Rust programming for embedded systems

Łukasz Andrzejewski

Contents

Chapter 1

Introduction to embedded systems

1.1 What is an embedded system?

An embedded system is a specialized computing platform engineered to carry out a dedicated function—often in real-time—under stringent resource constraints such as limited memory, processing capacity, and power consumption. Unlike general-purpose computers, which support a broad range of user applications and are designed for high-level interactivity, embedded systems are tightly integrated with the physical environment they monitor or control. They typically rely on microcontrollers or system-on-chip (SoC) solutions, where the CPU cores, memory (Flash or EEPROM), and peripherals are packaged into a single unit. These platforms focus on reliability, deterministic behavior, and optimization for size, cost, and power efficiency.

Core architectural elements

At the heart of an embedded system lies a microcontroller (e.g., ARM Cortex-M, AVR, RISC-V), which executes firmware stored in non-volatile memory. This firmware initializes hardware components, configures communication interfaces, and runs the core application logic. Peripheral circuits, such as analog-to-digital converters, pulse-width modulators, digital I/O pins, and communication interfaces (SPI, I2C, UART, CAN, USB), provide the link between the digital controller and real-world signals. Additional custom hardware blocks may handle specialized tasks—such as signal processing, security algorithms, or power management—reducing the load on the CPU core and enabling strict power budgets to be met. In more complex designs, an SoC might include GPUs or hardware accelerators to handle demanding tasks like computer vision or cryptographic functions.

Firmware and software stack

The embedded software, often referred to as firmware, is developed in low-level languages like C or C++, with assembly routines for performance-critical sections. Depending on the required performance and timing guarantees, the system may run on a bare-metal architecture (a “super loop” model with interrupt-driven event handling) or under a real-time operating system (RTOS). An RTOS offers deterministic task scheduling, interrupt handling, and inter-task communication mechanisms (e.g., queues, semaphores, mailboxes). Safety-critical industries, such as automotive or medical, often require compliance with strict development standards (e.g., ISO 26262, IEC 62304) and extensive testing to ensure reliability and meet functional safety requirements.

Operational flow and control

Embedded systems typically follow a repetitive cycle of sensing, processing, and actuating. Sensors measure physical variables—ranging from temperature and pressure to acceleration and voltage—and provide data through analog or digital interfaces. The microcontroller processes this data, applies the relevant control or decision algorithms (like PID loops or sensor fusion), and issues appropriate commands to actuators. These actuators can include motors, solenoids, LEDs, or displays, translating digital instructions back into the physical domain. Time-critical operations often rely on interrupts and prioritize tasks, ensuring that essential actions occur within defined latency bounds. In higher-level designs, additional functionality like wireless connectivity (Wi-Fi, Bluetooth, Zigbee) or internet connectivity (Ethernet) is integrated, enabling remote monitoring, configuration, or data offloading to cloud services.

Design constraints and optimization

Resource constraints are a defining characteristic of embedded systems. Memory (RAM and Flash) is commonly in the order of tens to hundreds of kilobytes in low-power MCUs, and CPU clock speeds may be only a few megahertz to a few hundred megahertz. These limitations drive careful optimization of program size, data structures, and algorithmic complexity. Low power consumption is often paramount, especially in battery-operated devices, leading to techniques like dynamic frequency scaling, sleep modes, and peripheral power gating. Hardware design must also meet rigorous cost targets, given that embedded products are frequently manufactured in high volumes. Component selection is scrutinized to balance functionality, performance, and price. Additionally, mechanical form factors, thermal considerations, and electromagnetic compatibility (EMC) constraints can significantly influence PCB layout and overall system architecture.

Application domains

Embedded systems permeate nearly every modern industry. In automotive applications, engine control units (ECUs) modulate fuel injection and ignition timing, while advanced driver-assistance systems (ADAS) perform sensor fusion from cameras, radar, and LiDAR. Consumer electronics integrate various embedded subsystems for battery management, sensor hubs, and user interface control. Industrial automation depends on programmable logic controllers (PLCs) and robotics platforms that demand real-time control loops, robust communication interfaces (like Modbus or industrial Ethernet), and high reliability. Medical devices such as pacemakers and insulin pumps rely on ultra-low-power operation and fault-tolerant design, conforming to strict regulatory guidelines. Even aerospace and defense systems employ embedded technologies under extreme operational conditions, where failure is not an option.

Emerging trends and future directions

As processing capabilities converge with shrinking silicon geometries, MCUs and SoCs are increasingly incorporating machine learning accelerators and hardware cryptographic engines. This trend facilitates on-device inference for edge AI use cases, where real-time decisions must be made without reliance on cloud services. Connectivity and security are also top priorities, leading to widespread adoption of secure bootloaders, hardware security modules, and over-the-air update mechanisms that protect devices against malicious tampering. Real-time operating systems continue to evolve with more advanced scheduling policies, integrated networking stacks, and modular frameworks, further streamlining embedded software development. The Internet of Things (IoT) illustrates how embedded devices can form extensive networks, processing vast amounts of sensor data at the edge while leveraging the cloud for analysis, reporting, and long-term storage.

1.2 Microcontroller vs. Microprocessor

When discussing the foundational components of embedded systems, the terms *microcontroller* (MCU) and *microprocessor* inevitably surface. Although both are types of integrated circuits (ICs), they differ significantly in terms of integration, performance, packaging, and target applications. At a high level, a microprocessor is the computational core (CPU) often found in general-purpose computing platforms, whereas a microcontroller is more specialized, encapsulating a CPU with on-chip memory and peripheral interfaces—an arrangement that is typically optimized for control-oriented tasks in resource-constrained environments.

Microprocessors

A microprocessor functions as the central processing unit (CPU) in a system, executing instructions, performing arithmetic operations, and managing logic tasks. Typically employed in desktop computers, laptops, and servers, microprocessors are known for their ability to support multitasking, handle large software stacks, and accommodate complex operating systems. They often incorporate multiple cores running at high clock speeds, which allows for parallel processing and significant computational throughput. However, they rely on external components—most notably RAM, ROM, and various input/output controllers—for storage and interaction with peripheral devices. Because of their high-performance design, microprocessors are well-suited for applications where substantial computational power is paramount. They excel in scenarios that demand flexibility, such as running full-scale operating systems, performing complex data analytics, or managing multimedia tasks. This level of capability comes with trade-offs in power consumption, cost, and board space, making microprocessor-based solutions less ideal for tightly integrated, low-power embedded applications.

Microcontrollers

In contrast to microprocessors, microcontrollers are specialized integrated circuits that place an emphasis on direct interaction with sensors, actuators, and other control elements. A microcontroller typically includes not just the CPU core (akin to a simplified microprocessor), but also on-chip program memory (Flash or ROM), data memory (RAM), and peripheral interfaces (e.g., GPIO, timers, ADCs, serial communication). This high degree of integration minimizes external component requirements, enabling microcontrollers to operate efficiently in environments with strict constraints on size, power, and cost. By design, microcontrollers often feature single-core CPUs operating at relatively modest clock speeds. Although they may not match the raw computational horsepower of microprocessors, they excel at handling dedicated, real-time control tasks with minimal overhead. Their lower clock frequencies and reduced power draw are especially beneficial in battery-powered or energy-sensitive applications, such as IoT nodes, home appliances, and automotive control systems. Furthermore, microcontrollers' simplified architecture makes them easier to program, which can expedite firmware development and streamline hardware-software integration.

System on Chip (SoC)

A System on Chip (SoC) extends the concept of integration even further, combining a CPU (microprocessor or microcontroller-like core) with multiple peripherals and specialized units into a single package. In many cases, SoCs blur the lines between traditional microcontrollers and microprocessors: 1. Microcontroller-Based SoCs are fundamentally microcontrollers but incorporate higher clock speeds, additional memory, and specialized hardware accelerators (e.g., for cryptography or signal processing). Espressif's ESP series, for instance, are labeled as SoCs, yet they share many traits with conventional microcontrollers. 2. Microprocessor-Based SoCs as seen in certain Intel or ARM-based platforms, merge microprocessor cores with ancillary chipsets or co-processors, providing a single-package solution typically without integrated main memory. These solutions often target complex, high-performance scenarios while seeking to reduce board footprint and power consumption relative to a multi-chip configuration. SoCs may include GPU cores, FPGA fabric, or other domain-specific accelerators (e.g., AI/ML inference engines), thereby accommodating more compute-intensive, feature-rich applications. This advanced integration makes SoCs suitable for products that require both the control-centric approach of a microcontroller and the processing capabilities of a microprocessor—ultimately, bridging the gap between embedded control tasks and higher-level data processing requirements.

Packaging Considerations The term *package* refers to the protective housing surrounding the silicon die(s) of microcontrollers, microprocessors, or SoCs. Packaging facilitates several crucial functions: 1. Physical Protection: Shields the silicon from mechanical damage, contamination, and environmental factors. 2. Thermal Management: Helps dissipate heat generated by the IC to maintain stable operating temperatures. 3. Connectivity: Provides pins, pads, or balls (depending on package style) for electrical connections to the PCB or external devices.

Memory mapping

Memory mapping assigns specific address ranges to various components within a microcontroller system, including peripherals, code memory, and data memory. This organization allows the processor core to interact with different blocks by referencing their designated memory addresses. For example, to retrieve sensor data from an Analog-to-Digital Converter (ADC), the processor accesses the address range allocated to the ADC, prompting it to transfer the data. Memory addresses are typically represented in hexadecimal notation (e.g., 0x100). In practice, if the processor needs to read the next instruction from code memory, it sends a read request to an address like 0x100 within the designated range. Similarly, to configure a timer peripheral, the processor writes data to an address within the timer's assigned range (e.g., 0x22F). This direct addressing simplifies software development by allowing programmers to control peripherals as if they were regular memory locations.

Application memory layout

When an embedded application is compiled and linked, the resulting memory layout is organized into specific regions that correspond to different parts of the program. The microcontroller utilizes two primary types of memory: Flash and SRAM. Flash memory is non-volatile and stores the static application image or executable code, retaining its contents even when power is lost. SRAM, on the other hand, is volatile and holds dynamic data generated during the application's runtime, similar to a computer's main memory.

Upon powering up, the application image resides in Flash memory, divided into three main regions: 1. .text: Contains all the program code, including functions and the Interrupt Vector Table (IVT), which maps interrupts to their corresponding Interrupt Service Routines (ISRs). 2. .rodata: Stores read-only data such as constants used by the application. 3. .data: Holds initialized global variables that the program uses during execution.

During the startup sequence, the microcontroller initializes the SRAM by loading the necessary data from Flash memory. This process involves copying the .data region from Flash to SRAM and setting up the runtime environment for the application. The organized memory layout ensures that the processor can efficiently execute code and manage data, facilitating reliable and predictable behavior of embedded applications.

Understanding memory mapping and application memory layout is crucial for effective embedded system development. Memory mapping enables the processor to efficiently interact with various peripherals by assigning specific address ranges, simplifying the control and data transfer processes. Meanwhile, comprehending the application memory layout ensures that code and data are organized optimally within the microcontroller's memory architecture.

Instruction Set Architecture (ISA)

Processor architecture defines the fundamental design and organization of a processor's central processing unit (CPU). In the embedded domain, popular architectures include ARM (Cortex-M processors), RISC-V, and Xtensa. The selection of a processor architecture by microcontroller manufacturers is influenced by factors such as performance requirements, power efficiency, cost, and the specific use case. At the core of a microcontroller is the CPU, which executes application code and instructions based on its ISA. The ISA serves as the hardware-software interface, defining the functions the processor can perform, such as arithmetic operations and data storage. Each ISA is unique to its architecture—examples include ARM, Intel x86, Xtensa, and RISC-V—and consists of machine instructions that are typically written in assembly language. High-level programming languages like C or Rust are compiled into assembly code specific to the processor's ISA, requiring the compiler to be aware of the underlying architecture to generate appropriate machine code.

Memory Architecture

Microcontrollers utilize two primary types of memory: code memory and data memory. Code memory, usually implemented with non-volatile Flash memory, stores the application code or instructions that are flashed to the microcontroller. Data memory, typically using volatile SRAM, holds the application data generated during runtime. Effective memory architecture is crucial for organizing and connecting these memory types to the processor core, as simultaneous access to both can lead to contention. There are two main memory architectures: Von-Neumann and Harvard. The Von-Neumann architecture uses a single bus for both code and data memory, which can lead to resource contention but offers simplicity. In contrast, the Harvard architecture employs separate buses for code and data memory, enhancing performance by eliminating bus contention and supporting higher efficiency, especially in pipelined architectures. Modern implementations of the Harvard architecture often feature read/write code memory, moving away from the traditional read-only configurations.

1.3 Microcontrollers from the ESP family

ESP microcontrollers, developed by Espressif Systems, are renowned for their versatility and widespread use in Internet of Things (IoT) applications. The ESP (Espressif Systems' Platform) family is particularly distinguished by its integrated Wi-Fi capabilities, and in some models, Bluetooth connectivity, making them ideal for connected devices. The two most notable ESP microcontrollers are the ESP8266 and the ESP32.

The ESP8266 was one of Espressif's first microcontrollers to gain significant popularity. It is celebrated for its built-in Wi-Fi functionality, providing a cost-effective solution for connecting devices to the Internet. The ESP8266 is equipped with a 32-bit RISC CPU based on the Xtensa LX3 core architecture, offering a balance of performance and efficiency suitable for a variety of embedded applications.

Building on the success of the ESP8266, the ESP32 emerged as a more powerful and feature-rich microcontroller. The ESP32 series not only includes enhanced Wi-Fi capabilities but also incorporates Bluetooth connectivity, optional dual-core processing, and a wider array of peripherals. This makes the ESP32 suitable for more demanding applications that require higher performance and greater connectivity options. Additionally, ESP32 microcontrollers come with different core architectures, including both Xtensa and the open-source RISC-V, providing developers with greater flexibility in their designs.

Espressif Systems offers a comprehensive suite of System-on-Chip (SoC) offerings tailored to diverse IoT applications, categorized into several families:

- ESP32 Series: Introduced in 2016 as a successor to the ESP8266, the ESP32 is based on the Xtensa dual-core 32-bit LX6 processor. It supports both Wi-Fi and Bluetooth 4.2, along with various I/O options, making it suitable for a wide range of applications.

- ESP32-Sx Series: Launched in 2020, the S-series features the more recent Xtensa LX7 processor. These SoCs are available in single and dual-core configurations and support Bluetooth 5.0, offering enhanced performance and connectivity.
- ESP32-Cx Series: Also emerging in 2020, the C-series incorporates the open-source RISC-V processor. These microcontrollers come in single-core and dual-core variants, supporting Bluetooth 5.0 and Wi-Fi 6, catering to applications requiring advanced connectivity and processing power.
- ESP32-Hx Series: Announced in 2021, the H-series builds on the C-series by adding more connectivity options, including Thread and Zigbee protocols, enhancing their suitability for complex IoT networks.
- ESP32-Px Series: The most recent addition, announced in 2024, the P-series is RISC-V-based and targets AI applications. It offers multi-core functionality with a dual-core RISC-V processor for high-performance workloads and a single-core RISC-V for low-power operations. Notably, the P-series does not incorporate connectivity features, focusing instead on secure and high-efficiency use cases.

These SoC offerings cater to the evolving needs of IoT development, providing a spectrum of features from ultra-low-power performance to advanced security measures. Espressif also provides various development kits and boards, such as the ESP32-C3-DevKitM-1 and ESP32-C3-AWS-ExpressLink-DevKit, which integrate their SoCs and come with different onboard components and pin configurations to suit various application requirements.

1.4 Rust in the context of embedded systems

Embedded development has significantly transformed over the years, expanding from limited, mission-specific devices to a vast landscape of versatile applications. Initially confined to isolated, air-gapped systems, embedded technologies now encompass a wide range of functionalities, including edge AI and software-defined hardware. This evolution has driven advancements in both performance and complexity, enabling embedded systems to meet the increasing demands of modern technology.

For decades, C and C++ have been the cornerstone languages in embedded development, renowned for their efficiency and ability to meet industry requirements. These languages have reliably supported the creation of robust embedded applications. However, as embedded systems become more sophisticated and the need for enhanced safety and security intensifies, the limitations of relying solely on C and C++ become apparent. The growing complexity of applications necessitates a shift towards more modern programming languages that can better address these emerging challenges.

Rust has emerged as a powerful alternative in the embedded programming arena, offering a unique combination of high performance, memory safety, and modern language features. Its ability to provide memory safety without sacrificing speed makes Rust an attractive choice for developing reliable and efficient embedded systems. Rust's expressiveness and advanced features not only enhance developer productivity but also position it to potentially revolutionize the embedded landscape by balancing performance with safety and modern programming paradigms. The embedded Rust ecosystem is experiencing rapid growth, driven by the increasing number of contributors and major industry players adopting Rust for their projects. This expansion is accompanied by continuous improvements in tooling and ecosystem support, making Rust more accessible and powerful for embedded developers. Additionally, educational resources focused on embedded Rust are significantly improving, providing learners with the necessary materials to stay current and effectively leverage Rust in their embedded projects.

Rust emerges as a compelling choice for embedded systems. Developed initially as Graydon Hoare's personal project and later adopted by Mozilla, Rust was designed to address common software failures by eliminating unsafe coding practices. As a modern, multi-paradigm compiled systems programming language, Rust offers memory safety, exceptional speed, zero-cost abstractions, and high portability. These attributes make it an ideal fit for embedded systems, which require both performance and reliability. Additionally, Rust simplifies the development process by providing a more integrated environment compared to the traditional complexities of setting up make scripts, unit testing, and package management found in languages like C and C++. Rust has gained significant traction in both cloud and embedded environments, with major tech companies integrating it into their systems. Notably, Microsoft and Google have reported that Rust has helped eliminate up to 70% of their security issues in certain areas. In the embedded domain companies adopting Rust alongside C and C++ in their job requirements. This growing adoption underscores Rust's effectiveness in enhancing security, performance, and developer productivity across diverse applications.

Key Features of Rust:

- Memory Safety - Rust's unique ownership system ensures memory safety without relying on a garbage collector. This system prevents common issues like null pointer dereferences, buffer overflows, and data races by enforcing strict rules on memory access and sharing.

- Fearless Concurrency - Rust facilitates safe and efficient concurrent programming through its ownership and borrowing mechanisms. The compiler enforces rules that allow multiple threads to access data concurrently while minimizing the risk of data races.
- Zero-Cost Abstractions - Rust offers high-level abstractions that do not incur runtime overhead. These abstractions are largely eliminated during compilation, resulting in performance that rivals low-level languages like C and C++.
- Static Typing - being statically typed, Rust ensures that variable types are known at compile time. This feature helps catch many errors early in the development process, leading to more robust and reliable software.
- Pattern Matching - Rust includes powerful pattern-matching capabilities that allow developers to express complex conditional logic in a concise and readable manner, enhancing code clarity and maintainability.
- Cross-Platform Support - Rust is designed for portability across various platforms, enabling developers to write code that can run on different operating systems and hardware architectures with minimal modifications.
- Community and Ecosystem - Rust boasts a vibrant and growing community, supported by its package manager, Cargo. Cargo simplifies dependency management and project setup, while the Rust ecosystem offers a wide range of libraries and frameworks tailored for different purposes.
- Integration with Other Languages - Rust is engineered to interoperate seamlessly with other languages, particularly C. This interoperability allows Rust code to integrate with existing projects and leverage libraries written in other languages, facilitating gradual adoption and enhancing flexibility. Rust stands out as a robust, secure, and efficient programming language well-suited for the demands of embedded systems. Its combination of memory safety, concurrency support, performance, and ease of integration positions Rust as a transformative tool in the embedded development landscape. As more companies and developers adopt Rust, its ecosystem and community continue to grow, further solidifying its role in advancing embedded technology.

1.5 Bare-metal programming vs. RTOS

In embedded development, there are two primary approaches to interacting with hardware: bare-metal programming and using a Real-Time Operating System (RTOS). Bare-metal programming involves writing code that directly interfaces with and controls the hardware without any intermediary layers. This approach offers maximum control and minimal overhead, making it ideal for resource-constrained systems, real-time applications, and scenarios requiring precise hardware management. Conversely, the RTOS approach introduces layers of abstraction by running application code within an operating system environment. An RTOS handles task scheduling and can provide additional functionalities such as memory management, networking, and security, simplifying development but adding overhead and reducing direct hardware control. The choice between bare-metal and RTOS depends on the application's complexity, performance requirements, and the need for system responsiveness. Generally, avoiding OS overhead is preferred to maximize performance, but an RTOS becomes necessary as hardware and system complexity increase, or when applications demand quick response times and efficient power management.

In the context of Rust programming for embedded systems, development can be categorized into no-std (core library) and std (standard library) approaches. no-std development refers to bare-metal programming without relying on the Rust standard library, providing greater control, efficiency, and smaller memory footprints. This approach is ideal for resource-constrained systems, real-time applications, and scenarios requiring direct hardware interaction. Conversely, std development involves using the Rust standard library, which offers a richer set of functionalities, standardized APIs, and abstractions that simplify development and enhance portability across different platforms. This approach is suitable for more complex applications that benefit from the extensive features provided by the standard library, albeit with increased resource usage and some overhead.

Espressif supports both std and no-std development for ESP devices through the esp-rs project on GitHub, which unifies efforts to integrate Rust with ESP microcontrollers. The Rust ESP ecosystem emphasizes portability by maintaining common abstractions across different ESP devices, allowing developers to reuse codebases across similar hardware configurations.

- ESP no-std Rust Ecosystem: Follows a layering approach common in many embedded Rust projects. It includes:
    1. Peripheral Access Crate (PAC): Provides low-level access to microcontroller registers specific to each ESP series, found in the `esp-pacs` repository.
    2. Microarchitecture Crate: Specific to processor core functions, supporting architectures like RISC-V.
    3. Hardware Abstraction Layer (HAL) Crates: Offer user-friendly APIs for peripherals, implementing Rust's safety mechanisms to prevent data races and ensure correct pin configurations. These are available in the `esp-hal` repository.
    4. Embedded Trait Crates: Led by the `embedded-hal` crate, these provide platform-agnostic traits that standardize peripheral interactions, enhancing portability across different hardware environments. Companion crates like

`embedded-hal-bus`, `embedded-hal-async`, and `embedded-hal-nb` extend the functionality of `embedded-hal`.

- ESP std Rust Ecosystem: Built on top of the ESP-IDF (IoT Development Framework), Espressif's official framework for ESP32 and ESP8266 microcontrollers. The ESP-IDF incorporates FreeRTOS for task scheduling and includes comprehensive features like peripheral drivers, networking stacks, and command-line tools. Rust support is layered on top of the existing C-based ESP-IDF through Rust's Foreign Function Interface (FFI), enabling developers to utilize Rust's safety features while leveraging the established ESP-IDF framework. Key components include:

    1. esp-idf-sys: Provides FFI bindings to the underlying ESP-IDF C APIs, allowing Rust code to call these functions.
    2. esp-idf-hal: Wraps the `esp-idf-sys` bindings with safe Rust abstractions, supporting peripherals like GPIO, SPI, I2C, Timers, and UART, and implementing `embedded-hal` traits.
    3. esp-idf-svc: Supports the implementation of ESP-IDF services such as Wi-Fi, Ethernet, HTTP, and MQTT, also implementing `embedded-svc` traits for standardized service interactions.

This integration ensures that Rust developers can access and utilize ESP-IDF functionalities while maintaining Rust's safety guarantees, facilitating the development of robust and secure embedded applications.

Development options

To develop in Rust with ESP microcontrollers, developers have two primary options: virtual hardware and physical hardware. - Virtual Hardware: Tools like Wokwi offer an online virtual simulation environment where developers can design, build, and test electronic circuits and embedded projects without needing physical components. Wokwi supports real-time simulation, Wi-Fi connectivity, and integrates seamlessly with Rust, providing a convenient platform for education, prototyping, and development without the complexities of hardware setup. - Physical Hardware: Alternatively, developers can use physical ESP development boards, which require setting up a local development environment, including toolchains for compiling, flashing, and debugging code. Espressif provides various development kits that integrate their SoCs, such as the ESP32-C3-DevKitM-1 and ESP32-C3-AWS-ExpressLink-DevKit. These boards vary in onboard components and pin configurations to cater to different application needs. While physical hardware offers hands-on experience and real-world testing, it involves additional setup steps and potential hardware-related challenges.

Compiler toolchains

Developing for embedded systems typically involves creating code on a host computer and deploying it to a target microcontroller with a different architecture. This process, known as cross-compiling, requires a specialized compiler toolchain configured to generate binaries compatible with the target's architecture, such as RISC-V. A compiler toolchain is a sequence of tools that transforms high-level code into a microcontroller-compatible binary executable. The toolchain usually includes three main stages:

- Compiling: The compiler converts high-level code (e.g., C, C++, Rust) into assembly language specific to the processor's Instruction Set Architecture (ISA). It performs syntax checking and code optimization but does not handle placement in memory or include pre-compiled library code.
- Assembling: The assembler translates assembly instructions into machine code, producing object files (e.g., with a `.obj` extension). These object files contain binary representations of the code that need to be linked.
- Linking: The linker combines object files from the compiler and pre-compiled libraries to create the final executable binary. It assigns memory addresses and organizes the program image, ensuring that all parts of the code are correctly placed in the microcontroller's memory space.

This compilation process is essential for generating the executable code that can run on the embedded device, ensuring compatibility between the host development environment and the target hardware.

Debug Toolchains

After compiling, the executable code must be transferred and debugged on the target microcontroller. A debug toolchain encompasses the tools required to download code to the microcontroller and perform debugging tasks. The debug toolchain typically involves three components: - Hardware Probe/Adapter: This device serves as the bridge between the host computer and the

microcontroller, connecting via interfaces such as JTAG, Serial Wire Debug (SWD), or UART. It facilitates the flashing of code to the microcontroller and enables live debugging. Hardware probes can be integrated into development boards or exist as standalone adapters. - Control Software: Software like OpenOCD communicates with the hardware probe to manage programming, debugging, and testing of the embedded target. It handles the communication protocols necessary to interact with the microcontroller. - Debugging Software: Tools such as the GNU Debugger (GDB) provide frameworks for debugging the embedded application. They allow developers to set breakpoints, inspect registers and memory, and step through code to identify and resolve issues within the microcontroller.

Together, these tools enable developers to efficiently develop, test, and debug embedded applications, ensuring that the code operates correctly on the target hardware.

Chapter 2

Working with peripherals and drivers

In embedded systems, peripherals are essential components that serve as the interface between the microcontroller and the physical world. They enable the conversion of physical signals, allowing the system to interact with external devices such as sensors, actuators, and user inputs. These peripherals are directly connected to the microcontroller's physical pins and are fundamental to the functionality of embedded applications. Common peripherals found in most commercial microcontrollers include General Purpose Input/Output (GPIO), Timers and Counters, Pulse Width Modulation (PWM), Analog-to-Digital Converters (ADCs) and Digital-to-Analog Converters (DACs), and Serial Communications.

2.1 General Purpose Input/Output (GPIO)

GPIO pins are versatile interfaces that can be configured as either digital inputs or outputs, providing significant flexibility in embedded applications. As inputs, GPIO pins can detect states such as switch or button presses, enabling user interaction and sensor data acquisition. As outputs, they can control devices like LEDs, motors, or other actuators by toggling between high (1) and low (0) states. The programmability of GPIO allows them to emulate the functions of other peripherals, making them indispensable for a wide range of tasks within embedded systems.

2.2 Timers and Counters

Timers and Counters are peripherals designed to handle specific timing and control functions within embedded applications. Although they share similar underlying circuitry, their applications differ based on their functionality. Timers are used to measure time intervals, generate precise delays, and determine the duration of events. They are crucial for tasks that require accurate timekeeping and scheduling. Counters, on the other hand, track the number of occurrences of specific events, such as counting pulses from a sensor or monitoring motor rotations. By incrementing or decrementing based on external events or clock pulses, counters facilitate event-driven operations and data collection in various applications.

2.3 Pulse Width Modulation (PWM)

Pulse Width Modulation (PWM) is a technique used to simulate analog signal variations using digital signals. It is commonly employed to control the speed of motors and the brightness of LEDs by adjusting the duty cycle—the percentage of time the signal remains high versus low within a given period. By varying the duty cycle, PWM allows for precise and efficient control over the average power delivered to a load. This capability enables smooth and accurate adjustments in device behavior, making PWM a critical peripheral for applications that require variable intensity or speed control.

2.4 Analog-to-Digital Converters and Digital-to-Analog Converters

ADCs and DACs are peripherals that facilitate the conversion between digital and analog signals, bridging the gap between the digital processing capabilities of microcontrollers and the analog nature of the physical world. An Analog-to-Digital Converter (ADC) converts continuous analog signals into discrete digital values. It samples an analog input, quantizes the signal, and

produces a digital representation. Common applications of ADCs include reading data from sensors such as temperature, light, and microphones, enabling the microcontroller to process real-world analog data in a digital format.

Conversely, a Digital-to-Analog Converter (DAC) performs the reverse function by converting digital signals into analog voltages or currents. DACs are used in scenarios where digital devices need to interact with the analog world, such as waveform generation or audio output. By taking a digital input and producing a continuous analog output, DACs enable the microcontroller to control analog devices and generate analog signals based on digital computations.

2.5 Serial communications

Serial communication peripherals enable the transmission and reception of data in a sequential, bit-by-bit manner over a single communication line. This method of data transfer is essential for connecting microcontrollers with other computing devices or peripheral components. Common serial communication protocols utilized in microcontrollers include:

- UART (Universal Asynchronous Receiver/Transmitter): Facilitates asynchronous serial communication, commonly used for debugging and communication with PCs or other microcontrollers.
- SPI (Serial Peripheral Interface): A synchronous protocol used for high-speed communication between microcontrollers and peripheral devices like sensors, displays, and memory devices.
- I2C (Inter-Integrated Circuit): A multi-master, multi-slave protocol ideal for communication with multiple low-speed peripherals over short distances.

Serial communications are fundamental for enabling microcontrollers to interact with external devices, exchange data, and perform coordinated operations within complex embedded systems.

2.6 The Pin Interface

Microcontroller pins serve as the physical connections between the internal peripherals and the external environment, enabling embedded systems to interact with the outside world. Given the variety of peripherals integrated within a microcontroller and the limited number of pins available on different packages, these pins often need to support multiple functions through a mechanism known as pin multiplexing. For instance, on the ESP32-C3 controller, not all pins support multiple functions, but some, like pin number 27, can operate as both UART and GPIO.

Determining the function of each pin is managed programmatically by configuring the microcontroller, which utilizes internal multiplexers to select the desired function for each pin. The structure of this selection process typically involves three stages:

- Multiplexer Selection: The first stage involves a multiplexer that chooses the pin function (e.g., GPIO, ADC) based on software control. This allows a single pin to serve multiple roles depending on the application's requirements.
- Peripheral Configuration: The second stage connects the selected function to the corresponding peripheral. This peripheral is configured and controlled by software running on the CPU, often requiring the activation of a clock source to synchronize operations across the system.
- System Bus Connection: The final stage links the peripheral to the system bus, enabling the CPU to access and manipulate the peripheral's internal registers for configuration and data retrieval.

Understanding the pin interface is crucial, especially when programming in bare-metal environments, as the exact structure and capabilities of the pin multiplexers can vary between different microcontrollers. Consulting the microcontroller's reference manual or datasheet is essential to comprehend the specific configurations and functionalities available.

Additionally, clock management plays a significant role in the pin interface. Clocks are used to synchronize operations across the microcontroller, but they also contribute to power consumption. To optimize power usage, manufacturers typically disable clocks for peripherals that are not in use, ensuring that only the necessary components consume power during operation.

The pin interface is a fundamental aspect of microcontroller design, enabling flexible and efficient interaction with the physical world through a limited number of pins. By leveraging pin multiplexing and careful software configuration, microcontrollers can support a wide range of functionalities within compact packages. Mastery of the pin interface, including understanding multiplexer configurations and clock management, is essential for developing robust and power-efficient embedded systems.

2.7 Polling vs. interrupts

In embedded systems, the processor can be notified of peripheral events through two primary methods: polling and interrupts. Polling is a “pull” mechanism where the processor repeatedly checks the status of a peripheral to determine if an event has occurred. For example, continuously monitoring a GPIO pin to detect a button press can lead to inefficiency, especially if the event is rare or infrequent. This constant checking wastes processing resources and power, making it unsuitable for scalable or power-sensitive applications. Additionally, in safety-critical systems like automotive airbag controls, polling can introduce delays that may compromise system responsiveness and safety.

Interrupts, on the other hand, operate on a “push” mechanism. In this approach, peripherals notify the processor of events as they occur, allowing the processor to respond immediately without the need for constant checking. When an interrupt is triggered, the processor temporarily halts its current tasks, retrieves the corresponding Interrupt Service Routine (ISR) from the Interrupt Vector Table (IVT), executes the ISR to handle the event, and then resumes normal operation. This method is more efficient and responsive, reducing unnecessary processor load and power consumption. However, interrupts introduce complexity in development and debugging due to their asynchronous nature and potential for data race conditions, where multiple threads access shared data concurrently.

Interrupts consist of three main components: - Interrupt Source: The peripheral generating the event, such as an ADC completion or a GPIO button press. - Interrupt Vector Table (IVT): A table mapping each interrupt source to its corresponding ISR’s memory address, allowing the processor to locate and execute the appropriate ISR when an interrupt occurs. - Interrupt Service Routine (ISR): The specific function that executes in response to an interrupt, handling the necessary operations related to the event.

Interrupts are managed by an interrupt controller, which prioritizes and handles multiple simultaneous interrupts, ensuring that higher-priority events are addressed first. Additionally, both peripheral-level and CPU-level configurations are required to activate and manage interrupts effectively.

A common question in embedded development is whether to use polling or interrupts for a particular application. The answer often depends on the specific requirements of the application:

- Application Complexity and Responsiveness: For simple applications that do not require quick response times, polling may be sufficient. However, as the complexity of the application or microcontroller hardware increases, interrupts become essential to maintain system responsiveness. In more complex systems, the overhead of managing multiple polling loops can hinder performance and resource utilization.

- Event Frequency: If events occur infrequently, polling can lead to inefficient use of processor resources, as the CPU spends time repeatedly checking for events that rarely happen. Interrupts are more efficient in such scenarios, as the processor remains free to perform other tasks and only responds when an event occurs.

- Power Consumption: In applications where power efficiency is critical, such as battery-operated devices, interrupts are advantageous. Microcontrollers can enter low-power sleep modes and rely on interrupts to wake up when necessary, thereby conserving energy. Polling, which requires the processor to remain active and continuously check for events, can lead to higher power consumption.

- Critical Timeliness: In safety-critical applications, such as automotive control systems, the timely detection and response to events are paramount. Interrupts ensure that the processor can handle urgent events immediately, whereas polling might introduce delays that could compromise safety.

- Low-Power Operation: Many microcontrollers use interrupts to wake up from low-power sleep modes. If an application requires frequent transitions between active and sleep states to save power, interrupts are necessary to efficiently manage these transitions without the need for constant polling.

Chapter 3

API reference

3.1 GPIO

General Purpose Input/Output (GPIO) is a fundamental feature in microcontrollers that allows digital interaction with external devices. GPIO pins can be configured as either inputs or outputs and operate in two modes: digital or analog.

- Digital Pins handle two states: high (e.g., 5V) and low (0V). They are essential for peripherals like timers, counters, PWM, and serial communication.
- Analog Pins can handle a range of voltage values (e.g., 0-5V) and are used with peripherals such as ADCs (Analog-to-Digital Converters) and DACs (Digital-to-Analog Converters).

Not all microcontroller pins support analog functions, so consulting the device datasheet is necessary for configuration.
Active States in GPIO:

- Active High: A high voltage level represents a “true” state.
- Active Low: A low voltage level represents a “true” state.

RTOS version

```
// Take peripherals (singleton instance) and configure pin direction

let peripherals = Peripherals::take().unwrap();
let input_pin = PinDriver::input(peripherals.pins.gpio1).unwrap();
let output_pin = PinDriver::output(peripherals.pins.gpio2).unwrap();

// Configure pin pull (input pins)

pin.set_pull(Pull::Up).unwrap(); // pull-up configuration
other_pin.set_pull(Pull::Down).unwrap(); // pull-down configuration

/*
  Configure pin drive (output pins)
  - Push-Pull (default): Can drive the pin both high and low.
  - Open-Drain: Can only pull the pin low.
*/

pin.into_output_od().unwrap();
PinDriver::output_od(peripherals.pins.gpio1).unwrap(); // alternative
```

```
/*
   Configure interrupt type (input pins)
   - Edge-Triggered: Detects rising, falling, or both edges.
   - Level-Triggered: Detects high or low voltage levels.
*/

pin.set_interrupt_type(InterruptType::PosEdge).unwrap();
pin.set_interrupt_type(InterruptType::NegEdge).unwrap();
pin.set_interrupt_type(InterruptType::AnyEdge).unwrap();
pin.set_interrupt_type(InterruptType::LowLevel).unwrap();
pin.set_interrupt_type(InterruptType::HighLevel).unwrap();

// Configure drive strength (output pins)

pin.set_drive_strength(DriveStrength::I5mA).unwrap();
pin.set_drive_strength(DriveStrength::I10mA).unwrap();

// Reading input by polling

loop {
    if pin.is_low() {
        log::info!("Low!");
    }
    if pin.is_high() {
        log::info!("High!");
    }
}

// Reading input by interrupts

fn gpio_callback() {
    // ISR code
}
unsafe { pin.subscribe(gpio_callback).unwrap() }
pin.enable_interrupt().unwrap();

// Writing output

pin.set_low().unwrap();  // set pin to low
pin.set_high().unwrap(); // set pin to high
```

Bare-metal version

```
let peripherals = esp_hal::init(Config::default());
let pin = Input::new(peripherals.GPIO3, Pull::Up);
pin.set_low();
let other_pin = Output::new(peripherals.GPIO3, Level::Low);
other_pin.set_drive_strength(DriveStrength::I5mA);

loop {
    if pin.is_low() {
        log::info!("Low!");
```

```
    }
    if pin.is_high() {
        log::info!("High!");
    }
}

// Interrupts

#![no_std]
#![no_main]

use core::cell::{Cell, RefCell};
use critical_section::Mutex;
use esp_backtrace as _;
use esp_hal::{
    gpio::{Event, Input, Pull, Io},
    prelude::*,
};
use esp_println::println;

static G_PIN: Mutex<RefCell<Option<Input>>> = Mutex::new(RefCell::new(None));

#[handler]
fn gpio() {
    critical_section::with(|cs| {
        // Obtain access to global pin and clear interrupt pending flag
        G_PIN.borrow_ref_mut(cs).as_mut().unwrap().clear_interrupt();
    });
}

#[main]
fn main() -> ! {
    let peripherals = esp_hal::init(Config::default());

    io.set_interrupt_handler(gpio);
    let some_pin = Input::new(peripherals.GPIO0, Pull::Up);
    some_pin.listen(Event::FallingEdge);
    critical_section::with(|cs| G_PIN.borrow_ref_mut(cs).replace(some_pin));
    loop {
      // do the work
    }
}
```

3.2 ADCs

The physical world operates on analog principles, with parameters like temperature, pressure, and speed existing as continuous values. This analog nature creates a gap when interfacing with digital systems such as microcontrollers and microprocessors, which rely on discrete digital values. To bridge this gap, embedded systems must measure these analog parameters and respond accordingly. Analog-to-Digital Converters (ADCs) play a crucial role by converting analog voltages into digital values, enabling digital systems to process real-world physical data effectively.

An ADC consists of several key components: the input signal, which is the analog voltage to be measured; the digital output, whose width is determined by the ADC's resolution (commonly 8, 10, 12, or 14 bits in controllers); a clock that drives the sampling process; and reference voltages that define the measurable voltage range. Higher resolution ADCs provide greater accuracy by

allowing more precise digital representations of the analog input.

The ADC conversion process involves three main steps:

- Sampling: The ADC takes regular samples of the analog signal at specific intervals determined by the sampling rate, which depends on the required information frequency.
- Quantization: Each sampled analog value is assigned a discrete digital value by dividing the analog range into finite intervals and mapping each sample to the nearest interval.
- Encoding: The quantized values are converted into binary format, with the number of bits corresponding to the ADC's resolution. For example, an 8-bit ADC can represent 256 distinct digital values.

ADCs employ different techniques for sampling and quantization, primarily categorized into:

- Successive Approximation ADCs (SAR ADCs): These use a binary search algorithm to iteratively approximate the input analog signal's value, making them common in microcontrollers like the ESP32-C3 due to their balance of speed and accuracy.
- Delta-Sigma (ΔΣ) ADCs: These oversample the input signal and use feedback loops to achieve high-resolution conversions, making them ideal for precision-critical applications such as audio and instrumentation.

Microcontrollers typically have more analog input pins than available ADC instances, meaning each pin does not have a dedicated ADC. To efficiently manage multiple analog inputs without requiring separate ADCs for each, microcontrollers use multiplexing. An input multiplexer selects one analog channel at a time, allowing the single ADC to sequentially sample and convert multiple signals. This approach conserves space and reduces costs while enabling the handling of multiple analog inputs.

ADCs support various conversion modes to accommodate different application needs:

- One-Shot Mode: Also known as single conversion mode, the ADC performs a single conversion and then stops until triggered again. This mode is energy-efficient and suitable for occasional sampling where precise timing is not critical.
- Continuous Mode: The ADC continuously performs conversions as long as it is powered and enabled, providing a steady stream of digital data. This mode is ideal for real-time data acquisition and processing but consumes more power.
- Scan Mode: The ADC sequentially samples multiple analog input channels, converting each into digital form. This mode is useful for systems with multiple sensors, allowing efficient conversion without individual triggers. Scan mode can operate in either one-shot or continuous manners.

RTOS version

```
let peripherals = Peripherals::take().unwrap();

// Creating an ADC Instance

let adc1 = AdcDriver::new(peripherals.adc1).unwrap();

/*
    Configure ADC Pin(s)/Channel(s)
    - Attenuation reduces the input signal's amplitude to fit within the ADC's
      reference voltage range.
    - Resolution determines the precision of the digital representation of
      the analog signal, with common options being 8, 10, 12, or 14 bits.
*/

let config = AdcChannelConfig {
    attenuation: DB_11,
    calibration: Calibration::Curve,
    resolution: Resolution::Resolution12Bit,
};
```

```rust
// Instantiate ADC channel

let mut channel = AdcChannelDriver::new(&adc1, peripherals.pins.gpio4, &config)
    .unwrap();

/*
    Reading an ADC measurement
    The code initiates a one-shot ADC measurement using the `read_raw` method,
    which returns a raw digital value (`u16`) representing the sampled analog
    signal. To convert this digital value back to a physical parameter
    (e.g., voltage), further calculations are necessary. For convenience,
    the `AdcDriver` also provides a `read` method that directly returns the
    measured voltage in millivolts, simplifying the process of interpreting
    ADC readings.

    Currently, the ESP-IDF Hardware Abstraction Layer (HAL) does not
    offer high-level interfaces for interrupt-based operations.
*/

let sample: u16 = channel.read_raw().unwrap();
```

Bare-metal version

```rust
let peripherals = esp_hal::init(Config::default());

// Create instance for ADC configuration parameters

let mut adc_config = AdcConfig::new();

// Enable a pin with attenuation

let mut adc_pin = adc_config.enable_pin(
    pin_instance,
    Attenuation::11dB,
);

// Create ADC driver for ADC1

let mut adc1 = Adc::new(peripherals.ADC1, adc_config);

// Blocking read of input

let adc_reading: u16 = nb::block!(adc1.read_oneshot(&mut pin)).unwrap();
```

3.3 Timers & Counters

Timers and counters are fundamental peripherals in embedded systems, offering powerful functionalities despite their simplicity. A timer generates periodic or one-time signals at specified intervals and can measure the time between external hardware events. Common applications include triggering interrupts for updating displays, measuring button press durations, or creating delays. Conversely, a counter increments its count with each event occurrence, useful for tracking the number of button presses, motor revolutions, or network packets received. While timers and counters can be implemented in software, hardware implementations

are preferred for maintaining high accuracy and efficiency. Most microcontrollers, including the ESP32-C3, come equipped with multiple dedicated timers and counters, each offering various features tailored to specific tasks.

Timers and counters share a similar core circuitry consisting of a count register that increments with each clock event. The primary difference lies in the nature of the clock input: timers use a synchronous periodic clock signal to measure time intervals, whereas counters use asynchronous event-based signals to tally occurrences. Advanced microcontroller timers, such as those in the ESP32-C3, include additional features to enhance functionality:

- Interrupts: Notify the processor of overflow events without continuous polling.
- Auto Reload: Automatically reload a predefined value upon overflow for repetitive timing tasks.
- Clock Source Configuration: Adjust the timer's clock frequency using prescalers to achieve desired timing resolutions.
- Upcounting/Downcounting: Configure the timer to count upwards or downwards based on application needs.
- Cascading Counters: Combine multiple counters to extend the maximum count range beyond a single register's capacity.

These features enable timers and counters to handle complex tasks efficiently, such as generating precise time delays, managing periodic interrupts, or tracking high-frequency events.

Timers and counters support various modes of operation to cater to different application requirements:

- One-Shot Mode: The timer counts up or down once until it overflows, then stops. Ideal for generating single pulses or measuring single events.
- Continuous Mode: The timer continuously counts without stopping, suitable for ongoing measurements or periodic interrupts.
- Input Capture: Captures the timer's current count value upon external events, useful for measuring event durations or signal frequencies.
- Output Compare: Triggers an action when the timer's count matches a predefined value, enabling waveform generation or synchronized actions.
- Pulse Width Modulation (PWM): A specialized form of output compare that generates waveforms with adjustable duty cycles, commonly used for motor control, LED dimming, and signal generation.

RTOS version

```
let peripherals = Peripherals::take().unwrap();

// Creating a timer with default configuration

let timer = TimerDriver::new(peripherals.timer00, &Config::new())
    .unwrap();

// Set start/reset count value to zero

timer.set_counter(0_u64).unwrap();

// Set timer to generate an alarm when the count reaches 1000

timer.set_alarm(1000_u64).unwrap();



// Enable the alarm to occur

timer.enable_alarm(true).unwrap();

// Enable timer to start counting

timer.enable(true).unwrap();
```

```rust
/*
    To utilize timers effectively, developers typically set a start value
    and enable the timer to begin counting. The timer will continue counting
    until it reaches its maximum value, resets to the start value,
    or matches a predefined compare value (alarm). The `TimerDriver` methods
    facilitate these actions, allowing for precise control over the timer's
    behavior.
*/

// Initialize timer clock value

let timer_clk = 1_000_000_u64; // 1 MHz clock


// Enable timer

some_timer.enable(true).unwrap();

loop {
    // Reset timer to start from 0
    some_timer.set_counter(0_u64).unwrap();
    // Execute tasks
    // Read counter value
    let count = some_timer.counter().unwrap();
    // Convert to seconds
    let count_secs = count / timer_clk;
    log::info!("Elapsed time in seconds is {}", count_secs);
}

/*
    Using interrupts allows the timer to notify the application when an alarm or
    overflow event occurs, eliminating the need for continuous polling. This is
    achieved by configuring an Interrupt Service Routine (ISR) that gets called
    upon timer events.
*/

fn timer_alarm_callback() {
    // Handle the timer alarm event
}

fn main() -> ! {
    let peripherals = Peripherals::take().unwrap();
    // Configure timer
    let mut some_timer = TimerDriver::new(peripherals.timer00, &Config::new())
        .unwrap();
    // Set start/reset count value to zero
    some_timer.set_counter(0_u64).unwrap();
    // Set timer to generate an alarm if its value reaches 1000
    some_timer.set_alarm(1000_u64).unwrap();
    // Enable the alarm to occur
    some_timer.enable_alarm(true).unwrap();
    // Attach the ISR to the timer interrupt
    unsafe { some_timer.subscribe(timer_alarm_callback).unwrap() }
```

```
    // Enable interrupts
    some_timer.enable_interrupt().unwrap();
    // Enable timer to start counting
    some_timer.enable(true).unwrap();

    loop {
        // Main application logic
    }
}
```

Bare-metal version

```
// Create a global variable for timer to pass between threads

static G_TIMER: Mutex<RefCell<Option<Timer<Timer0<TIMG0>, esp_hal::Blocking>>>
    = Mutex::new(RefCell::new(None));

#[handler]
fn tg0_t0_level() {
    critical_section::with(|cs| {
        // Clear timer interrupt pending flag
        G_TIMER
            .borrow_ref_mut(cs)
            .as_mut()
            .unwrap()
            .clear_interrupt();
        // Re-activate Timer alarm for interrupts to occur again
        G_TIMER
            .borrow_ref_mut(cs)
            .as_mut()
            .unwrap()
            .set_alarm_active(true);
    });
}

#[entry]
fn main() -> ! {
    let peripherals = esp_hal::init(esp_hal::Config::default());

    // Instantiate TimerGroup0
    let timer_group0 = TimerGroup::new(peripherals.TIMG0);

    // Instantiate Timer0 in timer group0
    let timer0 = timer_group0.timer0;




    // Configure timer to trigger an interrupt every second
    // Load count equivalent to 1 second
    timer0
        .load_value(MicrosDurationU64::micros(1_000_000))
```

```
        .unwrap();
    // Enable alarm to generate interrupts
    timer0.set_alarm_active(true);
    // Activate counter
    timer0.set_counter_active(true);
    // Attach interrupt and start listening for timer events
    timer0.set_interrupt_handler(tg0_t0_level);
    timer0.listen();
    // Move the timer to the global context
    critical_section::with(|cs| {
        G_TIMER.borrow_ref_mut(cs).replace(timer0)
    });

    loop {
        // Reset Timer count (to count from 0)
        timer0.reset();
        // Perform Some Operations
        // Determine Duration
        let dur = some_timer.now().duration_since_epoch().to_secs();
        log::info!("Elapsed Timer Duration in Seconds is {}", dur);
    }
}
```

3.4 PWM

Pulse Width Modulation (PWM) is a waveform generation technique that controls the duration of the "on" time within each period of a square wave signal. Unlike traditional square waves, which have a fixed 50% duty cycle (equal on and off times), PWM allows the on-time to vary between 0% (completely off) and 100% (always on). This variability in the duty cycle—the ratio of on-time to the total period—enables precise control over the average voltage and current delivered to electronic components. For example, a PWM signal with a 50% duty cycle and a peak voltage of 5V results in an average voltage of 2.5V.

The duty cycle is a crucial parameter in PWM, representing the percentage of time the signal is in the "on" state within a single period. Adjusting the duty cycle directly affects the average voltage of the PWM signal. For instance, increasing the duty cycle raises the average voltage, while decreasing it lowers the average voltage. This property makes PWM highly effective for applications that require varying power levels, such as controlling LED brightness, motor speeds, servo positions, and heating elements. By rapidly switching the signal on and off, PWM can simulate analog voltage levels, allowing digital systems to interface seamlessly with analog components.

The ESP32-C3 microcontroller generates PWM signals using dedicated peripherals separate from its general-purpose timers. Specifically, the ESP32-C3 includes two peripherals for PWM generation:

- LED PWM Controller (LEDC): Primarily designed for LED control, the LEDC peripheral can generate PWM signals with high precision. It offers six independent PWM generators (channels) driven by four timers. Each timer can be independently configured for clock and counting, while PWM channels select one of these timers as their reference. The PWM outputs are then connected to GPIO pins to produce the desired waveform.
- Remote Control Peripheral (RMT): While not the focus of this chapter, the RMT peripheral also supports PWM generation and can be used for various remote control and communication applications.

The LEDC peripheral's flexibility allows it to handle multiple PWM channels simultaneously, each with its own configuration, making it suitable for a wide range of applications beyond just LED control.

RTOS version

```
/*
    Configure Timer0 with a clock of 50Hz and a resolution of 14 bits
    The `TimerConfig` struct includes:
    - frequency: Defines the desired timer clock frequency.
    - resolution: Specifies the timer's counter width using an enumeration.
    - speed_mode: Specifies the timer speed mode.
*/

let peripherals = Peripherals::take().unwrap();

let timer_driver = LedcTimerDriver::new(
    peripherals.ledc.timer0,
    &TimerConfig::default()
        .frequency(50.Hz())
        .resolution(Resolution::Bits14),
).unwrap();

// Creating an LEDC PWM channel instance

let mut driver = LedcDriver::new(
    peripherals.ledc.channel0,
    timer_driver,
    peripherals.pins.gpio7,
).unwrap();

// Set desired duty cycle

driver.set_duty(1000_u32).unwrap();

// Get maximum possible duty value

let max_duty = driver.get_max_duty();

// Set duty cycle to 20%

driver.set_duty(max_duty * 20 / 100).unwrap();

// Enable PWM output

driver.enable().unwrap();
```

Bare-metal version

```
let peripherals = esp_hal::init(Config::default());
let some_output_pin = Output::new(peripherals.GPIO3, Level::Low);
let mut ledc = Ledc::new(peripherals.LEDC);
ledc.set_global_slow_clock(LSGlobalClkSource::APBClk);
let ledctimer = ledc.get_timer::<LowSpeed>(ledc::timer::Number::Timer0);
ledctimer.configure(timer::config::Config {
    duty: timer::config::Duty::Duty12Bit,
        clock_source: timer::LSClockSource::APBClk,
        frequency: 4u32.kHz(),
```

```
    })
    .unwrap();
let mut channel = ledc.get_channel(channel::Number::Channel0, some_output_pin);
let mut channel0 = ledc.get_channel(channel::Number::Channel0, led);
channel0.configure(channel::config::Config {
    timer: &ledctimer,
        duty_pct: 10,
        pin_config: channel::config::PinConfig::PushPull,
    })
    .unwrap();
```

3.5 Serial communication

Serial communication is a method of transmitting data sequentially over a single wire or a pair of wires, sending one bit at a time. This contrasts with parallel communication, where multiple bits are transmitted simultaneously across multiple channels. Serial communication is favored in embedded systems, microcontrollers, and various electronic devices due to its simplicity, reliability, and efficiency. The two primary modes of serial communication are I2C (Inter-Integrated Circuit) and SPI (Serial Peripheral Interface), each with its own advantages. I2C is a synchronous protocol designed for communication between microcontrollers and peripheral devices, offering a smaller footprint but lower bandwidth compared to SPI. SPI, also synchronous, is commonly used for short-distance communication between microcontrollers and peripherals, providing higher speed and bandwidth.

UART stands for Universal Asynchronous Receiver/Transmitter and is a widely used serial communication interface that transmits and receives data asynchronously. This means that UART communication does not rely on a shared clock signal between the transmitter and receiver. Instead, both devices must agree on a specific baud rate—the number of bits transmitted per second—to ensure accurate data transmission. UARTs are prevalent in computers, microcontrollers, and embedded systems for communicating with other devices or computers over serial connections. In the ESP32-C3 microcontroller, UART is utilized for serial monitoring, enabling functionalities such as the `log::info!()` macro for debugging and logging.

A UART communication channel typically consists of two main components: a transmitter and a receiver connected via a single wire for each direction. The transmitter converts parallel data from the device into a serial format, adding start and stop bits to indicate the beginning and end of each data packet. The receiver then converts the incoming serial data back into a parallel format for the device to process. UART communication can operate in full-duplex mode, allowing simultaneous transmission and reception of data using separate wires for each direction, or in half-duplex mode, where data transmission and reception occur alternately over a single wire.

For successful UART communication, both the transmitter and receiver must have matching configurations. Key configuration options include:

- Idle State: Determines the default state of the communication line when no data is being transmitted. For example, if the idle state is high, the receiver expects a transition from high to low to indicate the start of a transmission.
- Baud Rate: Specifies the number of bits transmitted per second. Both devices must use the same baud rate to ensure data integrity. Mismatched baud rates can lead to incorrect data sampling and communication errors.
- Data Bits: Defines the number of data bits per frame, commonly set to 8 or 9 bits. The number of data bits must be consistent between the transmitter and receiver.
- Parity: An optional error-checking mechanism that can be set to odd, even, or none. Parity helps detect errors in transmitted data by adding an extra bit based on the number of set bits.
- Stop Bits: Indicates the end of a data frame. UART frames typically include one or two stop bits to mark the conclusion of data transmission.
- Flow Control: An optional feature that manages the rate of data transmission to prevent buffer overflow. Hardware flow control uses additional lines to signal the transmitter to pause or resume sending data based on the receiver's buffer status.

Serial communication can be categorized into asynchronous and synchronous modes:

- Asynchronous Serial Communication: In this mode, data is transmitted without a shared clock signal between the sender and receiver. Instead, both devices must agree on a baud rate to synchronize data transmission. UART is a prime example of asynchronous communication, relying on start and stop bits to frame data packets.

- Synchronous Serial Communication: This mode uses a shared clock signal to synchronize data transmission between devices, eliminating the need for start and stop bits. Synchronous protocols like SPI and I2C rely on a common clock to ensure that data bits are transmitted and received accurately and efficiently.

Several serial communication protocols are commonly employed in embedded systems, each suited to different use cases based on their characteristics:

- UART (Universal Asynchronous Receiver/Transmitter): A popular asynchronous protocol used for point-to-point communication between devices. UART is ideal for simple, low-speed data transmission tasks such as debugging, logging, and interfacing with serial peripherals.
- I2C (Inter-Integrated Circuit): A synchronous protocol designed for communication between multiple devices using only two wires (SDA for data and SCL for clock). I2C is suitable for connecting sensors, EEPROMs, and other low-speed peripherals to microcontrollers.
- SPI (Serial Peripheral Interface): A high-speed synchronous protocol that uses four wires (MOSI, MISO, SCLK, and SS) for communication between a master device and one or more slave devices. SPI is ideal for applications requiring faster data transfer rates, such as interfacing with flash memory, displays, and high-speed sensors.

RTOS version

```rust
let peripherals = Peripherals::take().unwrap();

/*
    Configure a UART instance
    - uart: An instance of a UART peripheral.
    - tx: An output pin for transmitting serial bits.
    - rx: An input pin for receiving serial bits.
    - cts: (Optional) A pin for Clear To Send control flow.
    - rts: (Optional) A pin for Request To Send control flow.
    - config: A reference to a UART configuration.
*/

let tx = peripherals.pins.gpio5;
let rx = peripherals.pins.gpio6;
let config = config::Config::new().baudrate(Hertz(115_200));
let uart = UartDriver::new(
    peripherals.uart1,
    tx,
    rx,
    Option::<gpio::Gpio0>::None,
    Option::<gpio::Gpio1>::None,
    &config,
).unwrap();


// Sending a single byte over UART

uart.write(&[25_u8]).unwrap();

// Receiving a single byte over UART

let mut buf = [0_u8; 1];

// The `BLOCK` constant ensures that the method waits for data.
```

```
uart.read(&mut buf, BLOCK).unwrap();

/*
    Configure a I2C instance
    - i2c: An instance of an I2C peripheral.
    - sda: A bidirectional pin instance for the Serial Data Line.
    - scl: A bidirectional pin instance for the Serial Clock Line.
    - config: A reference to an I2C configuration.
*/

let i2c = peripherals.i2c0;
let config = I2cConfig::new().baudrate(100.kHz().into());
let i2c_driver = I2cDriver::new(i2c, sda, scl, &config).unwrap();

// Sending a single byte over I2C to address 0x65

i2c_driver.write(0x65, &[25], BLOCK).unwrap();

// Receiving a single byte over I2C from address 0x65

let mut buf = [0_u8; 1];

i2c_driver.read(0x65, &mut buf, BLOCK).unwrap();
```

Bare-metal version

```
let peripherals = esp_hal::init(Config::default());

let uart_tx = Output::new(peripherals.GPIO21, Level::Low);

let uart_rx = Input::new(peripherals.GPIO20, Pull::Up);

let uart_config = Config {
    baudrate: 115200,
    data_bits: DataBits::DataBits8,
    parity: Parity::ParityNone,
    stop_bits: StopBits::STOP1,
    clock_source: ClockSource::Apb,
    ..Default::default()
};

let mut uart = Uart::new_with_config(
    peripherals.UART0,
    uart_config,
    io.pins.gpio21,
    io.pins.gpio20,
)
.unwrap();

uart.write_bytes(&[25_u8]).unwrap();

let mut buf = [0_u8; 1];
```

```
uart.read(&mut buf).unwrap();


let i2c = I2c::new(peripherals.I2C0, master::Config::default())
        .unwrap()
        .with_sda(peripherals.GPIO1)
        .with_scl(peripherals.GPIO2);

i2c.write(0x65, &[25]).unwrap();

let mut buf = [0_u8; 1];
i2c.read(0x65, &mut buf).unwrap();
```

3.6 Networking

ESP devices have gained popularity for their robust connectivity and Internet of Things (IoT) capabilities, supported by both software and hardware innovations. In the Rust programming environment, the `esp-idf-svc` and `embedded-svc` crates provide comprehensive support for a wide range of networking services, including WiFi, Ethernet, HTTP client & server, MQTT, WebSockets (WS), Network Time Protocol (NTP), and Over-The-Air (OTA) updates. Establishing network access is a fundamental step for any IoT service, with WiFi being a common protocol used for this purpose. This section introduces the basics of programming WiFi, focusing on establishing a simple connection rather than delving into intricate configuration details, to maintain clarity and avoid code verbosity.

RTOS version

```
let peripherals = Peripherals::take().unwrap();

// Connect to WiFi

let sysloop = EspSystemEventLoop::take()?;
let nvs = EspDefaultNvsPartition::take()?;
let wifi = EspWifi::new(peripherals.modem, sysloop, Some(nvs))?;
wifi.set_configuration(&Configuration::Client(ClientConfiguration {
    ssid: "SSID".try_into().unwrap(),
    password: "PASSWORD".try_into().unwrap(),
    auth_method: AuthMethod::None,
    ..Default::default()
}))?;
wifi.start()?;
wifi.connect()?;
wifi.wait_netif_up()?;

// Configure an HTTP connection

let httpconnection = EspHttpConnection::new(&Configuration {
    use_global_ca_store: true,
    crt_bundle_attach: Some(esp_idf_sys::esp_crt_bundle_attach),
    ..Default::default()
})?;

// Exchanging requests & responses
```

```
let url = "https://blog.pl";
let _request = httpconnection.initiate_request(Method::Get, url, &[])?;
httpconnection.initiate_response()?;

let status_code = httpconnection.status();
let status_msg = httpconnection.status_message();
if let Some(content_type) = httpconnection.header("Content-Type") {
    // Process the Content-Type header
}



// Create and configure an HTTP server

let httpserver = EspHttpServer::new(&Configuration::default())?;

httpserver.fn_handler("/", Method::Get, |request| {
    let mut response = request.into_ok_response()?;
    response.write("content".as_bytes())?;
    Ok::<(), anyhow::Error>(())
})?;

// Using SNTP

let ntp = EspSntp::new_default().unwrap();
while ntp.get_sync_status() != SyncStatus::Completed {
    // Wait or perform other tasks
}
let current_time = SystemTime::now();
log::info!("Synchronized System Time: {:?}", current_time);
```

Chapter 4

Setup

4.1 Environment

- Install Rust
- Install Python
- Install git
- Install espup, toolchains and tools:

```
cargo install espup
espup install
cat espup install >> .profile
sudo apt-get install git wget flex bison gperf python3 python3-pip python3-venv
     cmake ninja-build ccache libffi-dev libssl-dev dfu-util libusb-1.0-0
cargo install ldproxy
cargo install esp-generate
sudo apt install libssl-dev openssl pkg-config
cargo install cargo-generate
cargo install cargo-espflash
cargo install espflash
```

- Set up the environment variables

```
cat $HOME/export-esp.sh >> [path to profile]
```

4.2 Creating and building project

```
cargo generate esp-rs/esp-idf-template cargo
cargo build
```

Chapter 5

Common questions

1. **Why is `sys::link_patches()` added at the main method level?** In embedded Rust, when working with bindings to C libraries (such as when using `bindgen` to generate Rust bindings for C code), you might come across `sys::link_patches()` in your code. This function is often needed for the following reasons:

- Forcing linker to include symbols: Some embedded C libraries use weak symbols or link-time optimizations that may cause unused functions or variables to be discarded by the linker. Calling `sys::link_patches()` ensures that those symbols are referenced, preventing them from being removed.
- Ensuring correct initialization: In some cases, C libraries require certain initialization routines to be linked in, even if they are not directly used in Rust. Calling `sys::link_patches()` forces the inclusion of these routines.
- Overcoming dead code elimination: The Rust compiler aggressively removes unused code (LLVM optimizations). If your embedded system relies on certain weakly linked symbols, this function helps keep them included.
- Platform-specific fixes: Some embedded platforms might require linking patches to ensure compatibility between Rust and C functions.

2. **What does the `Ordering::Relaxed` parameter mean when used with atomic types?** In Rust, when working with atomic types (`AtomicUsize`, `AtomicBool`, etc.), you often need to specify an ordering constraint to control how operations are synchronized across multiple threads. `Relaxed` is one such ordering constraint. Using `Ordering::Relaxed` means that the operation is atomic but does not impose any synchronization guarantees beyond ensuring the operation itself is not torn or reordered within the same thread. In other words:

- The operation executes atomically (i.e., it completes as a single unit).
- No guarantees are made about memory visibility or ordering relative to other memory operations.
- Other threads may not immediately see the updated value.

3. **What is CTS and RTS in UART Communication?** In the context of UART (Universal Asynchronous Receiver-Transmitter) communication, CTS (Clear to Send) and RTS (Request to Send) are hardware flow control signals. CTS (Clear to Send) is an input signal for a device. It indicates whether the device is ready to receive data. When low (0), the device is not ready to receive data. When high (1), the device is ready to receive data. RTS (Request to Send) is an output signal for a device. It indicates that the device wants to send data. The device sets RTS high (1) to tell the other device it is ready to transmit. The other device will then check CTS before sending data. CTS and RTS are used for hardware flow control, preventing data loss in UART communication:

- Device A wants to send data to Device B → It checks CTS from Device B.
- If CTS is high, Device A sends the data.
- If CTS is low, Device A waits before sending data.
- Device A controls RTS to signal when it wants to receive data.

4. **What is BCD?** A Binary-Coded Decimal is a representation of decimal numbers where each digit is stored as a separate 4-bit binary value. Example: Decimal 25 in BCD: 2 → 0010, 5 → 0101 so, 25 in BCD is 0010 0101 (0x25 in hexadecimal).

5. **What is the meaning of macros used in the bare-metal variants?**

- `#![no_std]` - Removes the Rust standard library (std). Embedded systems often lack an OS or runtime that provides standard library features (such as heap allocation, threads, and file I/O). Instead, Rust provides a minimal subset called the core library, which includes essential functionalities like primitives, iterators, and basic math.
- `#![no_main]` - Disables the default Rust entry point (fn main()). The standard Rust main() function relies on an OS-provided runtime, which is not available in a bare-metal context. Instead, the developer must define a custom entry point using other attributes (like #[entry] from cortex-m-rt or #[main] in other frameworks).
- `#[main]` - It is marking the main function as the entry point.