UF UNIVERSITY of FLORIDA
Herbert Wertheim College of Engineering

Computer Engineering
CEN3907C

Introduction to Embedded Rust

Overview

As Computer Engineers, we have many options when it comes to choosing tools to build our products. The Rust programming language is a tool that was developed approximately 10 years ago, with a focus on memory safety and a large set of features. While not initially created for embedded systems, the last few years have seen active development and improvement to Rust on small microcontrollers, such that it is now a viable tool to use in practice.

Learning Objectives

- Gain a working knowledge of the Rust programming language on embedded systems.
  - Understand the **basic syntax, structures, and features of Rust**.
    - Primitive types, Structs, Traits, etc. Functions, Modules
  - Understand how the **borrow checker** prevents simultaneous mutable access to memory.
  - Understand the **requirements and limitations** of using Rust on bare metal.
  - Understand the *embedded-hal* abstractions and the board specific *hal* and *pac* crates.
- Learn how to use the cross-platform *cargo* build system.
  - Configure the compiler and the target platforms.
  - Specify and fetch external *crates*.
  - Build and load an executable.
  - Interpret output from the borrow-checker and compiler.

Resources

The following resources, in addition to the tips and hints below, should provide you with enough information to complete this assignment. If you have not done so already, follow the instructions in the HW Intro document to compile and load the Rust example onto your board.

- The Embedded Rust Book
  - A comprehensive overview of Rust on Embedded Devices.
- adafruit_feather_rp2040
  - This *crate* exposes some pre-configured pins for the Feather RP2040, as well as re-exports the rp2040_hal (see below).
- rp2040_hal
  - This *crate* contains the hardware abstraction layer for the RP2040, as well as re-exporting the rp2040_pac (peripheral access crate). The hal gives you access to all of the on-chip peripherals such as gpio, i2c, pio, etc. Most of the traits in *embedded_hal* are implemented for the peripherals, allowing for use with a variety of external components (i.e. the accelerometer over i2c, and the NeoMatrix via pio).
- embedded_hal
  - This *crate* contains a collection of *traits* (a trait is a set of methods) that enables authors of hal crates and library crates to share a common abstraction over peripheral drivers. For example, the *embedded_hal* defines a collection of blocking i2c traits (found under embedded_hal::blocking::i2c) such as Write and Read. This allows drivers for specific devices (such as the LIS3DH) to require that the types they are built from implement certain functionality and can work across devices.
- ws2812_pio
  - This *crate* generates the required signal to drive the ws2812 LEDs via pio. It implements the SmartLedsWrite trait, making it easy to drive a string of LEDs with a list of RGB8 color values.
- smart_leds
  - This *crate* contains useful types, functions, and Traits for working with addressable RGB LEDs such as the ws2812.
- LIS3DH Datasheet
  - Provides register mappings and definitions for the accelerometer.
- Code walk-through video
  - Provides a walk-through of a partial solution to this assignment.

Specification

For this lab, you shall implement an application in embedded Rust that displays four different animations on the NeoMatrix display, depending on the orientation of the device (+x, +y, -x, -y). What the animations display is up to you, but they must meet the following criteria:

- 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.

Suggested Approach to Development

Learning a new programming language can be difficult – especially one with features and build tools that are unfamiliar. The following order of operations should help your learning and development progress as smoothly as possible:

- Review the Drivers for Embedded Systems lab (even if you did not implement it), in order to gain an understanding of the requirements to drive the NeoMatrix and interface with the accelerometer (i.e. pinouts, power enable pin, certain GPIO function limitations).
- Clone, build, and run the provided Rust application from the HW Intro document. Research and experiment with modifying this code, as well as the Cargo.toml file and the .config/cargo file. Try to understand how the build system fetches crates, how you import those crates into the namespace of your app, and the basics of the *embedded_hal* and *rp2040_hal*.
- Create a new project to start your development. Copy over the relevant files, and create your main.rs and animations.rs files. Try implementing the *Ws2812* struct from the *ws2812_pio* crate with the proper pin, and then try sending it some data using a list of *RGB8* values from the *smart_leds* crate.

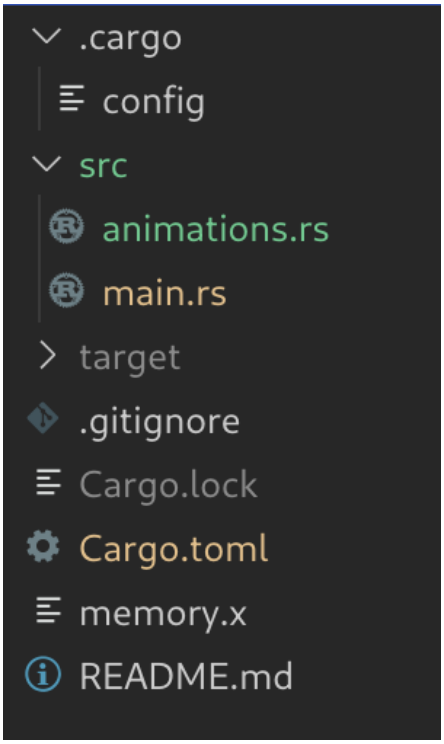
Submission

Your submission will be composed of the following elements:

- Your complete source code as a **ZIP archive on Canvas**. This will be extracted into a folder, then compiled and loaded onto the board by running *cargo run*.
- **Report in PDF format on Canvas**.

ZIP Archive

Your ZIP archive should have the following directory/file structure:



- The README.md file should be personalized and have specific explanations about your project's structure, as well as instructions on how to build and load your code.

Report

When filling in the report template, please include responses to the following:

- What challenges did you encounter during this project?
- Do you have any suggestions for improvement to the structure of this project?
- What are your takeaways from Rust in comparison to other widely used embedded languages (namely C/C++)?