Brandon Davis
11/15/2024
45045888

Report – Introduction to Embedded Rust Lab

Introduction

In the field of embedded systems development, selecting the right programming language and tools is crucial to ensuring efficient, safe, and maintainable code. The Rust programming language, known for its focus on memory safety and concurrency, has gained significant traction in recent years, particularly for use in embedded systems. Originally designed for high-level applications, Rust has evolved to support embedded development, offering strong performance with the added benefit of preventing common memory-related errors such as data races and buffer overflows.

This lab report explores the application of Rust in embedded systems, with an emphasis on understanding the language's unique features and how they can be leveraged to build reliable, low-level code for microcontrollers. Through this lab, the goal is to become familiar with key Rust concepts such as primitive types, structs, traits, crates, and modules, while also gaining experience with the borrow checker, which enforces memory safety by preventing data races and other unsafe behaviors.

Additionally, the lab provides an opportunity to work with the embedded-hal (Hardware Abstraction Layer) and platform-specific crates that enable direct interaction with hardware components. By learning to use the cross-platform Cargo build system, the lab will help develop skills in configuring the Rust compiler, managing dependencies, and building and deploying code to embedded systems.

Overall, this lab aims to provide a comprehensive understanding of how Rust can be effectively utilized in embedded systems, equipping students with the practical knowledge required to develop safe and efficient applications for resource-constrained environments.

Design

The code created for this lab is for an embedded system using the Raspberry Pi Pico (RP2040) with Rust, leveraging the adafruit_feather_rp2040 HAL and various peripheral drivers.

The code sets up the hardware for controlling WS2812 NeoPixel LEDs, reading accelerometer data from an LIS3DH sensor, and handling USB communication. The system continuously monitors accelerometer data to change the behavior of LED animations based on the device's orientation (e.g., different light patterns for different accelerometer values). It also supports USB communication, enabling debug messages to be sent via USB. The code includes initialization of peripherals such as GPIO, I2C, timers,

and USB, as well as animation logic using smart LEDs, while handling system events like interrupts and panic situations. The program runs indefinitely, updating the LED patterns and reporting accelerometer data over USB.

Due to the specification of the lab the code for the animations had to have this included:

- The animations must be presented as structs in their own module. Each one can then be instantiated in the application code and sent to the NeoMatrix.
- Each animation must implement the next() method, which updates the animation to the next frame when called.
- Each animation must implement the to_list() method, so that the current frame can be sent to the NeoMatrix as a list of sixty-four 32-bit color values.

Embedded Rust Animation Project for Feather RP2040 Structure

main.rs:
Configures the rp2040, i2c, usb_manager, animations
Also, it configures the accelerometer to tell its x, y, and z acceleration
and that is printed on the serial.

This repo contains 4 different Animations depending on the way the device is oriented in embedded rust for the Adafruit Feather RP2040.
This is done through using the Accelorameter and checking if +x, +y, -x, or -y is the orientation of the device. If it is in that way, it will play an animation based on that orientation

usb_manager.rs:
Code that will configure the USB peripheral as a serial port to allow for printing of formatted strings via the `write!` macro. Additionally, panic messages are sent to the serial port, and will show up when properly connected to a utility such as minicom, nRF terminal, or putty.

animations.rs:
Creates structs for 4 different animations
1. Color Cycle Pulse
2. Sprial that changes color each thim it finishes
3. Waterfall row animation
4. Bounceing a row and colum back and forth

How to run the project:

1. unizp the folder downloaded from canvas
2. Run cargo clean

3. Plug in rp2040 featherlight and its battery
4. Hold the bootloader and reset button once the device is plugged in
5. Run Cargo run flash the device by holding the bootloader and reset
8. Project should be loaded onto the board

Questions:

1. What challenges did you encounter during this project?
    - Configuration file:
        - One challenge encountered during this project was setting up and configuring the `Cargo.toml` file, especially when dealing with multiple crates and dependencies. Ensuring that all the necessary dependencies were properly listed and that versions were compatible was sometimes tricky, particularly when certain crates had specific version requirements or when some dependencies were not well documented for embedded use. Additionally, configuring the target architecture and ensuring compatibility between the Rust toolchain and the embedded platform (RP2040) required some trial and error.
    - Creating Different Animations'
        - Creating different animations for the LED matrix was another significant challenge. Since embedded development involves a lot of low-level operations, managing the pixel array, controlling the timing of each animation, and ensuring smooth transitions between states took a bit of experimentation.
    - Looking at rust create documentation to find out how some of the crates work
        - Rust's documentation for embedded development, while comprehensive, can sometimes be sparse or fragmented. For example, understanding how to work with certain crates (such as `usbd_serial`, `usb_device`, or `embedded-hal`) required looking through multiple resources and examples to figure out how to wire them up properly. Sometimes, I needed to refer to GitHub to understand edge cases or how to integrate specific functionality into the project. While Rust's documentation for basic features is usually clear, getting deep into embedded-specific libraries required patience and research.
2. Do you have any suggestions for improvement to the structure of this project?
    - One of the suggestions I have for improvement for the structure of this project
        - The .toml File:
            - One improvement for the structure of the project could be related to organizing and managing the dependencies in the `Cargo.toml` file. Currently, the `Cargo.toml` could be better organized by

grouping related dependencies together or separating them into different categories (e.g., core, hardware, USB, animations). This can make it easier to manage the project's dependences, especially when adding or removing crates. Additionally, using features like optional dependencies for things like debugging or testing could make the project more flexible. Another potential improvement would be to ensure the file is consistently updated with the latest versions of crates and that any unused or redundant dependencies are removed from the demo or documentation in the lab.

- Also, as the project grows in complexity (e.g., adding more sensors or peripherals), splitting the project into multiple crates (such as separating the USB functionality or the animation code) might improve maintainability. This would make it easier to test and reuse parts of the project across different applications.

3. What are your takeaways from Rust in comparison to other widely used embedded languages (namely C/C++)?
    - Rust Crate Files are very useful
        - One of the standout features of Rust compared to C/C++ is the “cargo package manager” and the `Cargo.toml` configuration. The ease with which dependencies are handled in Rust is extremely powerful. You can specify exactly which version of a crate you need, and Cargo automatically handles downloading and compiling dependencies. This saves a lot of time compared to C/C++’s dependency management, which often requires manual installation or handling libraries and tools separately. With Rust, crates like `embedded-hal`, `usb-device`, and `usbd_serial` provide high-level abstractions for hardware control, making it easier to integrate peripherals without manually handling every register or low-level detail.
    - Its very high level compared to C/C++
        - Rust provides a high-level abstraction that is comparable to C++ but with far more emphasis on “safety” and “concurrency”. The borrow checker ensures that issues like “null pointer dereferencing”, “buffer overflows”, and “data races” are caught at compile time, which is a significant improvement over C/C++. This makes writing safe code easier, especially in embedded systems where memory management and concurrency are often the source of hard-to-find bugs. With C/C++, I would often need to manage memory manually, worry about undefined behavior, and deal with issues that arise due to lack of type safety.

In addition, “Rust’s error handling model” (using `Result` and `Option` types) forces developers to handle potential errors explicitly, which leads to more robust and predictable behavior. In contrast, C/C++ relies on traditional error codes or exceptions, which can sometimes lead to less predictable and harder-to-maintain code.

- Rust’s Ecosystem for Embedded Development:
  - While the ecosystem for embedded Rust is still maturing, it is already competitive with C/C++ in many areas. Crates like `embedded-hal` offer a unified way to access hardware functionality across different platforms, making it much easier to port code between platforms. The ecosystem is still growing, and certain areas (e.g., real-time systems or advanced hardware control) may not be as mature as C/C++, but the rapid pace of development in the Rust embedded community suggests that it will continue to improve.

  - Overall, Rust provides a safer, more modern approach to embedded systems development compared to C/C++, while maintaining performance close to C’s level. Its tooling, dependency management, and strong type system make it a very appealing choice for embedded systems projects, particularly for developers who value safety and concurrency.

Conclusion

In this project, I explored the potential of using Rust for embedded systems development, specifically on the RP2040 platform. Through the implementation of various hardware interfaces, including LED animations, USB serial communication, and I2C sensor interaction, I gained experience in working with Rust's ecosystem and its crates.

Rust's memory safety features and concurrency model can provide significant advantages over traditional embedded languages like C and C++. The borrow checker, error handling, and strict type system helped minimize common issues such as memory corruption and undefined behavior, leading to more reliable and maintainable code; the compiler would recognize issues and recommend how to fix them very effectively.

Additionally, Rust's crate ecosystem— with libraries like embedded-hal and usbd_serial— greatly simplified hardware interaction and USB communication, making the development process smoother.

Particularly, configuring the Cargo.toml file for complex dependencies and understanding how to properly implement certain crates where some challenges that where faced. Also creating the LED animations and fine-tuning their performance on a resource-constrained platform was difficult. However, these hurdles helped deepen my understanding of embedded development with Rust and how to balance high-level abstractions with low-level hardware control.For future improvements, refining the project structure by organizing dependencies more effectively in the Cargo.toml file.

As Rust's embedded ecosystem continues to evolve, it will become an even more powerful tool for embedded systems development. Ultimately, this project demonstrated that Rust is a strong alternative to traditional embedded languages, combining safety, performance, and modern tooling. With continued improvements in its embedded libraries and a growing community, Rust is poised to be a compelling choice for developers working on embedded systems, especially for projects that demand high reliability and performance.