BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2026
http://excel.fit.vutbr.cz

Arduino Software Development in Rust

Bc. Michal Žatečka*

Abstract
Embedded systems development has traditionally relied on C and C++ due to their performance and low-level hardware control. However, these languages lack memory-safety guarantees, leaving systems potentially vulnerable to critical errors and security exploits. The Rust programming language offers a modern alternative, providing memory safety at compile time without compromising performance. This work explores the options of adopting Rust for the AVR architecture, specifically for Arduino development. It analyzes existing Rust embedded abstraction layers and compares them against the traditional C/C++ approach. Based on this analysis, the project designs a new CLI tool and a Visual Studio Code extension. This tool enables users to choose between a pure Rust approach or a hybrid that integrates the existing C++ Arduino framework. The proposed design aims to lower the barrier to entry into the Rust-embedded world while maintaining the flexibility required for legacy integration.

*xzatec02@stud.fit.vut.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

Embedded systems are utilized almost everywhere, requiring high safety and reliability. The majority of embedded applications are written in C, which offers high performance and direct hardware interaction, but this flexibility comes at the cost of memory safety. Rust mitigates these issues by employing a unique ownership model that ensures memory safety at compile time without compromising performance.

While Rust is beneficial, compiling it for 8-bit AVR architectures is challenging. The build process requires numerous manual configuration steps, and no existing tool fully automates this workflow for Arduino.

Currently, developers rely heavily on C/C++ frameworks such as Arduino and on universal tools like PlatformIO. While PlatformIO handles dependency management and multi-architecture builds effectively, it locks developers into a C/C++ paradigm. On the Rust side, developers must manually configure Cargo and, if they want to use a C/C++ library, write custom build.rs scripts for library linking and manually translate or generate ABI bindings using tools like bindgen, which can be intimidating for newcomers. [1]

The result of this work is a standalone CLI project manager and a Visual Studio Code extension acting as a thin client. The tool serves as a wrapper for Cargo and PlatformIO, automating dependency management, building, and flashing. The tool offers a dual-mode strategy: a "Pure Rust" mode utilizing high-level Board Support Crates (BSC), and a "Hybrid mode" that automatically compiles and links the C++ Arduino framework via generated bindings.

2. Tool Design and Architecture

The developed CLI tool consists of four logical layers shown in Figure 1. The controller dictates the build pipeline based on whether the user is utilizing pure Rust or integrating C/C++.

A robust and automated build pipeline is critical to the tool's success, as manual compilation for AVR targets requires intricate configuration that can discourage users. The designed tool acts as an orchestrator for Cargo and PlatformIO, executing a specific build pipeline based on the chosen development mode.

2.1 Pure Rust Pipeline

For applications utilizing only Rust and its embedded abstraction layers, the pipeline follows a streamlined process. Cargo orchestrates the rustc compiler. The Rust frontend parses the source code, validates safety rules, and lowers it into LLVM Intermediate Representation (LLVM-IR). The LLVM backend then generates optimized machine code and links it into an Executable and Linkable Format (ELF) file. [2]

Because an ELF file cannot be uploaded directly to an AVR microcontroller, a post-processing step transforms the ELF into a HEX binary file. This HEX file is then ready to be flashed to the device.

2.2 Hybrid Rust and C/C++ Pipeline

When a user opts to integrate the existing C/C++ Arduino framework, the pipeline incorporates additional steps to ensure seamless interoperability. The tool instructs PlatformIO to compile the required C/C++ framework into a static library tailored to the target platform.

To allow Rust to interact with the compiled C/C++ code, the tool automatically adds a crate containing a generated Rust interface (bindings file) using the bindgen utility.

A custom build script (build.rs) is utilized to link the generated static library to the Rust project. Cargo then compiles the Rust code and links the C/C++ static library, outputting a combined ELF file.

Similar to the pure Rust mode, the final ELF file undergoes ELF-to-HEX transformation to produce the uploadable binary.

2.3 Visual Studio Code extension

To improve the developer experience, the system includes a Visual Studio Code extension that communicates with the CLI tool via JSON over standard output. The extension provides a GUI for easily configuring new projects, selecting boards, and executing tasks such as building and monitoring without using the command line.

3. Binary Size Comparison

Because embedded systems operate with highly constrained memory, evaluating the final executable binary size is critical. We tested the tool's output using three different applications on an Arduino Uno and compared the binary sizes against the reference using C++ Arduino framework applications.

3.1 Results

The results shown on Figure 3 demonstrate that pure Rust programs (using the avr-hal crate) compile to binaries that are significantly smaller than the reference C++ Arduino programs.

The hybrid implementation produced the largest binaries across all tests, confirming that integrating C++ Arduino libraries with Rust results in a noticeable size overhead compared to utilizing pure Rust abstractions.

4. Conclusions

The goal of this project was to simplify Rust embedded development for the AVR architecture, providing an alternative to memory-unsafe C/C++. The developed CLI tool and VS Code extension successfully automate the build pipeline, dependency resolution, and flashing procedures. Our experiments confirmed that writing pure high-level, safe Rust code using Board Support Crates produces smaller binaries. Future work will focus on improving support for additional Arduino framework libraries and on extending the supported architectures.

Acknowledgements

I would like to thank my supervisor RNDr. Marek Rychlý Ph.D. for his guidance and support throughout this project.

References

[1] RUST EMBEDDED WORKING GROUP. *The Embedded Rust Book*. 2025.

[2] RUST COMPILER TEAM. *Rust Compiler Development Guide*. 2026.