	Comprehensive Rust for Embedded System (60 Hours)
	This curriculum provides a well-structured learning path for mastering Rust development, integrating specific topics with debugging, target-specific optimization, and code optimization techniques.
	Learning Objectives: • Gain a strong foundation in core Rust concepts • Apply asynchronous programming with Tokio and Futures • Work with Rust Foreign Function Interface (FFI) for C interoperability • Develop socket programming applications in Rust • Profile and optimize Rust code for performance • Understand Rust's memory model and manage lifetimes effectively • Build applications for embedded systems with Rust (no_std environment) • Implement robust error handling mechanisms in Rust applications • Leverage the Rust libc crate for platform-specific functionality
	Table of Contents
Hours 0-8:	• Rust Fundamentals • Data Types • Introduction to functions • Return values • Function arguments • The borrowing concept • Using Panic! • Error handling with match • Structuring Data • Structs • Related Data • Instantiating Structs • Tuple Structs • Pattern Matching • Enums • Defining Types • Expressions • Match control flow operator • Rust Collections • Lists • Values • Vectors • Keys & Hash Maps • Generics • Types • Traits • Lifetimes
Hours 9-14:	**Asynchronous Programming with Tokio and Futures** • **Hands-on:** Building a simple web server using Tokio • Introduction to asynchronous programming in Rust • Understanding Tokio, an asynchronous runtime for Rust • Working with futures and async/await syntax • Implementing asynchronous tasks and handling concurrency • Error handling in asynchronous code • Debugging asynchronous code with `--debug` flag **Hours 15-20: Rust Foreign Function Interface (FFI) with C** • **Hands-on:** Integrating Rust with a C library to perform image processing • Introduction to FFI and its importance • Interfacing Rust code with C libraries • Using `extern` blocks and `unsafe` code • Passing data between Rust and C functions • Handling different types and memory management

- Debugging FFI code with `--debug` flag
- Optimizing FFI code with `--Z mir-opt-level`, `--Z fuel=<crate>=<value>`, and `--codegen-units`
- Target-specific optimization with `RUSTFLAGS=”-C target-cpu=native”`

Hours 21-25: Socket Programming in Rust

- **Hands-on:** Building a chat application using Rust sockets
- Overview of networking in Rust
- Creating TCP and UDP sockets
- Implementing server-client communication
- Handling connections and streams
- Error handling and asynchronous networking
- Debugging socket code with `--debug` flag

Hours 26-41: Rust Benchmarking and Optimization for Target Hardware

- **Hands-on:**
- Rust code optimization
- -C target-cpu=native (assuming it's an Intel processor).
- Understanding Rust's performance characteristics
- Hands on Benchmark | perf | FlameGraph | valgrind
- Profiling Rust code and identifying bottlenecks
- Techniques for benchmarking and performance testing
- Optimizing Rust code for specific hardware targets
- Using compiler flags and optimization techniques

Hours 42-47: Rust Memory Model and Lifetimes

- **Hands-on:** Implementing a data structure with strict lifetime requirements
- Understanding Rust's ownership model
- Exploring references and borrowing in Rust
- Lifetimes and how they enforce memory safety
- Avoiding common pitfalls related to memory management
- Advanced memory management techniques
- Debugging memory-related issues with `--debug` flag
- Optimizing memory usage with `--Z mir-opt-level`, `--Z fuel=<crate>=<value>`, and `--codegen-units`
- Target-specific optimization with `RUSTFLAGS=”-C target-cpu=native”`

Hours 48-54: Rust for Embedded Systems (no_std, Interrupts)

- **Hands-on:** Writing firmware for a microcontroller using Rust
- Introduction to embedded systems development with Rust
- Using the `no_std` environment and custom allocators
- Interfacing with hardware peripherals and sensors
- Handling interrupts and real-time constraints
- Building and deploying Rust code on embedded platforms
- Debugging embedded code with `--debug` flag
- Optimizing embedded code with `--Z mir-opt-level`, `--Z fuel=<crate>=<value>`, and `--codegen-units`
- Target-specific optimization with `RUSTFLAGS=”-C target-cpu=native”`

Hours 55-60: Error Handling and Panic in Rust

- **Hands-on:** Writing a robust file parsing library with comprehensive error handling
- Understanding error handling mechanisms in Rust
- Using `Result` and `Option` for error propagation
- Handling panics and unwinding behavior
- Customizing panic behavior with `panic` macros
- Best practices for error