A Systems-Level IDE for Embedded Development

Carsten Thue-Bludworth
University of Florida
Gainesville, FL, USA
carstentb@ufl.edu

ABSTRACT

This paper outlines the design, status, and impact of an integrated development environment for developing embedded firmware in the Rust programming language. Named Iron Coder, this IDE has a focus on *systems-level* development, in which the programmer can graphically define the hardware architecture of their system, and the IDE will generate boilerplate code and hints that relate to each hardware aspect of the system.

With impacts that aid accessibility for newcomers to embedded Rust, as well as enhance the rapid prototyping of embedded systems, Iron Coder has the potential to accelerate the usage of Rust in the fourth industrial revolution.

BACKGROUND & MOTIVATION

Modern open-source hardware ecosystems such as Adafruit's Feather boards (Figure 1), Sparkfun's MicroMod boards, and Raspberry Pi's single-board computers, among others, are built around modular components, allowing engineers and makers to mix-and-match main boards with a variety of peripheral boards to make systems that can perform many different functions. Programming languages such as Arduino and CircuitPython have aimed to reduce the barrier to entry for writing firmware for these types of systems by formulating consistent ways of writing code for a multitude of hardware boards, integrating hardware and library management tools into their ecosystems, and allowing for the community-led expansion of supported boards and libraries. The combined evolution of this hardware-software ecosystem over the last two decades has radically shifted the embedded development space from an expensive and intellectually challenging professional endeavor, to one of today's most accessible realms of computing [2].

The rapid success and growth of hobby development boards has had a significant impact on computer science and engineering education, with physical computing used to inspire and excite new developers [3]. Furthermore, many of these boards contain capabilities that also position them as feasible targets for professional engineers to develop and test products and systems. The availability of low-cost, high-performance development boards is accelerating the development of novel technologies and finding use in a variety of industries such as the IoT, industrial automation, connected infrastructure and agriculture, and edge-AI [4]. This so-called "fourth industrial revolution" has the potential to significantly impact the future of our world [1].

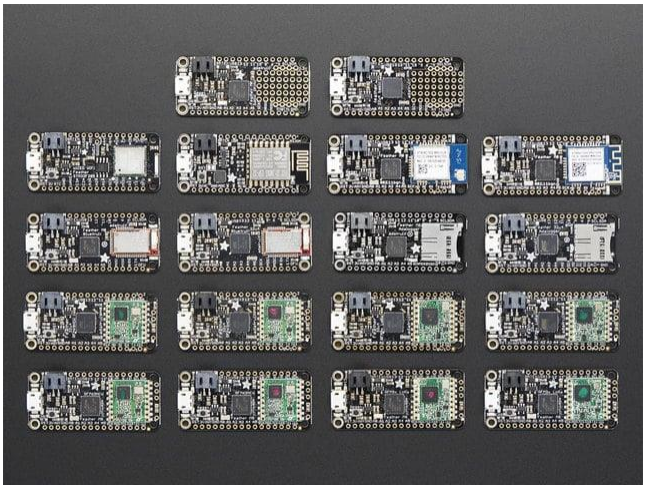

Figure 1 - Adafruit Feather Boards

As a rapidly growing field that is likely to highly influence our lives, the safety, robustness, and maintainability of these systems is of primary importance. For this reason, Rust is positioned to be an important programming language for the next generation of computer technologies. Rust is a modern systems programming language with a focus on memory safety and performance; however, it offers other conveniences, such as *cargo,* a centralized build system and dependency manager, and *rustup*, a centralized toolchain management utility which allows for the cross-compilation

of code for a variety of target architectures. Furthermore, Rust integrates styles and ergonomics from a variety of programming paradigms, contributing to its versatility as a low-level, performant language that also has great zero-cost abstractions and high-level constructs [6].

Iron Coder is inspired by the current open-source, modular hardware-software ecosystem, and aims to bring Rust into the minds of new embedded developers. The embedded Rust space is maturing and seeing growing usage, but remains as a space for those who are already familiar with embedded work or Rust programming. With a stylized, hardware-centric interface, Iron Coder aims to be visually interesting, and always keep the *system* at the center of the development process. By providing a backend to generate project templates and hardware-initialization code, Iron Coder helps overcome some of the difficulties that entry-level developers might face when first starting. Iron Coder aims to allow more people, especially novices, to develop embedded firmware in Rust, with important impacts on the future of engineering education and embedded systems.

DESIGN

Iron Coder is written in pure Rust, utilizing the standard library and a collection of publicly available crates. From the start, a set of goals and ideals drove the development, including cross-platform support, smooth performance even on low-power devices, robust/crash-free operation, and community extensibility. Utilizing the strengths of Rust as a performant, compiled language with an excellent build system, along with the benefits of its robust error-checking features and general compile-time safety, these goals were able to be realized. The project is hosted on GitHub and is open to contributions from the community, and Rust's Doc comment feature is extensively used to document the code. Additionally, the system that is used to add new board support to the IDE is in text files that anybody can create or edit. The documentation around the Iron Coder board TOML format is in progress, and when used in conjunction with a public repository of boards (like Arduino's *library manager*) will serve as a foundation for an extensive variety of hardware to be available for use in Iron Coder.

The core external library that Iron Coder uses is **`egui`**, an immediate mode GUI library published under the MIT license [5]. This library provides the utilities to open windows, create GPU rendered layouts with text and graphics, and process IO such as keystrokes, in a flexible and intuitive way. Current GUI features include a "system editor" view (Figure 2), in which boards, represented as 2D images with interactable pins, can be added to the system. Similar to a schematic editor, virtual wires can be used to connect boards together. This mode offers an overview of the hardware layout, along with cursor-hover tooltip information about specific pins and connections, and right-clickable menu options for the boards.

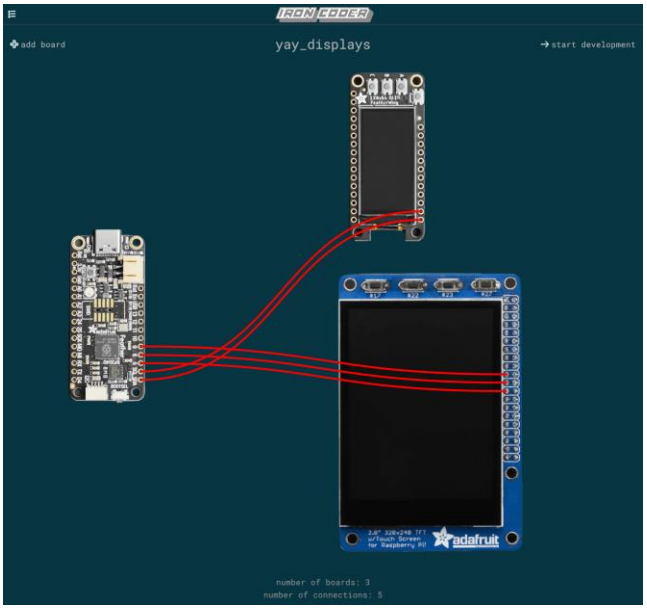

Figure 2 - The "system editor" view of Iron Coder.

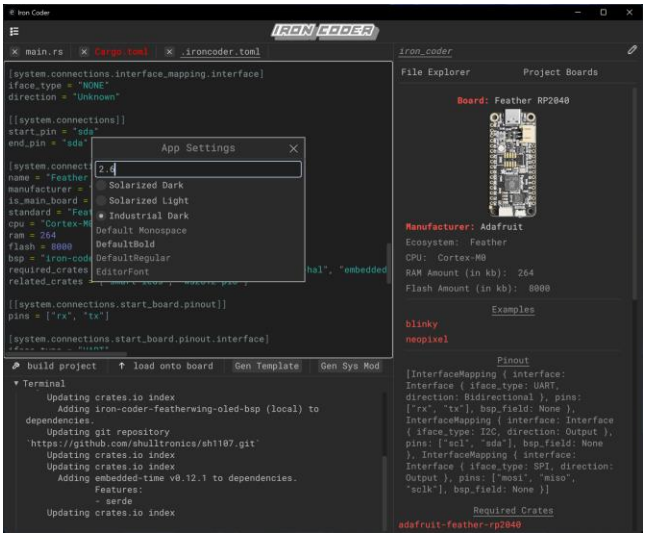

Figure 3 - The "developer view" of Iron Coder, with many of the core features visible.

After defining the system architecture, the developer can switch to the "development view", where the code editing occurs (Figure 3). This view includes a simple text editor that supports multiple tabs and syntax highlighting, a file explorer that the developer can use to browse and create

directories and files, and a board viewer, where the current system boards are shown along with their properties, related Rust crates, and example projects for inspiration. Action buttons are used to compile and load code onto the system (invoking *cargo build* and *cargo run*, respectively), and a built-in status terminal provides feedback to the developer in real-time.

While **`egui`** consumed much of the development time, the backend of Iron Coder was also a significant effort and is decoupled from the front-end as much as possible. While not currently the case, Iron Coder has the potential to be used as a standalone library that could link hardware and firmware design in other contexts as well, such as CLI driven code generators, integration between schematic capture software and IDEs, etc.

The backend consists of a few key structs, the root of which is the **`Project`**. A project represents an open editor session and includes a **`System`** definition (a set of development boards with exactly one "main" board and the connections between those boards), and a set of properties about the project such as its name and location on the host computer's filesystem. The **`serde`** and **`toml`** crates are used to save this information to a configuration file called *.ironcoder.toml*, enabling projects to be loaded and saved to the host computer in the traditional sense. The app contains logic via the menu to open, save, and rename projects (Figure 4).

Figure 4 - Iron Coder's main menu

Another core feature of the backend, which is still in progress, is the ability to parse and generate Rust source code as it relates to the system hardware. Each supported board in Iron Coder is required to have a *board support package*, or BSP. While BSPs are already a common practice in embedded Rust, Iron Coder BSPs have a special structure and require fields that match the names of the pinouts in the Board's manifest and SVG files. By following this convention, the application can identify, construct, and link together firmware components that match the system architecture. This ability is realized via the **`syn`** and **`quote`** crates popular in the Rust metaprogramming space [7]. The intended goal of this feature is to aid the developer in writing the hardware initialization code for the current project. Furthermore, a preliminary inclusion of Rust Analyzer via the **`rust_ap_ide`** crate demonstrates the feasibility of including language server support to help the developer navigate their code base. The end goal of this feature is to be able to quickly identify, jump to, and edit areas of code that relate to specific hardware aspects of the system.

Apart from the technical design decisions and implementations, general aesthetic and usability considerations were integrated into the project. The UI is continuously scalable for comfortable use on high-DPI displays, a color scheme backend is in place for usage in a variety of lighting conditions, and icons are used throughout the application to visually indicate the actions that UI elements will have.

STATUS & LIMITATIONS

Iron Coder is stable and usable as a general code editor and is available to be built from source code; however, it is still a work in progress, with development ongoing. The features that make it unique, such as the code generation capabilities, the hardware-aware language server support, and other embedded-specific utilities are still in the proof-of-concept phase. Only three boards currently have official support, due to limited development person-power and an API that is frequently changing. The instructions and formats for adding new boards need official and published documentation, and for the project to become widely adopted, other community resources such as tutorials and an official website and forums, will need to be created.

FUTURE WORK

As an ongoing project, Iron Coder will continue to grow and change. As the **`embedded_hal`** traits approach the 1.0 release, Rust's use in embedded development is becoming stable, making this a great time to encourage a more widespread user base. Iron Coder has many outstanding goals and milestones, including:

- Completion of code-generation backend
- Publishing of documentation related to community contribution

- Packaging and distribution for cross-platform usage
- Support for multiple programming languages (such as CircuitPython, C++, etc)
- Translation and localization for international usage

As an IDE designed to lower the barrier to entry for embedded Rust development, more research regarding the state-of-the-art in CS education would benefit the goals of the project. With a solid technical foundation in place, the future development of this IDE will include UI and backend elements that are most likely to encourage newcomers to learn and enjoy their coding experience while also instilling practical and proper engineering intuition. Effort will be made to integrate the future work of this project with students and faculty in UF's Department of Engineering Education, with which the author is affiliated. The project will be available to students in UF's Computer Engineering Capstone course as a senior project, and as features stabilize and mature, the software will be introduced to new programming students with the goal of getting user testing and feedback from the target audience.

Additionally, the project will be introduced to the community at large in a variety of ways, including publishing via social media, forums, and other maker communities on the internet, as well as through outreach with some of the vendors whose hardware and software inspired Iron Coder's conception, such as Adafruit, Sparkfun, and Raspberry Pi.

CONCLUSION

Rust's promises to improve the safety and robustness of software has important implications for the field of embedded systems. As the language stabilizes for usage on microcontrollers and other low-power, hardware-centric systems, there is a need to increase the language's accessibility to novice programmers. Physical computing has seen a major evolution in the last two decades, becoming affordable and inspiring as a method of teaching CS and engineering concepts. Iron Coder was developed out of these two facts, aiming to expand the accessibility of embedded Rust development to young or novice engineers.

Iron Coder's design centers around the idea of a hardware system, capturing the inspiration that makes physical computing a compelling endeavor, and tying that into the process of writing code. By utilizing Rust's strengths in library management, cross-compilation, and metaprogramming, many of the initial difficulties of embedded development can be lessened, with the result of increasing the joy and desire of newcomers to continue pursuing computing as a hobby or profession.

REFERENCES

[1] Schwab, K. (2017) 'Introduction', in *The Fourth Industrial Revolution*. New York: Currency.

[2] *Circuitpython* (no date) *CircuitPython*. Available at: https://circuitpython.org/ (Accessed: 11 August 2023).

[3] Hodges, S. *et al*. (2020) *Physical computing:a key element of Modern Computer Science Education*, Lancaster EPrints. Available at: https://eprints.lancs.ac.uk/id/eprint/135064/ (Accessed: 11 August 2023).

[4] *Tensorflow Lite for microcontrollers - experiments with google* (no date) *Google*. Available at: https://experiments.withgoogle.com/collection/tfliteformicrocontrollers (Accessed: 11 August 2023).

[5] Ernerfeldt, E. (no date) *Emilk/Egui: Egui: An easy-to-use immediate mode GUI in rust that runs on both web and Native, GitHub*. Available at: https://github.com/emilk/egui (Accessed: 11 August 2023).

[6] Rust (no date) *Rust Programming Language*. Available at: https://www.rust-lang.org/ (Accessed: 11 August 2023).

[7] *The rust programming language* (no date) *Macros - The Rust Programming Language*. Available at: https://doc.rust-lang.org/book/ch19-06-macros.html (Accessed: 11 August 2023).