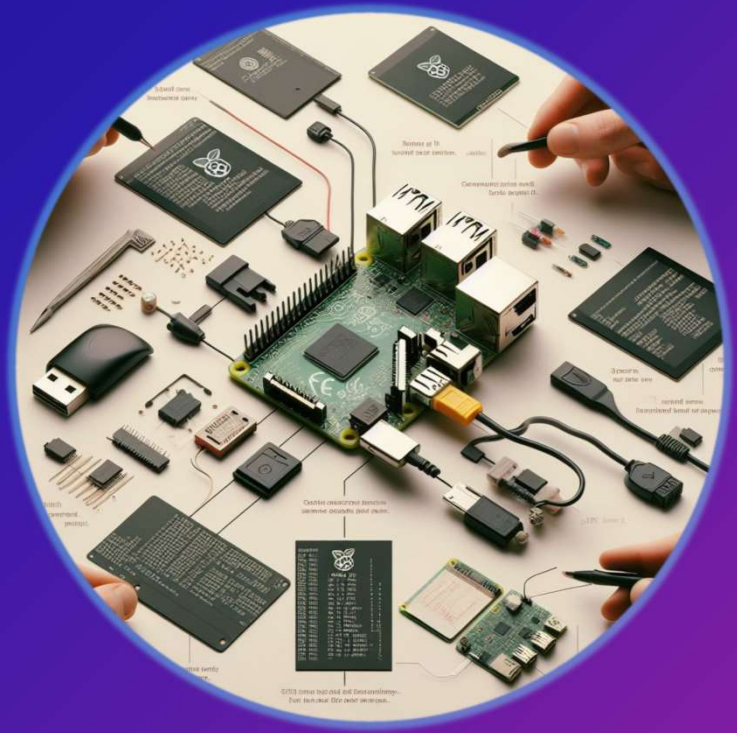
# دومین دورهمی گروه کاربران لینوکس تعبیه شده ( E-LUG )

Rust in Embedded System
(Advantages & Applications)

By: M. Moslemi AbarGhan



۲۸ فروردین ماه ۱۴۰۳

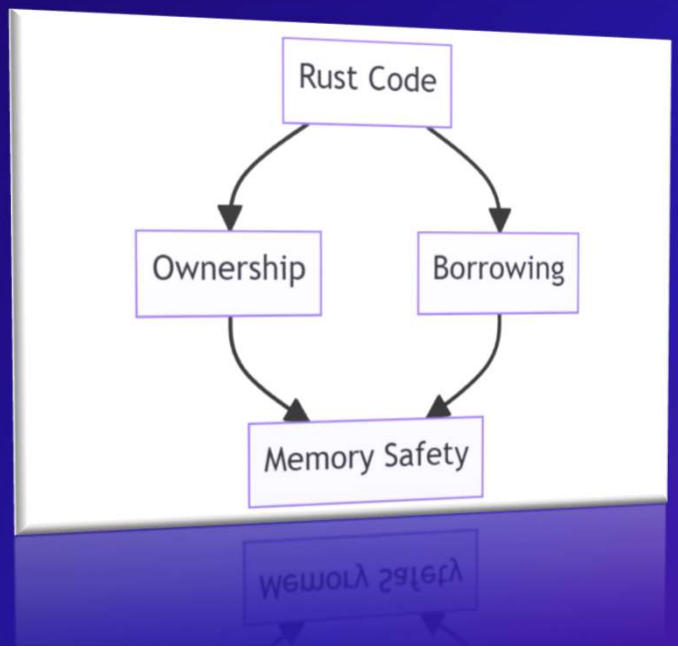

# Contents of This Presentation

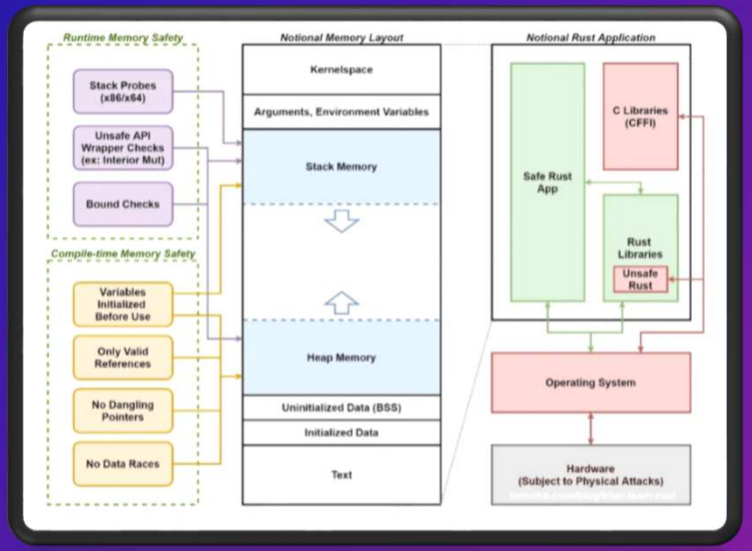
1. *Memory Safety*
2. *Concurrency and Asynchronous Programming*
3. *Zero-cost Abstractions*
4. *Static Typing*
5. *Minimal Runtime*
6. *Instructions for use.*
7. *Cargo Package Manager*
8. *Integration with C*
9. *Unsafe Code*

# Memory Safety

# *Memory Safety*

Rust ensures memory safety without sacrificing performance by using ownership, borrowing, and lifetimes.



# *Memory Safety*

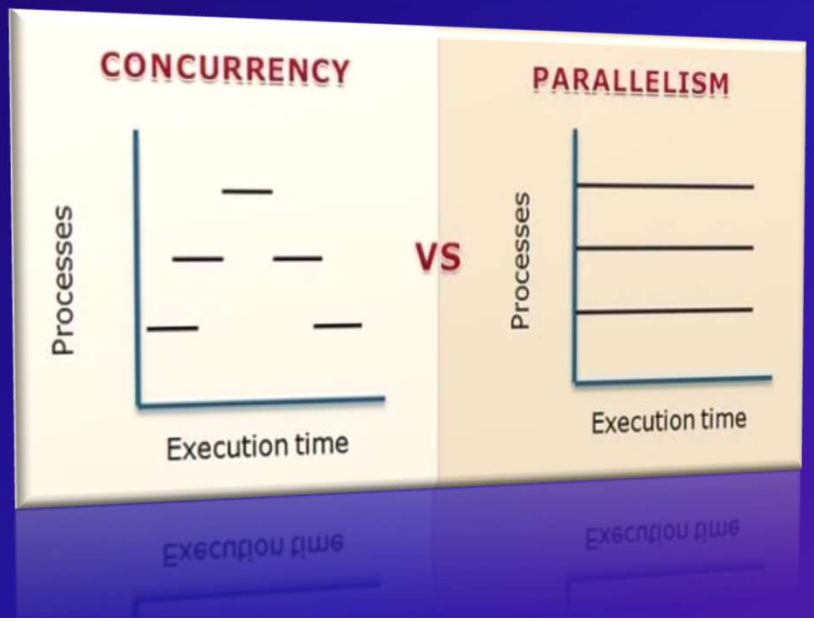
Rust ensures memory safety without sacrificing performance, which is crucial for embedded systems where memory management errors can lead to system crashes or vulnerabilities.

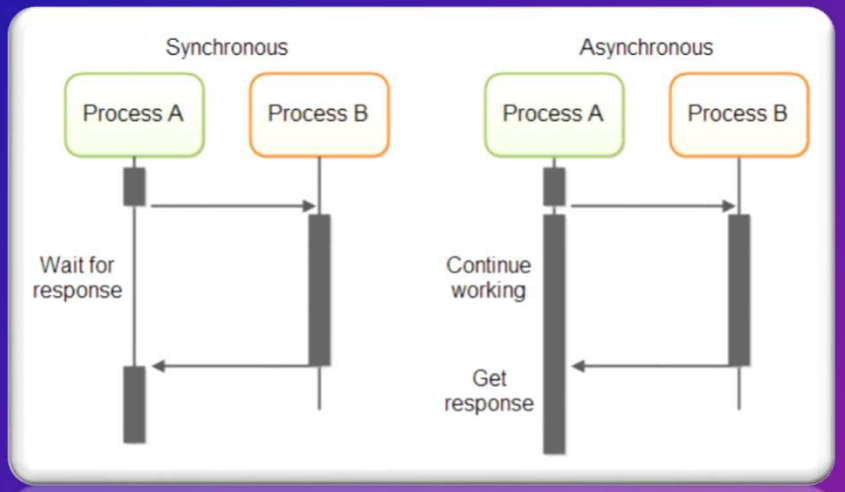
```rust
fn main() {

    let mut data = [0u8; 10];   // Initialize an array with 10 elements
    data[15] = 42;              // This will cause a compile-time error due to array bounds checking
}
```


E-LUG
iPAN
Concurrency and
Asynchronous
Programming

# *Concurrency and Asynchronous Programming*

Rust provides built-in support for concurrency with features like threads, message passing, and the `async/await` syntax for asynchronous programming.



# *Concurrency and Asynchronous Programming*

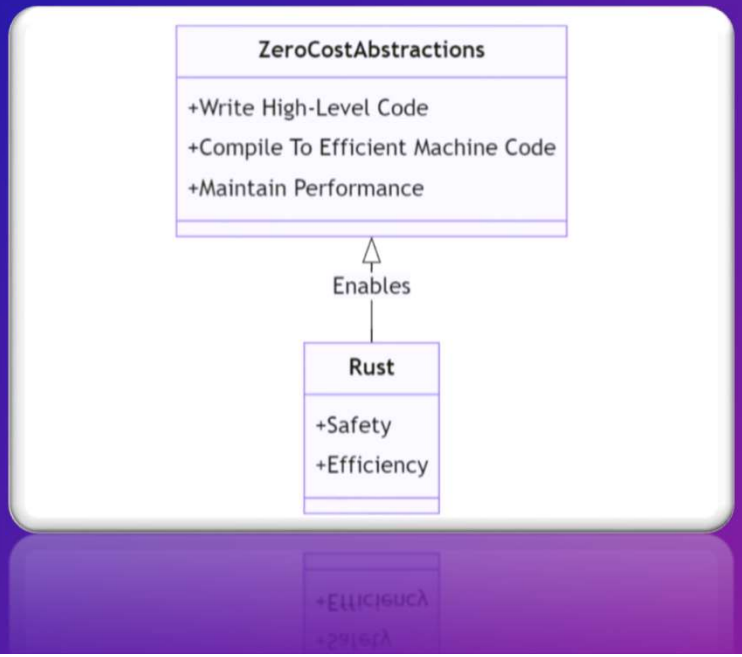
Rust provides built-in support for concurrency and asynchronous programming, allowing developers to take advantage of multicore processors and handle asynchronous events efficiently in embedded systems.

```rust
use std::thread;

fn main() {

    let handle = thread::spawn(|| {
        // Code to be executed in a separate thread
        println!("Hello from another thread!");
    });

    // Wait for the thread to finish execution
    handle.join().unwrap();
}
```

# Zero-cost Abstractions

# *Zero-cost Abstractions*

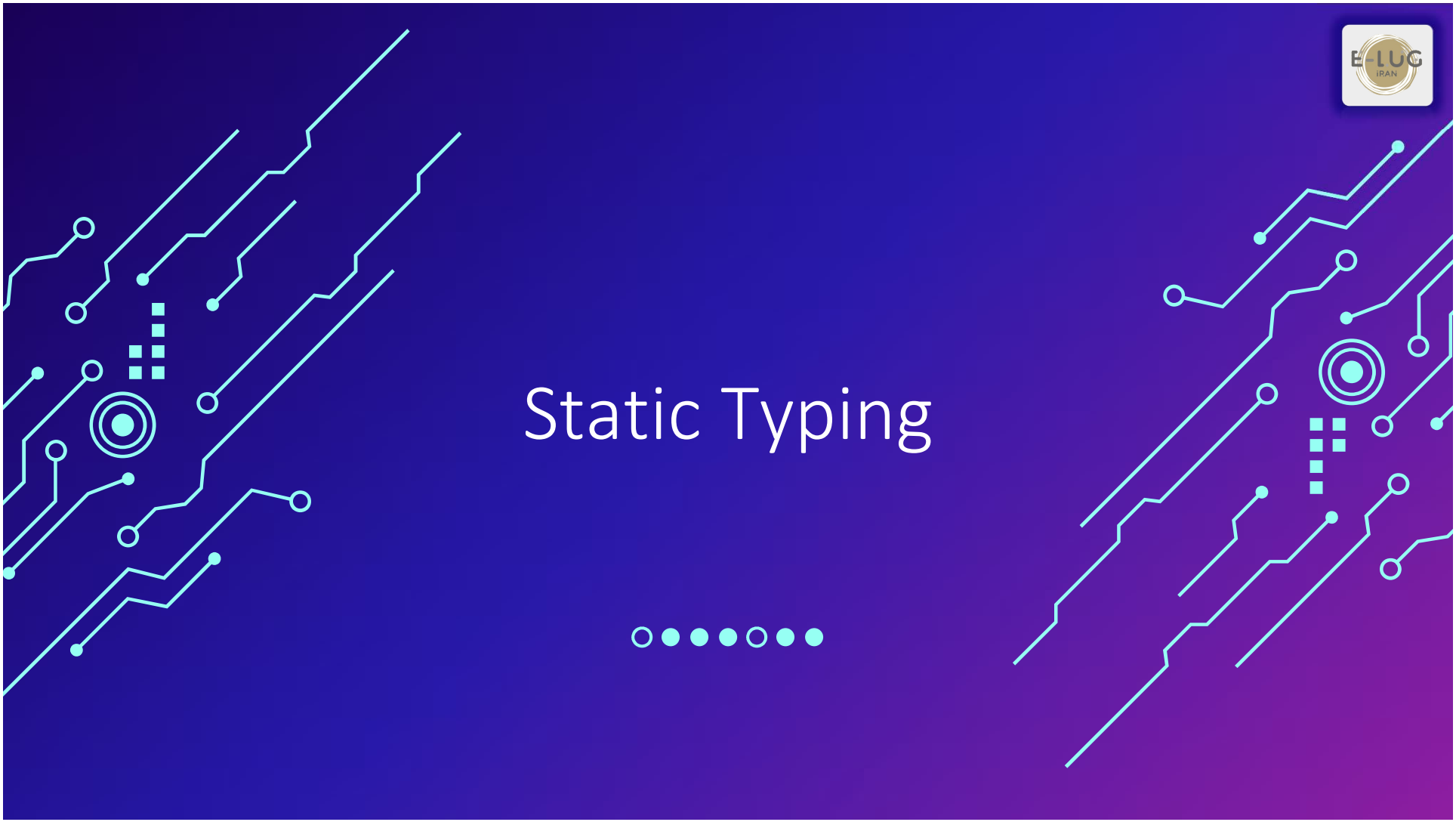
Rust allows developers to write high-level code without incurring runtime overhead, thanks to its emphasis on zero-cost abstractions.

# *Zero-cost Abstractions*

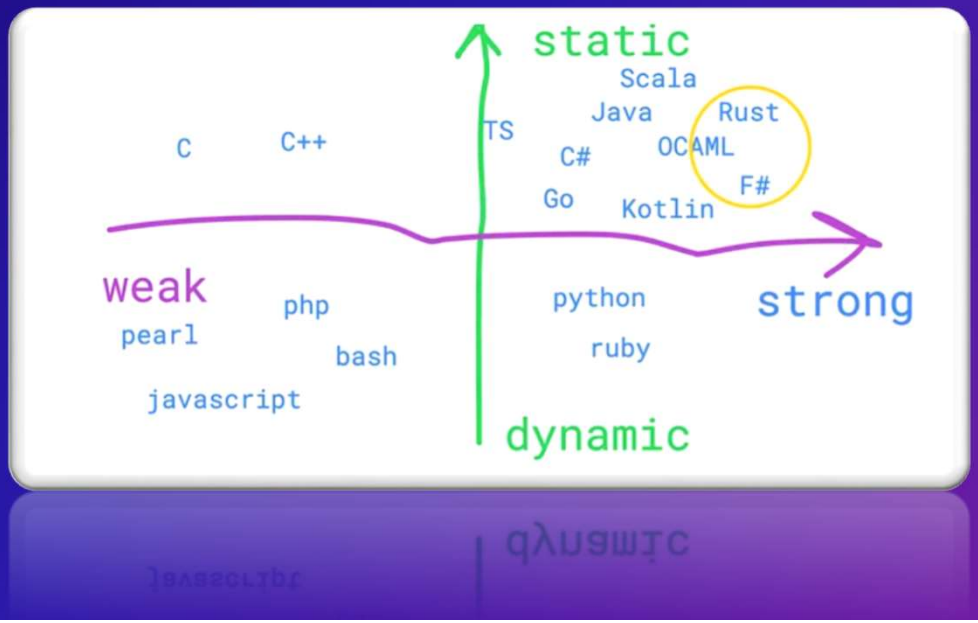
Rust's zero-cost abstractions allow developers to write high-level code without incurring runtime overhead, which is important for embedded systems where efficiency is paramount.

```rust
fn main() {
    let result = 10 + add_numbers(20, 30);  // No runtime overhead for the function call
    println!("Result: {}", result);
}

fn add_numbers(a: i32, b: i32) -> i32 {
    a + b
}
```


E-LUG
iPAN
Static Typing

# *Static Typing*

Rust is a statically typed language, which means that it checks your program's types at compile-time. But unlike other such languages, it also features a powerful feature called type inference that makes Rust more convenient and easier to use.



# *Static Typing*

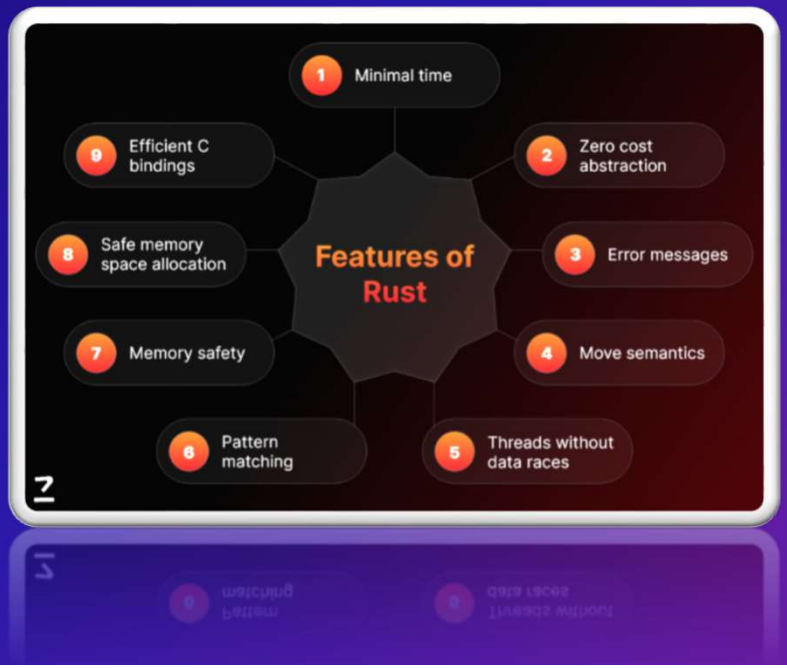
Rust's strong static typing system helps catch errors at compile-time, reducing the likelihood of runtime errors in embedded systems.

```rust
fn main() {
    let x: i32 = "hello";  // This will cause a compile-time error due to type mismatch
    println!("x: {}", x);
}
```

# Minimal Runtime

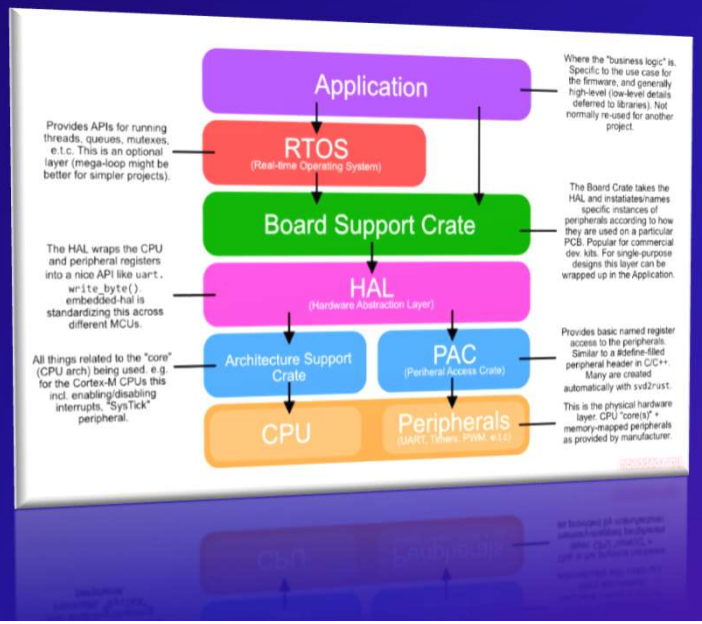
# *Minimal Runtime*

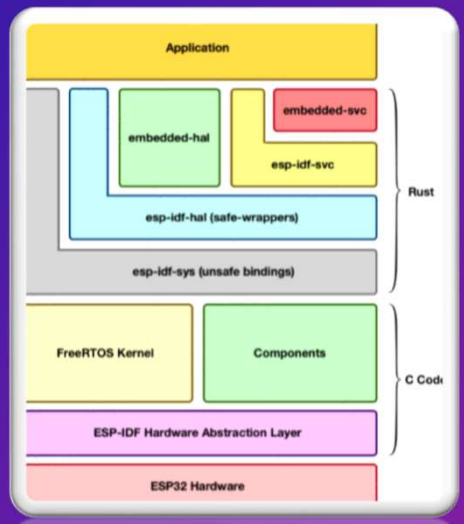
Rust makes better sense to achieve more performance and energy saving by it’s minimal runtime.

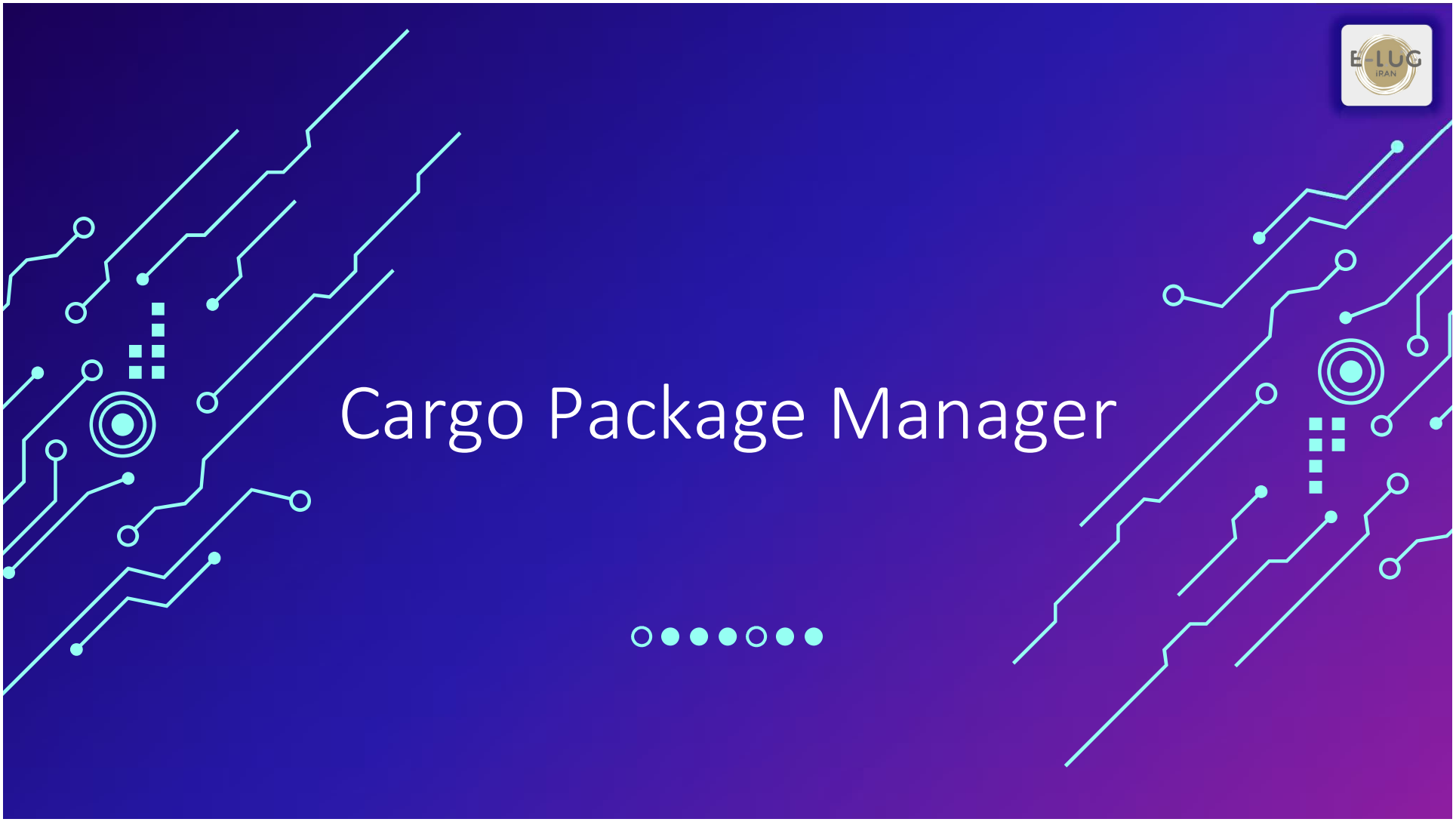


# *Minimal Runtime*

Rust's minimal runtime makes it suitable for embedded systems with limited resources, as it doesn't require a large runtime environment like some other languages.



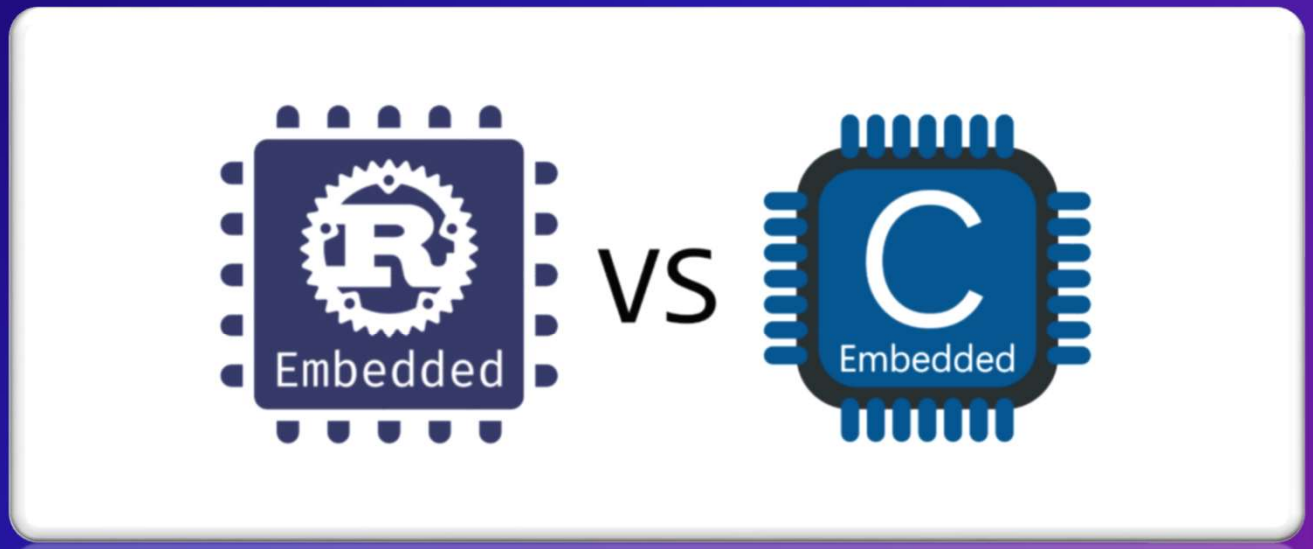

E-LUG
iPAN
Cargo Package Manager

# *Cargo Package Manager*

Rust comes with Cargo, a powerful package manager and build system that simplifies dependency management and project.

# *Cargo Package Manager*

Rust's Cargo build system simplifies dependency management and project configuration, making it easier to manage complex embedded systems projects.

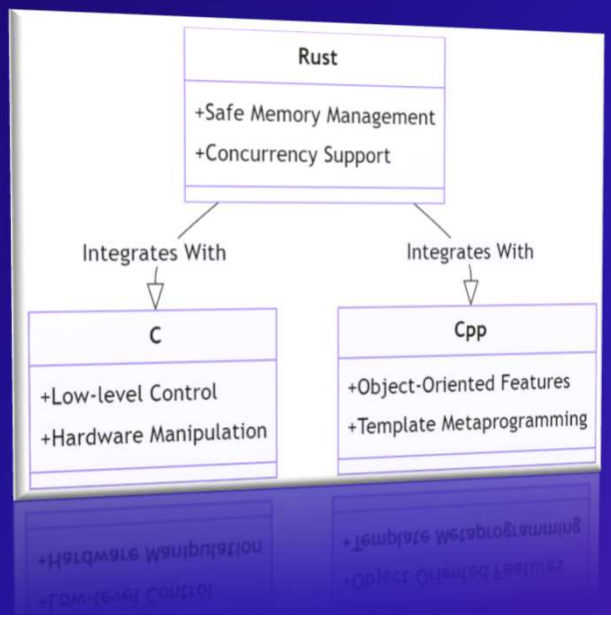


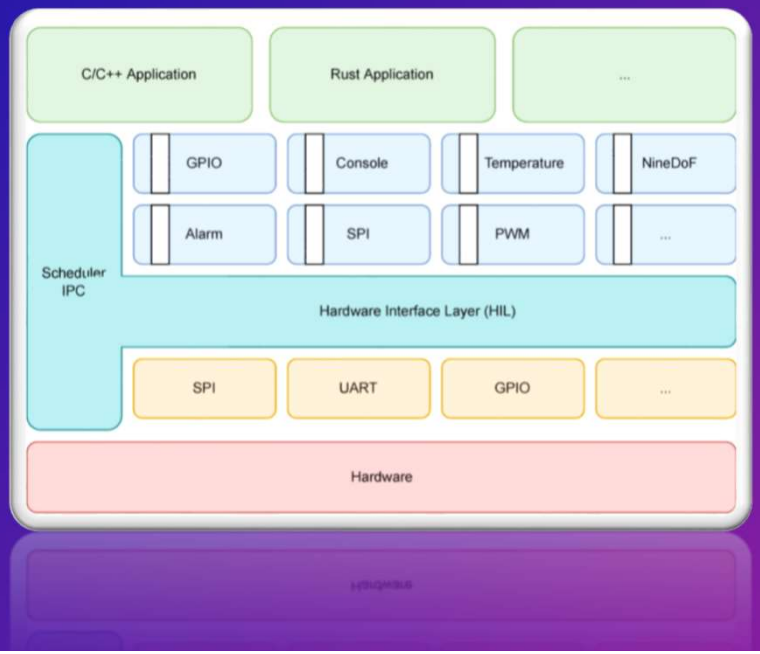
# Integration with C

# *Integration with C*

Using Rust code inside a C or C++ project mostly consists of two parts:

1) Creating a C-friendly API in Rust
2) Embedding your Rust project into an external build system

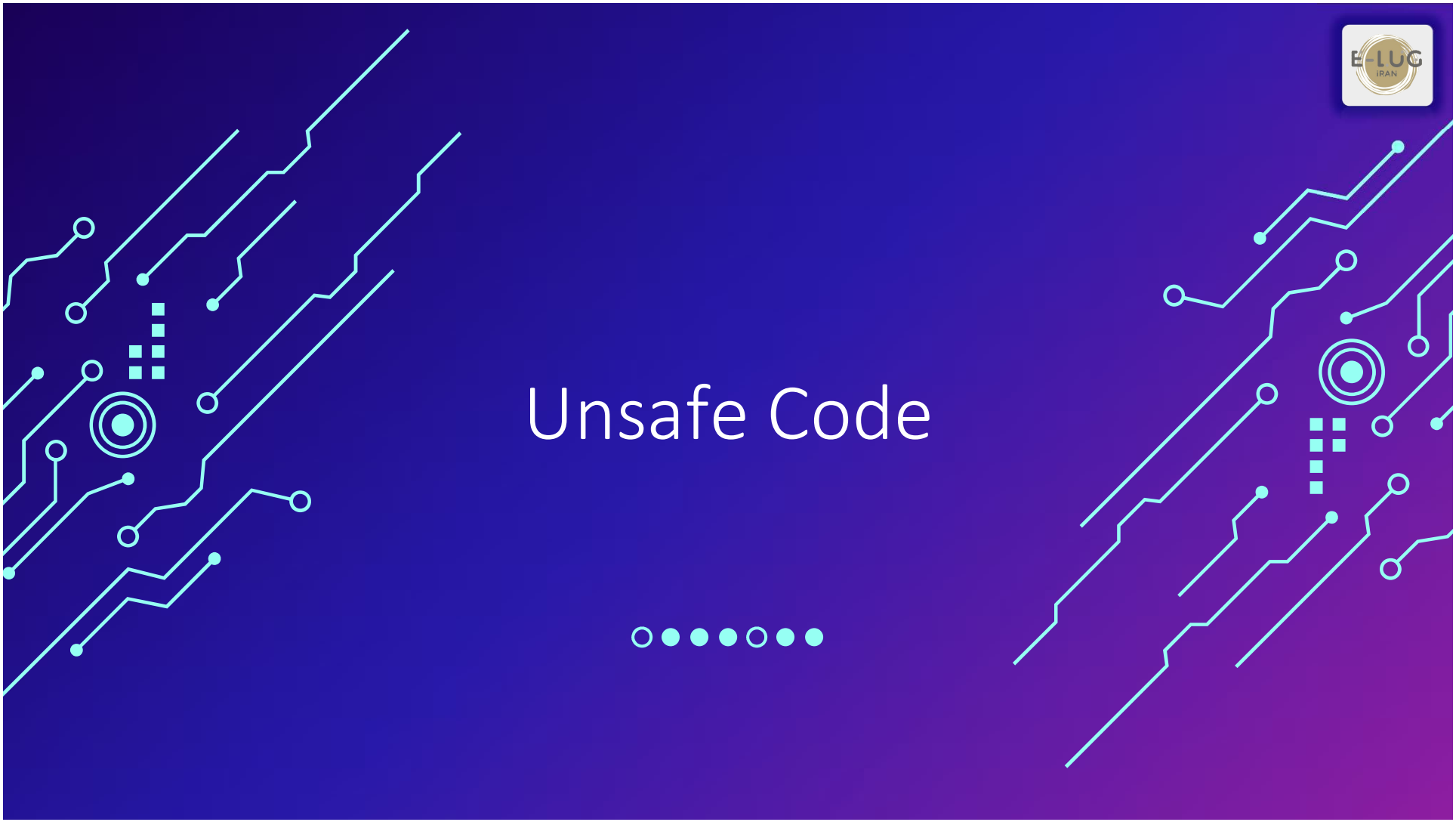


# *Integration with C*

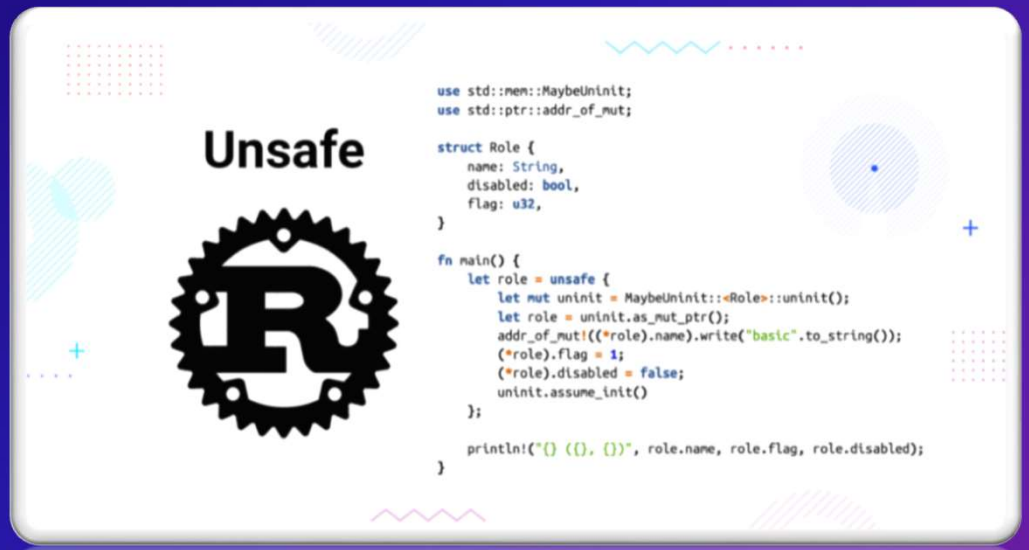
Rust can easily integrate with existing C code, which is commonly used in embedded systems development, allowing developers to leverage existing libraries and infrastructure.

```rust
extern crate libc;

fn main() {

    unsafe {

        // Call a C function from Rust
        libc::puts("Hello from C");
    }
}
```


E-LUG
iPAN
Unsafe Code

# *Unsafe Code*

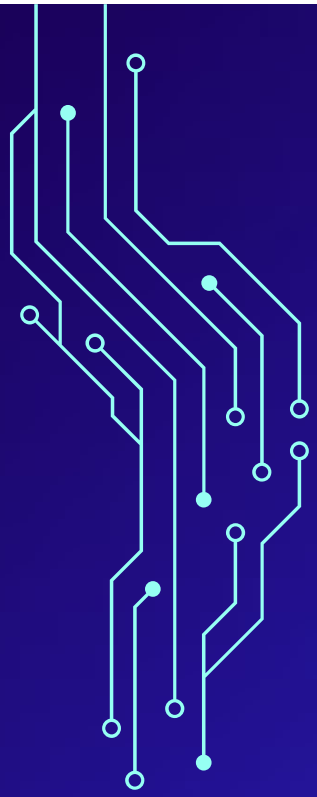
Rust allows writing unsafe code when necessary for performance or interfacing with low-level systems, but provides tools to encapsulate and isolate unsafe operations.



# *Unsafe Code*

While unsafe code should be used judiciously, Rust allows developers to write low-level code when necessary for interacting with hardware or optimizing performance in embedded systems.

```rust
fn main() {

    let mut data = [0u8; 10];

    unsafe {
        // Perform unsafe operations such as dereferencing raw pointers
        *data.as_mut_ptr().offset(15) = 42;
    }
}
```



# References

- *www.Wikipedia.org*

# با تشکر

ما را در شبکه های اجتماعی دنبال کنید: