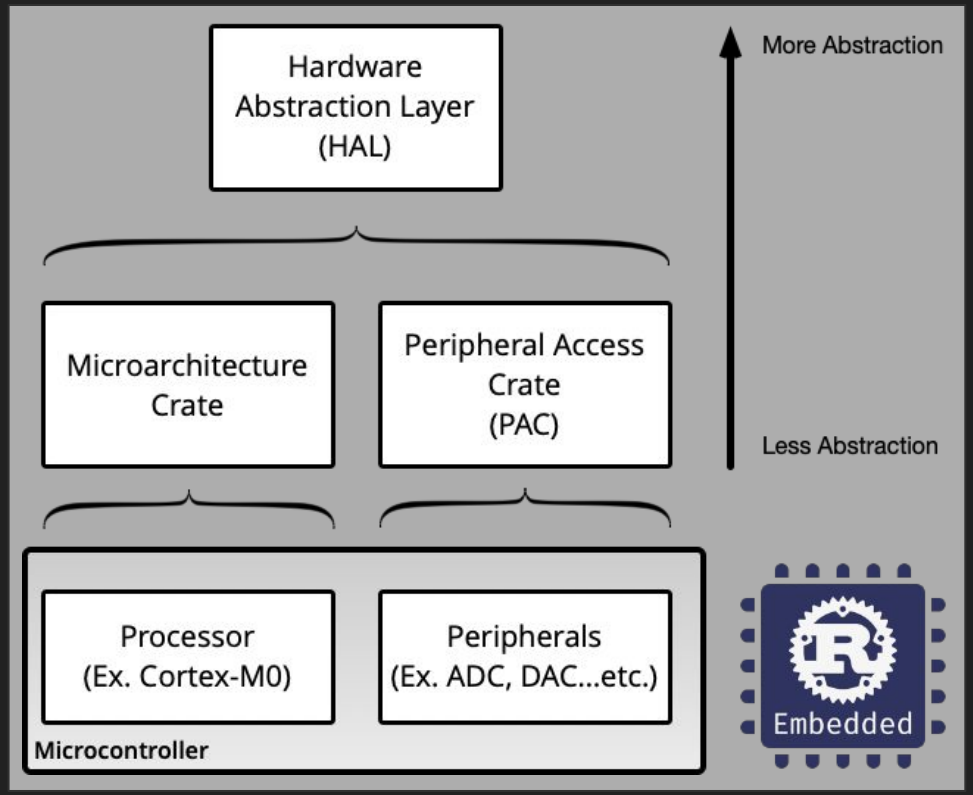
Audit your Software Supply Chain

Cargo extensions audit your entire software dependency tree for compliance

Cargo Tool	Feature
cargo-deny	Check the licenses of all dependencies
cargo-geiger	Audit for unsafe lines of code
cargo-audit	Check for reported security vulnerabilities
cargo-machete	Check for unused dependencies and trim

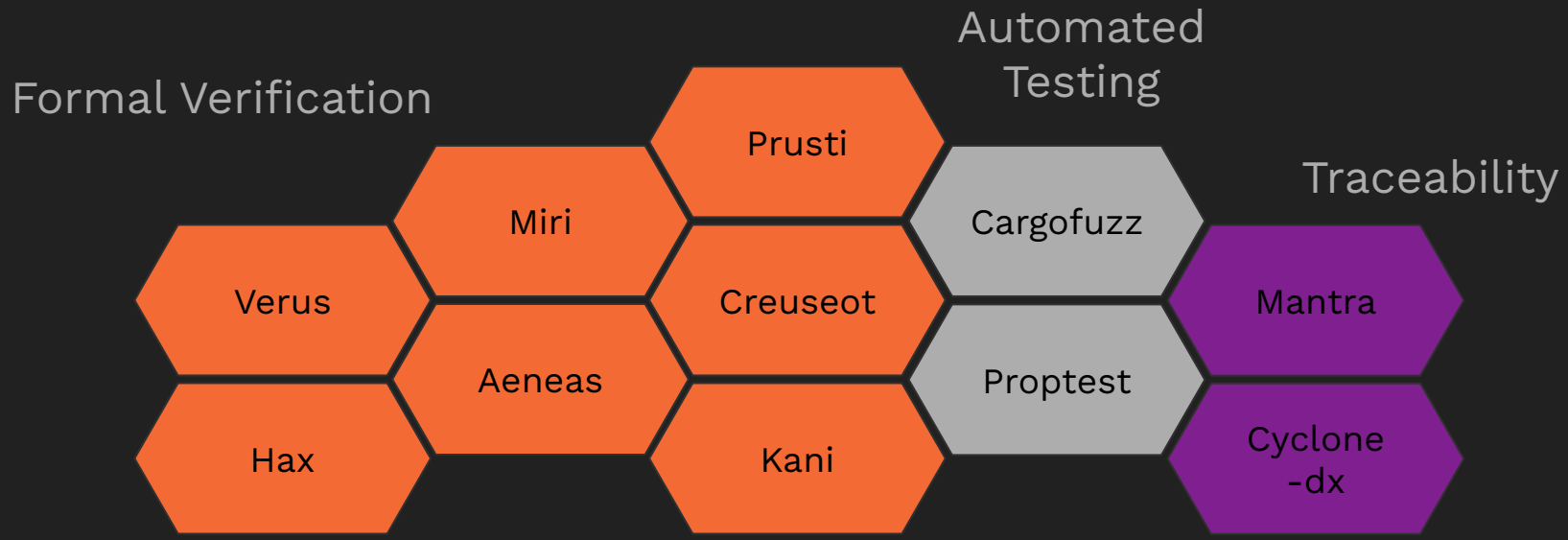
Rust Embedded Working Group

Embedded WG defines a Hardware Abstraction Layer (embedded_hal) that standardizes interfaces between hardware drivers, board support packages, and MCU crates.



Formal Verification and Testing

The Rust community's love for correctness > lots of tools for correctness

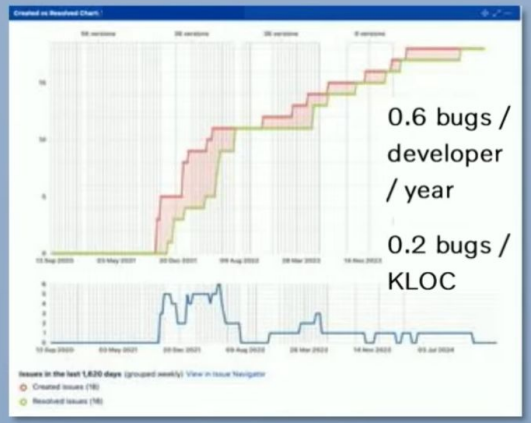


Volvo's Rust Success

VOLVO

Productivity, Quality & Maintainance

Findings from Google (Rust Nation '24):	Volvo Cars:
• 2X Productivity	✅ 2-4X!
• Higher confidence in correctness	✅
• Code is easier to review	✅
• Higher confidence in individual contributions	✅



https://www.youtube.com/watch?v=2JIFUk4f0iE

Volvo implemented a full ECU in Rust and the results were fantastic: ~100x less bugs than legacy projects

Full talk:

Rust Adoption in Automotive

Automotive OEMs

RIVIAN

VOLVO

BMW

RENAULT

woven by TOYOTA

Software Suppliers

BOSCH

Elektrobit

HIGHTEC

VECTOR

ETAS

APTIV

Hardware MCU Suppliers

infineon

Rust Startup Ecosystem

oxid[os] AUTOMOTIVE

veecle

ferrous systems

Pictorus

[NDA'd aerospace companies]

What does Rust unlock?

SECURITY

Write more secure code, future-proof against cybersecurity regulations

VELOCITY

C++ > Rust teams report increases in developer efficiency and speed.

UNITY

Focus on one language and technology that runs everywhere: from UI to embedded control.

SAFETY

Fix your code when you compile it, not when you run it.

SPEED

The speed of a compiled language means less compute, less energy, and less overhead.

TOOLKIT

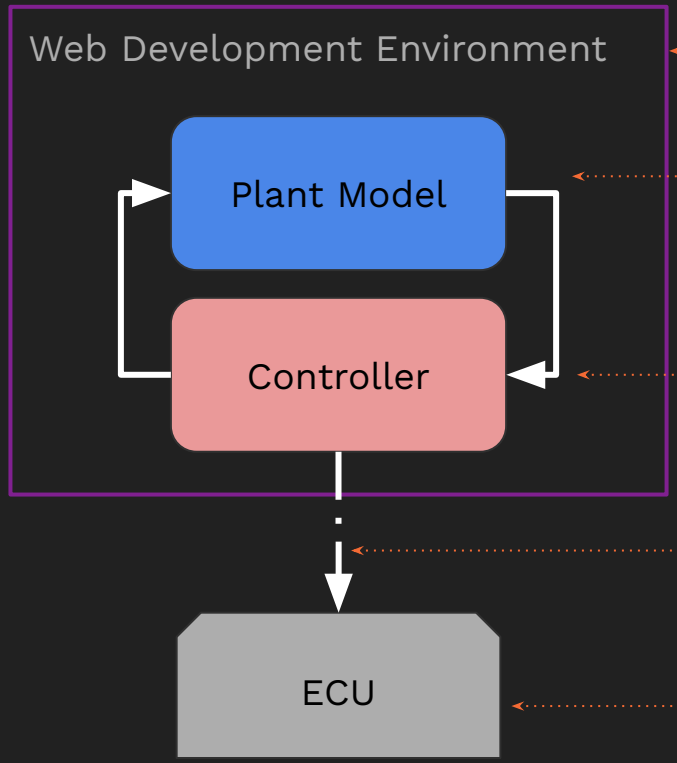
Modern dependency management, build tooling, code formatting, and compilation.

Pictorus: Future-Proof MBD

Integrating Software Engineering

Modernizing model based development by integrating the best practices and processes from the larger software engineering industry

The Pictorus Stack



Cloud-based web app

Use Python, Rust, and Pictorus blocks for expressive plant modelling

Use the entire Rust embedded ecosystem

Export app as binary, C static/dynamic lib, or Rust source code

Download a binary or deploy directly over the cloud to embedded Linux or bare metal devices

Integrating Code with Pictorus

Pure Block Based

Code in Diagram

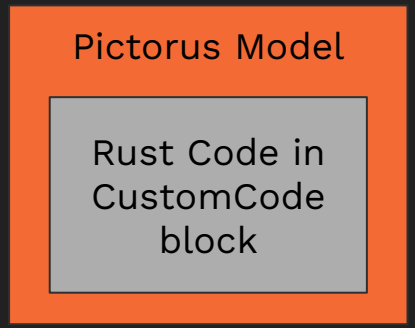
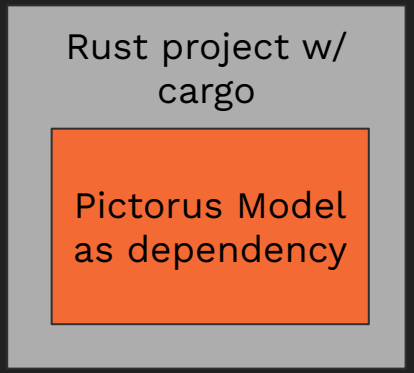


Diagram in Code



Pictorus allows a gradual shift from pure no-code visual projects to controls applications inside full embedded platforms

Git and CI/CD Integration

Release Version

CI/CD

Deploy

Create a version in Pictorus and automatically generate a Pull Request of the source code in GitHub

Trigger GitHub actions to run CI/CD automated build and tests

Reference the Git repository in a Rust project and automatically pull the latest version

Pictorus on Raptor

Beta Hardware Target: GCM111

Initial target is the GCM111

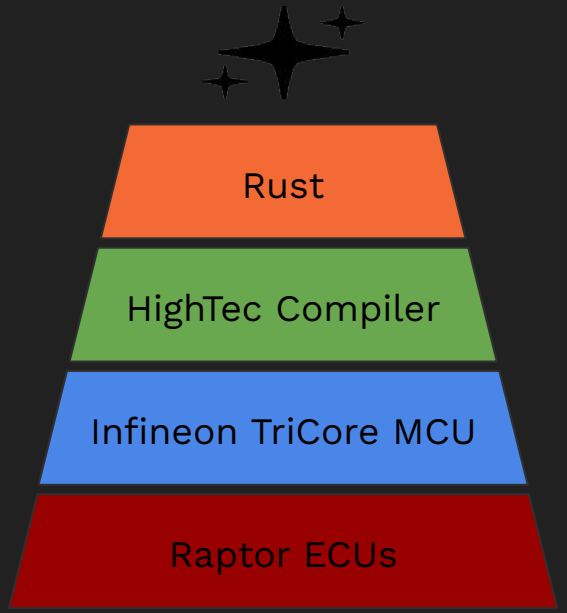
- Infineon Aurix TC367 @ 300MHz
- GPIO: 31 input, 23 output
- 20 ADCs
- Comms: 1 SENT, 2 CAN FD, 1 LIN
- Voltage: 8-16v

Future Hardware Support

Pictorus runs wherever Rust runs

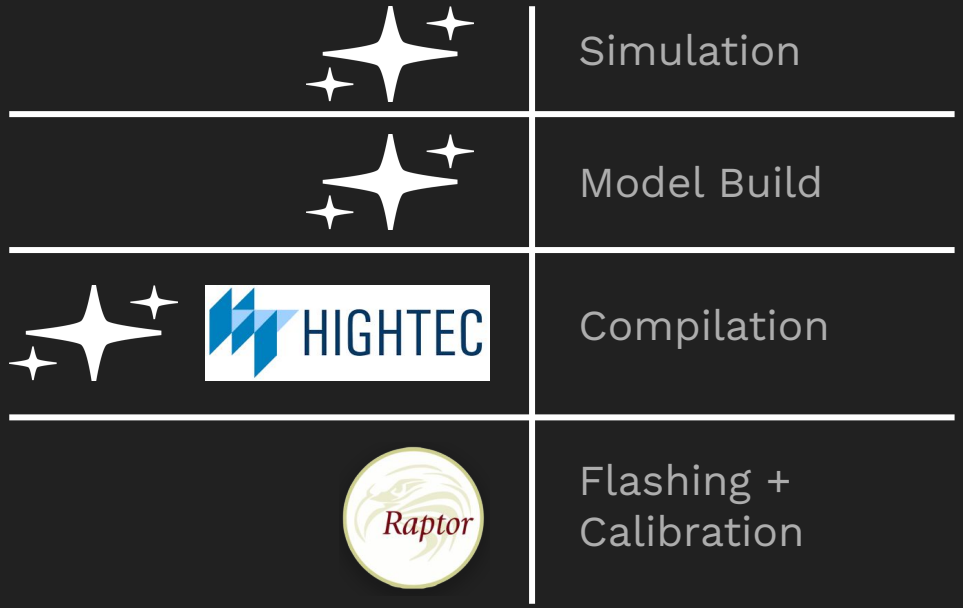
Rust is supported on Infineon TriCore MCUs by the HighTec Compiler

Many great Raptor-supported ECUs run on TriCore! Contact us with your preferred targets.



Software Support

Pictorus generates Raptor-compatible build artifacts, unlocking the power of the full Raptor suite.



Questions?

Backup

Lifetimes

```
fn main() {
    let text = String::from("Hello, Rust!"); // text created
    {   // creates a new scope
        let user = String::from("Xander");  // user created
        println!("Hello, {}", user); //
    }   // user dropped
    println!("{}", text); //
} // text dropped
```

The compiler infers when data is created and when it goes out of scope

Memory allocation and deallocation is generated as part of the program with no overhead

Borrow Checker Rule 1

```
fn main() {
    let mut s;
    {
        let t = String::from("hello");  // t created in the inner scope
        s = &t; // reference created to t
    } //  t dropped here because the scope has ended
    println!("{}", s);  // !! compiler error because s references dropped t
}
```

Rule 1: Any borrow must last for a scope no greater than that of the owner.

Borrow Checker Rule 2

```
fn main() {
    let mut s = String::from("hello");//  data created

    // two immutable references are created
    let r1 = &s;
    let r2 = &s;
    println!("{}, {}", r1, r2); // refs are used and dropped

    let r3 = &mut s; // a single mutable reference is created
    println!("{}", r3); // mutable reference is used and dropped
}
```

Rule 2: You may have one or the other of these two kinds of borrows, but not both at the same time:

- One or more immutable references
- Exactly one mutable reference

Expressive (Algebraic) Type System

Structs

```
struct Time {
    hours: u8,
    minutes: u8,
    seconds: u8,
    timezone: String,
    pm: bool,
}

impl Time {
    fn time_string(&self) -> String {
            return format!("{0}:{1}:{2} {3} TZ {4}",self.hours, self.minutes, self.seconds, if self.pm
{"pm"} else {"am"}, self.timezone)
    }
}
```

A classic object that can contain a set of data and functions/methods that operate on that data

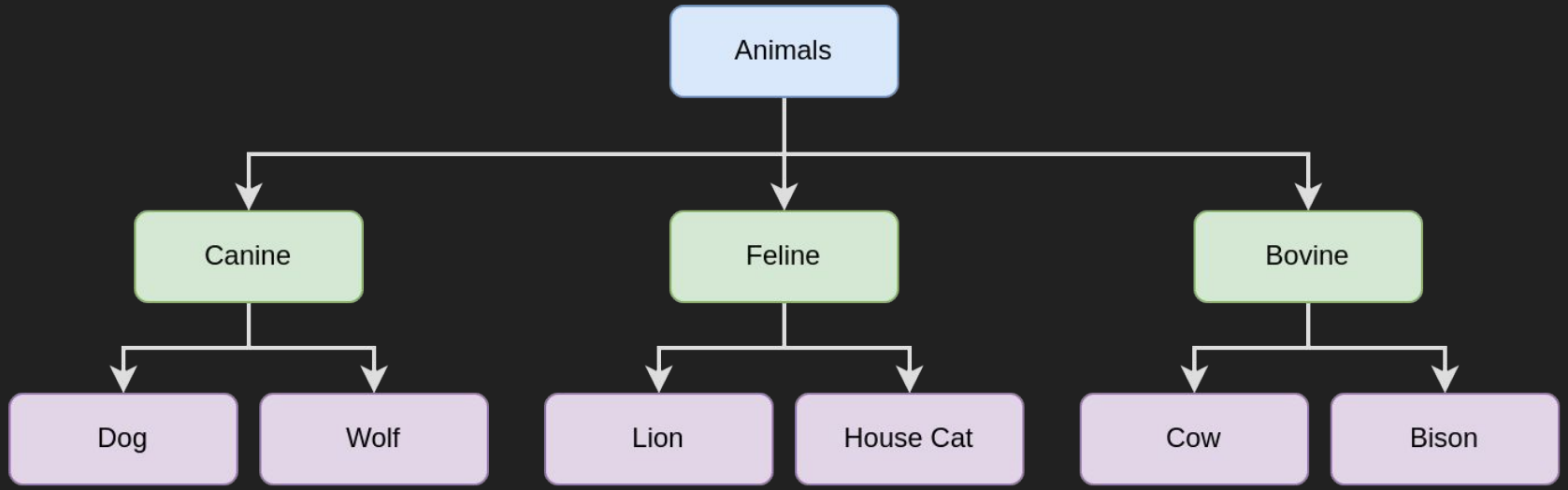
Structs Cannot Inherit

```
struct Time {
    hours: u8,
    minutes: u8,
    seconds: u8,
    pm: bool,
}

struct IntlTime(Time) {
    timezone: String,
}
```

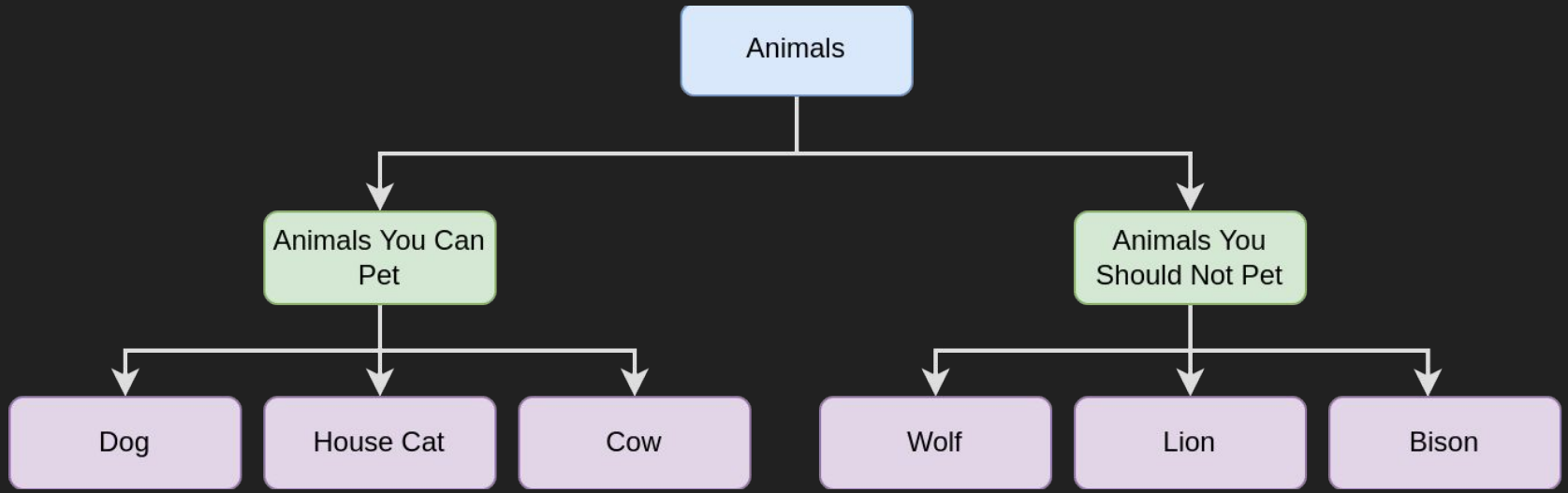
Unlike classes in other languages structs can’t inherit from other structs. So how is polymorphism achieved in Rust?

Hierarchical Inheritance Structure



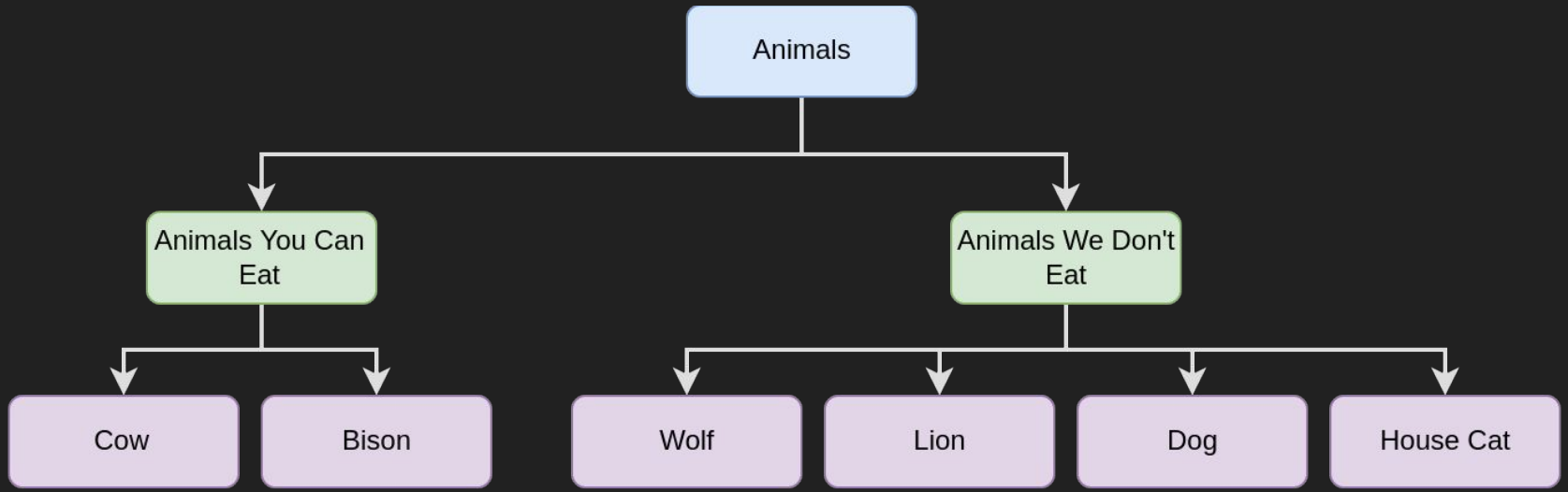
This looks reasonable but what if I care about the animals I can pet? Where should I define the pet() function?

Hierarchical Inheritance Structure



This makes sense but what if I now need to consider edible animals?

Hierarchical Inheritance Structure



This is becoming problematic!

The Trait System

```rust
trait Edible {
    fn eat(self) {}
}

trait Pet {
    fn pet(&self) {}
}
```

```rust
struct Cat {}

impl Pet for Cat {
    fn pet(self) {}
}


fn eat_something(dyn Edible) {}
```

- Define generic trait definitions
- Apply them to data structures with impl blocks
- Apply as many as you need!
- Write generic functions that take in any object that implements a trait

Rust’s Rich Enums

```
enum Sensor {
    Off,
    On(f64),
    Faulted(u8),
}
```

Rust enum’s can easily define states of devices and each state can also contain a data payload

Matching on an Enum

```
let sensor = Sensor::Off;
match sensor {
    Sensor::Off => return default,
    Sensor::On(value) => return value,
    Sensor::Faulted(fault_code) => panic!("Returned error {}", fault_code),
}
```

Matching on an enum allows Rust to get the value out or handle certain conditions

Some Favorite Enums

```
enum Option<T> {
 None,
 Some(T),
}
```

```
enum Result<T, E> {
 Ok(T),
 Err(E),
}
```

Almost every signal is actually an enumeration; it can either be there or not. The Option enum means every signal is handled correctly - even when it’s missing.

Most functions can either return successfully or they can error. The Result enum means that when a function fails the program handles the exception.

Forced Pattern Matching

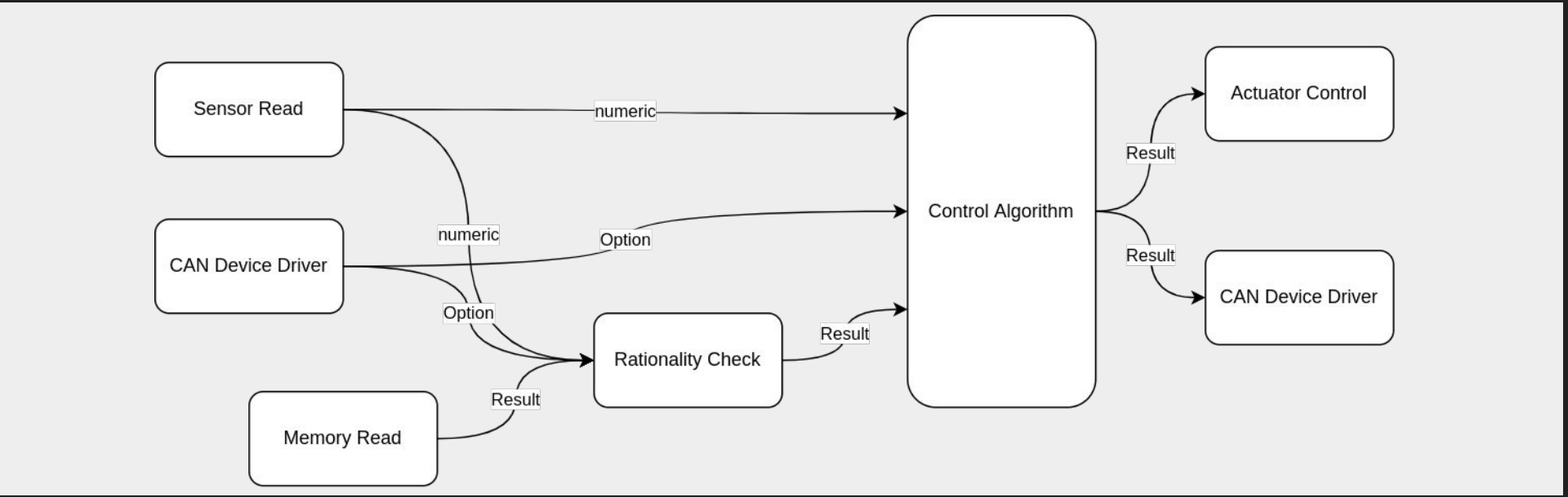
```rust
enum DeviceState {
    Off,
    On,
    Faulted,
}

let dev = DeviceState::Off;
match dev {
    DeviceState::Off => todo!(),
    DeviceState::On => todo!(),
}
```

```
error[E0004]: non-exhaustive patterns:
`DeviceState::Faulted` not covered
 --> src/main.rs:10:11
   |
10 |     match dev {
   |           ^^^ pattern
`DeviceState::Faulted` not covered
```

The Rust compiler forces every branch of a pattern to be matched, meaning the unhappy path is handled throughout the program

Rust Type System in a Control Algorithm



Result when an operation could fail, Option when data could be missing

The Certification Story

Functional Safety

Rust has been IEC-61508 certified

Most tooling is currently present for ISO-26262 certification:

- 3 certified compilers (Ferrocene, HighTec, AdaCore GNAT Pro)
- Static analysis (TrustInSoft)
- Certified core-lib with std-lib
- Requirement tracing with mantra

MISRA Rust

MISRA-Rust is coming but what will be in it? MISRA studied the C ruleset to determine what rules applied to Rust

Of the 223 MISRA-C rules, 63% did not apply due to Rust's inherent features. Another 7% only partially applied.

MISRA C:2025 Addendum 6

Applicability of MISRA C:2025 to the Rust Programming Language

March 2025