Rust programming for embedded systems

Contents

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.

1.1.1 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.

1.1.2 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.

1.1.3 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.

1.1.4 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.

1.1.5 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.

1.1.6 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.

1.2.1 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.

1.2.2 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.

1.2.3 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.

1.2.4 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.

1.2.5 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.

1.2.6 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.

1.2.7 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 particu-

larly 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:

1. 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.
2. 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.
3. 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.
4. 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.
5. 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:

1. 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.
2. 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.
3. 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++.
4. 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.
5. 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.
6. 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.
7. 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.

8. 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.

1.5.1 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.

1.5.2 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:

1. 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.
2. 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.
3. 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.

1.5.3 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: 1. 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. 2. 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. 3. 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.

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:

1. 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.
2. 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.
3. 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: 1. Interrupt Source: The peripheral generating the event, such as an ADC completion or a GPIO button press. 2. 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. 3. 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.

3 GPIO (std)

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.
  - *Example:* In a doorbell system, a GPIO pin connected to a doorbell button goes high when pressed, triggering the doorbell to ring.
- Active Low: A low voltage level represents a “true” state.
  - *Example:* In a burglar alarm system, a window sensor pulls the GPIO pin low when a window is opened, activating the alarm.

Understanding whether a GPIO pin uses an active high or active low state is crucial for correctly setting up circuits with sensors, switches, and other digital devices, ensuring the microcontroller responds appropriately to changes in the external environment.

3.0.1 Configuring GPIO

Configuring General Purpose Input/Output (GPIO) pins in embedded Rust involves several systematic steps to ensure proper interaction between the microcontroller and external devices. The process leverages Rust's safety features and patterns, such as the singleton pattern, to manage hardware resources efficiently.

3.0.1.1 Take the Peripherals

- Singleton Pattern: Ensures only one instance of each peripheral exists throughout the application.
- Implementation:

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

  - `Peripherals::take()` returns an `Option`, ensuring that subsequent calls return `None`, maintaining a single instance.

3.0.1.2 Configure Pin Direction

- PinDriver Struct: Utilized to set a pin as either input or output.
- Options:
  1. Input Configuration:

```
// Allows the microcontroller to read the voltage level on the pin
let some_pin = PinDriver::input(device_per.pins.gpio3).unwrap();
```

  2. Output Configuration:

```
// Enables the microcontroller to control the voltage level on the pin
let some_pin = PinDriver::output(device_per.pins.gpio2).unwrap();
```

- Type Safety: Pins have specific types (`Input` or `Output`) that restrict available methods based on their configuration.

3.0.1.3 Configure Pin Pull (Input Pins Only)

- Pull-Up/Pull-Down Resistors: Stabilize input pin states.
- Method:

```
some_pin.set_pull(Pull::Up).unwrap();   // Pull-up configuration
some_other_pin.set_pull(Pull::Down).unwrap(); // Pull-down configuration
```

3.0.1.4 Configure Pin Drive (Output Pins Only)

- Drive Modes:
  - Push-Pull (Default): Can drive the pin both high and low.
  - Open-Drain: Can only pull the pin low.
- Method to Set Open-Drain:

```
some_output_pin.into_output_od().unwrap();
```

- Alternative Instantiation:

```
PinDriver::output_od(device_per.pins.gpioX).unwrap();
```

3.0.1.5 Configure Interrupt Type (Input Pins Only)

- Interrupt Detection Types:
  - Edge-Triggered: Detects rising, falling, or both edges.
  - Level-Triggered: Detects high or low voltage levels.
- Configuration Example:

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

3.0.1.6 Configure Drive Strength (Output Pins Only)

- Drive Strength Options: Varying current capabilities (e.g., 5mA, 10mA, 20mA, 40mA).
- Configuration Example:

```
some_pin.set_drive_strength(DriveStrength::I5mA).unwrap();
some_pin.set_drive_strength(DriveStrength::I10mA).unwrap();
some_pin.set_drive_strength(DriveStrength::I20mA).unwrap();
some_pin.set_drive_strength(DriveStrength::I40mA).unwrap();
```

3.0.1.7 Interacting with GPIO

1. Writing/Controlling Output:
   - Methods: `set_low()` and `set_high()`
   - Example:

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

2. Reading Input by Polling:
   - Methods: `is_high()` and `is_low()`
   - Example:

```
loop {
    if some_pin.is_low() {
        println!("Input is low!");
    }
    if some_pin.is_high() {
        println!("Input is high!");
    }
}
```

   - Note: Utilizes a continuous loop to repeatedly check the pin state.
3. Reading Input by Interrupts:
   - Interrupt Service Routines (ISRs): Functions triggered by specific GPIO events.
   - Configuration Steps:
     1. Set Interrupt Type.
     2. Subscribe ISR to Interrupt:

```
unsafe { some_pin.subscribe(gpio_int_callback).unwrap() }
```

     3. Enable Interrupt:

```
some_pin.enable_interrupt().unwrap();
```

4. Define ISR:

```
fn gpio_int_callback() {
   // ISR code
}
```

- Example Structure:

```
fn main() -> ! {
    let dp = Peripherals::take().unwrap();
    let some_pin = PinDriver::input(dp.pins.gpio0).unwrap();
    some_pin.set_pull(Pull::Up).unwrap();
    some_pin.set_interrupt_type(InterruptType::PosEdge).unwrap();
    unsafe { some_pin.subscribe(gpio_int_callback).unwrap() }
    some_pin.enable_interrupt().unwrap();

    // Application code...
}
```

- ESP32-C3 Specifics: Interrupts are disabled after an event and must be re-enabled within the ISR or processing code to handle subsequent events.

4 GPIO (no-std)

4.0.0.1 Initialize the ESP & Gain Access to Peripherals

Before utilizing any device peripherals, it's essential to configure the ESP device itself. This involves setting up the device clocks and gaining access to peripheral instances using the singleton pattern, which ensures that only one instance of each peripheral is accessed throughout the application. The `esp-hal` crate facilitates this initialization with a streamlined approach.

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

The `init` function takes an `esp_hal::Config` struct as an argument and returns instances of the peripherals and system clocks. Using the default configuration simplifies the setup process by applying standard settings.

```
pub struct Config {
    pub cpu_clock: CpuClock,
    pub watchdog: WatchdogConfig,
}
```

This configuration struct allows for customization of system parameters such as CPU clock speed and watchdog settings, although the default values are typically sufficient for basic applications.

4.0.0.2 Create an IO Driver

To control GPIO pins, an IO driver must be instantiated. The `esp_hal::gpio` module provides the `Io` struct, which serves as the driver for managing individual IO pins.

```
let io = Io::new(peripherals.GPIO, peripherals.IO_MUX);
```

This code creates a new IO driver instance by passing the GPIO and IO_MUX peripherals obtained during initialization. The IO driver provides access to the individual GPIO pins for further configuration.

4.0.0.3 Configure Pin Direction

After establishing the IO driver, each GPIO pin must be configured as either an input or an output. This configuration determines how the pin will interact with external components.

4.0.0.3.1 1. Input Configuration

Configuring a pin as an input allows the microcontroller to read its state. This is achieved using the `Input` struct, which requires specifying the pin and its pull configuration.

```
pub fn new(pin: impl Peripheral<P = P> + 'd, pull: Pull) -> Self
```

```
let some_input_pin = Input::new(io.pins.gpio3, Pull::Up);
```

In this example, GPIO3 is configured as an input with a pull-up resistor, ensuring that the pin reads a high logic level when inactive.

4.0.0.3.2 2. Output Configuration

Configuring a pin as an output enables the microcontroller to control its state, setting it to either high or low.

```
pub fn new(pin: impl Peripheral<P = P> + 'd, initial_output: Level) -> Self
```

```
let some_output_pin = Output::new(io.pins.gpio3, Level::Low);
```

Here, GPIO3 is set as an output with an initial low level. The push-pull configuration allows the pin to actively drive the signal both high and low.

4.0.0.4 (Output Pins Only): Configure Drive Strength

While optional, configuring the drive strength of output pins can be necessary for applications requiring higher current levels. This is done using the `set_drive_strength` method, which selects the desired drive strength from the available options.

```
// Configure a pin with a 5mA Drive
some_pin.set_drive_strength(DriveStrength::I5mA);
// Configure a pin with a 10mA Drive
some_pin.set_drive_strength(DriveStrength::I10mA);
// Configure a pin with a 20mA Drive
some_pin.set_drive_strength(DriveStrength::I20mA);
// Configure a pin with a 40mA Drive
some_pin.set_drive_strength(DriveStrength::I40mA);
```

These configurations adjust the current-driving capability of the GPIO pin, allowing it to handle different levels of electrical load as required by the application.

4.0.1 Interacting with GPIO

Once the GPIO pins are configured, they can be interacted with through various methods depending on their direction (input or output).

4.0.1.1 Writing/Controlling Output

Output pins can be controlled by setting their state to high or low using the `set_high` and `set_low` methods.

```
// Set pin output to low
some_pin.set_low();
// Set pin output to high
some_pin.set_high();
```

These methods allow the microcontroller to manipulate the voltage level on the output pin, enabling control over connected devices such as LEDs or relays.

4.0.1.2 Reading Input by Polling

Input pins can be read by continuously polling their state using the `is_high` and `is_low` methods within a loop.

```
loop {
    // Check if input pin is low
    if some_pin.is_low() {
        println!("Input is low!");
    }
    // Check if input pin is high
    if some_pin.is_high() {
        println!("Input is high!");
    }
}
```

This approach involves repeatedly checking the pin's state, which can be resource-intensive but is straightforward to implement.

4.0.1.3 Reading Input by Interrupts

Interrupts provide a more efficient way to handle input changes by triggering an Interrupt Service Routine (ISR) when specific events occur, such as a button press.

```
#![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));

// ISR Definition
#[handler]
fn gpio() {
    // Start a Critical Section
    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();
    });
}

#[entry]
fn main() -> ! {
    // Take Peripherals & Configure Device
    let peripherals = esp_hal::init(esp_hal::Config::default());
    // Create IO Driver
```

```
    let io = Io::new(peripherals.GPIO, peripherals.IO_MUX);

    // Interrupt Configuration
    // Register interrupt handler
    io.set_interrupt_handler(gpio);
    // Configure pin direction
    let some_pin = Input::new(io.pins.gpio0, Pull::Up);
    // Configure input to trigger an interrupt on the falling edge
    // and start listening to events
    some_pin.listen(Event::FallingEdge);
    // Now that pin is configured, move the pin to the global context
    critical_section::with(|cs| G_PIN.borrow_ref_mut(cs).replace(some_pin));

    // Following Application Code
    loop {}
}
```

In this example:

1. Global Variable Setup: A global `G_PIN` variable is defined using a `Mutex` and `RefCell` to safely share the GPIO input pin between the main thread and the ISR.
2. ISR Definition: The `gpio` function is marked with the `#[handler]` attribute and serves as the ISR. It clears the interrupt flag to allow future interrupts.
3. Main Function:
   - Initialization: Peripherals are initialized, and an IO driver is created.
   - Interrupt Handler Registration: The ISR is registered to handle GPIO events.
   - Pin Configuration: GPIO0 is set as an input with a pull-up resistor and configured to trigger an interrupt on a falling edge.
   - Global Context Assignment: The configured input pin is moved to the global context within a critical section to ensure thread-safe access.

5 ADCs (std)

5.0.1 Analog Nature of the Physical World and the Need for 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.

5.0.2 ADC Conversion Process: Sampling, Quantization, and Encoding

The ADC conversion process involves three main steps: 1. 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. 2. 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. 3. 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.

5.0.3 Types of ADCs: Successive Approximation and Delta-Sigma

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 ($\Delta\Sigma$) 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.

5.0.4 Input Multiplexing in Microcontrollers

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.

5.0.5 ADC Conversion Modes: One-Shot, Continuous, and Scan

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.

5.0.6 Configuring ADCs

5.0.6.1 Take the Peripherals The initial step in configuring Analog-to-Digital Converters (ADCs) involves initializing the necessary peripherals, akin to the process used for setting up General-Purpose Input/Output (GPIO) pins. This setup is essential for all peripherals involved in ADC operation and ensures that each component is correctly prepared before moving on to subsequent configuration stages.

5.0.6.2 Configure an ADC Instance Not all pins on a microcontroller support analog functions. For the ESP32-C3, specific analog pins are mapped to particular ADC instances, as detailed in the device's reference manual. For example, using GPIO4 requires configuring ADC1. Multiple pins can share the same ADC instance through an internal multiplexer, allowing a single ADC to handle multiple input channels efficiently. Configuration is performed using the `AdcDriver::new` method, which initializes the desired ADC instance.

```
// Creating an ADC Instance
let adc1 = AdcDriver::new(peripherals.adc1).unwrap();
```

This code snippet demonstrates how to instantiate ADC1 using the `AdcDriver::new` method, preparing it for subsequent configurations.

5.0.6.3 Configure ADC Pin(s)/Channel(s)

After instantiating the ADC, the next step is to configure the specific pins or channels to be used. This involves setting the pin to analog mode and configuring additional parameters such as attenuation and resolution.

- 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.

Configuration is performed using the `AdcChannelDriver::new` method, which requires references to the ADC instance, the specific pin, and the channel configuration settings.

```
// The ADC Channel Config Configuration Struct Definition
pub struct AdcChannelConfig {
    pub attenuation: adc_atten_t,
    pub resolution: Resolution,
    pub calibration: bool,
}
```

This struct defines the configuration parameters for an ADC channel, including attenuation level, resolution, and whether calibration is enabled.

```
// Configure ADC Channel
let ch_config = AdcChannelConfig {
    attenuation: DB_11,
    calibration: true,
    resolution: Resolution::Resolution12Bit,
};

// Instantiate ADC Channel
let mut adc_chan = AdcChannelDriver::new(&adc1, peripherals.pins.gpio4, &ch_config)
  .unwrap();
```

In this example, GPIO4 is configured as an ADC pin with 11 dB attenuation, 12-bit resolution, and calibration enabled. The `AdcChannelDriver::new` method ties the configuration to ADC1 and GPIO4.

5.0.7 Interacting with ADCs

5.0.7.1 Blocking Read of Input

To obtain a measurement from the ADC, a blocking approach can be used where the application initiates a one-shot measurement and waits for the result before continuing execution. This method is straightforward but halts other code execution until the measurement is complete, making it suitable for applications with occasional sampling needs where precise timing is not critical.

```
// Reading an ADC Measurement
let sample: u16 = adc_chan.read_raw().unwrap();
```

This 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.

5.0.7.2 Non-blocking Read of Input by Interrupts

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

However, developers can implement non-blocking reads using Rust's asynchronous programming features or by accessing lower-level bindings through `esp-idf-sys`. Non-blocking reads allow the application to continue executing other tasks while the ADC performs measurements, which is essential for applications requiring real-time data processing.

Configuring and interacting with ADCs on the ESP32-C3 involves several key steps:

1. Peripheral Initialization: Setting up necessary peripherals.
2. ADC Instance Configuration: Selecting and initializing the appropriate ADC instance based on the pin used.
3. ADC Pin/Channel Configuration: Configuring specific pins with appropriate attenuation and resolution settings.
4. Reading ADC Values: Performing measurements either through blocking (synchronous) reads or implementing more complex non-blocking (asynchronous) methods.

6 ADCs (no-std)

6.0.0.1 Initialize the ESP & Gain Access to Peripherals

Before utilizing any device peripherals, the ESP device must be configured, primarily by setting up the device clocks and gaining access to peripheral instances. The singleton pattern is employed to ensure that only one instance of each peripheral is accessed throughout the application, enhancing safety and resource management.

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

The `init` function initializes the device with default configuration values, returning instances of peripherals and system clocks. The `esp_hal::Config` struct allows customization of system parameters such as CPU clock speed and watchdog settings, though the default values are typically sufficient for basic applications.

```
pub struct Config {
    pub cpu_clock: CpuClock,
    pub watchdog: WatchdogConfig,
}
```

This configuration struct enables developers to tailor the system's behavior by adjusting clock speeds and watchdog parameters as needed.

6.0.0.2 Create an IO Driver

With access to peripherals established, the next step is to create an IO driver, which provides control over individual IO pins. The `esp_hal::gpio` module offers the `Io` struct for this purpose.

```
let io = Io::new(peripherals.GPIO, peripherals.IO_MUX);
```

This code initializes the IO driver by passing the GPIO and IO_MUX peripherals obtained during initialization. The IO driver facilitates further configuration and management of specific GPIO pins.

6.0.0.3 Configure Analog Pin and Channel

Not all pins on a microcontroller support analog functions. Depending on the chosen pin, the corresponding ADC instance must be configured. For example, using GPIO4 requires configuring ADC1. Multiple pins can be sampled by the same ADC instance through an internal multiplexer.

After selecting the appropriate pin, the ADC channel configuration is created by specifying the pin and attenuation settings. Attenuation reduces the amplitude of the input signal to fit within the ADC's reference voltage range.

```rust
pub enum Attenuation {
    Attenuation0dB = 0,
    Attenuation2p5dB = 1,
    Attenuation6dB = 2,
    Attenuation11dB = 3,
}
```

```rust
// 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::Attenuation11dB,
);
```

In this example, the ADC channel is configured with an attenuation of 11 dB for a specific pin. Attenuation allows the ADC to handle higher input voltages by scaling them down, ensuring accurate measurements within the ADC's reference voltage range.

6.0.0.4 Create an ADC Driver With the ADC channel configured, an ADC driver can be created to manage the ADC instance and perform measurements. The `esp_hal::analog::adc::Adc` type provides the necessary abstraction for this purpose.

```rust
// Create ADC Driver for ADC1
let mut adc1 = Adc::new(peripherals.ADC1, adc_config);
```

This code initializes an ADC driver for ADC1, associating it with the previously configured ADC channel. The ADC driver enables the application to perform analog measurements through the configured ADC instance.

6.0.1 Interacting with ADCs

Once configured, ADCs can be interacted with to perform measurements. This involves reading analog input values either in a blocking or non-blocking manner.

6.0.1.1 Blocking Read of Input A blocking read involves initiating a one-shot measurement and waiting until the result is available. This ensures that the application receives the measurement before proceeding, albeit at the cost of halting other operations during the wait.

```rust
let adc_reading: u16 = adc1.read_oneshot(&mut analog_pin).unwrap();
```

While labeled as non-blocking, the `read_oneshot` method returns a `WouldBlock` error if the ADC is not ready, indicating that the operation cannot be completed immediately. To implement a true blocking approach, the `block!` macro from the `nb` crate can be used to wait until the measurement is ready.

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

The `block!` macro ensures that the code waits until the ADC reading is available before proceeding, providing a straightforward way to obtain accurate measurements without handling errors manually.

7 Programming Timers & Counters (std)

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.

7.0.1 Counter/Timer Structure and Features

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:

1. Interrupts: Notify the processor of overflow events without continuous polling.
2. Auto Reload: Automatically reload a predefined value upon overflow for repetitive timing tasks.
3. Clock Source Configuration: Adjust the timer's clock frequency using prescalers to achieve desired timing resolutions.
4. Upcounting/Downcounting: Configure the timer to count upwards or downwards based on application needs.
5. 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.

7.0.2 Counter/Timer Modes of Operation

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

1. One-Shot Mode: The timer counts up or down once until it overflows, then stops. Ideal for generating single pulses or measuring single events.
2. Continuous Mode: The timer continuously counts without stopping, suitable for ongoing measurements or periodic interrupts.
3. Input Capture: Captures the timer's current count value upon external events, useful for measuring event durations or signal frequencies.
4. Output Compare: Triggers an action when the timer's count matches a predefined value, enabling waveform generation or synchronized actions.
5. 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.

Timers and counters are versatile peripherals essential for managing time-based and event-based tasks in embedded systems. The ESP32-C3 microcontroller offers robust timer and counter functionalities with features like interrupts, auto-reload, clock source configuration, and various modes of operation. By leveraging these capabilities, developers can implement precise timing mechanisms, event counting, waveform generation, and more. The provided Rust code examples illustrate basic configurations for

one-shot and continuous timer modes, including interrupt handling, enabling developers to integrate timers and counters effectively into their applications.

7.0.3 Configuring Timers

Configuring timers on the ESP32-C3 involves several methodical steps to ensure accurate and efficient timing operations. This process is similar to configuring other peripherals like GPIOs and includes initializing peripherals, setting up timer instances, and configuring specific timer settings.

7.0.3.1 Take the Peripherals The initial step mirrors the peripheral initialization process used for GPIOs, as demonstrated before. This involves acquiring and setting up the necessary peripherals required for timer operation.

7.0.3.2 Configure a Timer Instance The ESP32-C3 chip features two hardware timer groups, each containing a general-purpose hardware timer and a system watchdog timer. These timers are 54-bit wide with 16-bit prescalers and offer functionalities such as auto-reload, alarm generation, up/down counting, and interrupt generation. To configure a timer, the `TimerDriver::new` method from the `esp_idf_hal::timer` module is used. This method requires a Timer peripheral instance (e.g., Timer00) and a reference to a configuration instance.

```
// Creating a Timer Instance
let some_timer = TimerDriver::new(peripherals.timer00, &Config::new())
  .unwrap();
```

In this example, `Timer00` is instantiated using the default configuration, which sets the divider to 80, and both `xtal` and `auto_reload` to `false`. The ESP32-C3 provides two general-purpose timers, `timer00` and `timer01`, which can be chosen based on application requirements.

7.0.3.3 Configure Timer Control Methods After instantiating the timer, various control methods are available to manage its behavior. These methods allow enabling/disabling the timer, setting the counter value, configuring alarms, and more.

```
// Enable or disable the timer
pub fn enable(&mut self, enable: bool) -> Result<(), EspError>

// Manually set the current counter value
pub fn set_counter(&mut self, value: u64) -> Result<(), EspError>

// Enable or disable the alarm feature
pub fn enable_alarm(&mut self, enable: bool) -> Result<(), EspError>

// Set the alarm compare value
pub fn set_alarm(&mut self, value: u64) -> Result<(), EspError>
```

These methods provide granular control over the timer's operation, allowing developers to start, stop, reset, and configure alarm conditions as needed.

```
// Set start/reset count value to zero
some_timer.set_counter(0_u64).unwrap();

// Set timer to generate an alarm when the count reaches 1000
some_timer.set_alarm(1000_u64).unwrap();
```

```
// Enable the alarm to occur
some_timer.enable_alarm(true).unwrap();

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

In this example, the timer is configured to start counting from zero and generate an alarm when the count reaches 1000. The alarm is enabled, and the timer is started to begin counting.

Note: While the ESP32-C3 timers support features like up/down counting, the current `esp-idf-hal` library may not provide high-level methods for these functionalities. Developers may need to use lower-level abstractions or await future library updates to access these features.

7.0.4 Interacting with Timers

7.0.4.1 Controlling Timers/Counters 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 to Start Counting
some_timer.enable(true).unwrap();

loop {
    // Reset Timer Count to start from 0
    some_timer.set_counter(0_u64).unwrap();
    // Perform Some Operations
    // Read Counter Value
    let count = some_timer.counter().unwrap();
    // Convert to Seconds
    let count_secs = count / timer_clk;
    // Print Timer Elapsed Time (from 0)
    println!("Elapsed Timer Duration in Seconds is {}", count_secs);
    // Additional logic can be added here
}
```

In this example, the timer is enabled to start counting from zero. The loop performs operations and periodically reads the current count value, converting it to seconds based on the timer's clock frequency (1 MHz in this case).

7.0.4.2 Reading Timers/Counters by Interrupts 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.

```
// ISR Definition
fn timer_alarm_int_callback() {
    // ISR Code: Handle the timer alarm event
}

// Main Function
```

```rust
fn main() -> ! {
    // Any Startup Code
    // Take Peripherals
    let peripherals = Peripherals::take().unwrap();
    // Configure Timer
    let mut some_timer = TimerDriver::new(peripherals.timer00, &Config::new()).unwrap();
    // Timer Settings
    // 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();
    // Interrupt Setup
    // Attach the ISR to the timer interrupt
    unsafe { some_timer.subscribe(timer_alarm_int_callback).unwrap() }
    // Enable Interrupts
    some_timer.enable_interrupt().unwrap();
    // Enable Timer to Start Counting
    some_timer.enable(true).unwrap();
    // Following Application Code
    loop {
        // Main application logic can continue here
    }
}
```

This example demonstrates setting up a timer to generate an alarm when the count reaches 1000. An ISR (`timer_alarm_int_callback`) is defined to handle the alarm event. The timer is configured, the ISR is subscribed to the timer interrupt, and interrupts are enabled to allow the ISR to be called upon timer events.

Configuring timers on the ESP32-C3 involves initializing the necessary peripherals, setting up timer instances with appropriate configurations, and utilizing control methods to manage timer operations. Timers can operate in various modes, such as one-shot or continuous, and can interact with interrupts to handle overflow or alarm events efficiently. The provided Rust code examples illustrate how to create and configure timers, set alarms, and handle timer events through both polling and interrupt-driven approaches. By leveraging these configurations, developers can implement precise timing mechanisms, event counting, and responsive interrupt handling in their embedded applications.

8 ProgrammingTimers & Counters (no-std)

Configuring timers on ESP devices using Rust involves a structured process that ensures precise time-based operations essential for various embedded applications. This configuration leverages the `esp-hal` crate, which provides abstractions for managing timer peripherals within the ESP-IDF framework. The setup process is divided into several key steps, each building upon the previous to establish a reliable timer mechanism.

8.0.0.1 Initialize the ESP & Gain Access to Peripherals

The first step mirrors the initialization process outlined in previous sections. It involves configuring the ESP device and gaining access to its peripherals using the singleton pattern, which ensures that only one instance of each peripheral is accessed throughout the application. This is achieved with the `esp_hal::init` function, which

initializes the device with default configurations.

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

The `init` function accepts an `esp_hal::Config` struct and returns instances of the peripherals and system clocks, preparing the device for subsequent configurations.

8.0.0.2 Instantiate a Timer Group & Obtain Timer Handle

Once peripherals are initialized, the next step is to create a timer group and obtain a handle for a specific timer within that group. Timer groups allow for the organization and management of multiple timers, facilitating coordinated timing operations.

```
let timer_group0 = TimerGroup::new(peripherals.TIMG0);
```

```
// Instantiate Timer0 in Timer Group 0
let mut timer0 = timer_group0.timer0;
```

By instantiating `timer_group0` and obtaining `timer0`, developers can manage and control Timer 0 within Timer Group 0, setting the foundation for precise timing operations.

8.0.0.3 Configure Analog Pin and Channel

Configuring the timer involves setting up the timer's parameters, such as the start value, compare value, and enabling features like auto-reload. This ensures that the timer operates according to the desired specifications, whether it's for counting, generating interrupts, or triggering alarms.

```
// Start the timer
fn start(&self)

// Stop the timer
fn stop(&self)

// Reset the timer
fn reset(&self)

// Enable auto-reload of load value
fn enable_auto_reload(&self, auto_reload: bool)

// Load a compare value to the timer
fn load_value(&self, value: MicrosDurationU64) -> Result<(), Error>
```

```
// Set Start/Reset Count Value to Zero
timer0.reset();
// Enable Timer to Start Counting
timer0.start();
```

In this example, `timer0` is reset to zero and then started, initiating the counting process. These methods allow for precise control over the timer's behavior, enabling functionalities such as periodic interrupts or timed events.

8.0.0.4 Create an ADC Driver

After configuring the timer, an ADC driver can be created to manage analog-to-digital conversions. This step involves associating the timer with the ADC instance, facilitating synchronized analog measurements based on timer events.

```rust
// 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));

// ISR Definition
#[handler]
fn tg0_t0_level() {
    // Start a Critical Section
    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);
    });
    // Any other ISR Code
}

#[entry]
fn main() -> ! {
    // Take Peripherals
    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;

    // Interrupt Configuration
    // 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);
```

```
    // Following Application Code
    timer0.listen();
    // Move the timer to the global context
    critical_section::with(|cs| {
        G_TIMER.borrow_ref_mut(cs).replace(timer0)
    });
    loop {}
}
```

In this comprehensive example:

1. Global Variable Setup: A global `G_TIMER` variable is defined using a `Mutex` and `RefCell` to safely share the timer instance between the main thread and the Interrupt Service Routine (ISR).
2. ISR Definition: The `tg0_t0_level` function is marked with the `#[handler]` attribute and serves as the ISR. It clears the interrupt flag and re-activates the timer alarm to allow for subsequent interrupts.
3. Main Function:
   - Initialization: Peripherals are initialized, and a timer group is instantiated.
   - Timer Configuration: `timer0` is configured to trigger an interrupt every second by loading a count value equivalent to one second (`1_000_000` microseconds) and enabling the alarm.
   - Interrupt Handler Registration: The ISR is attached to `timer0` to handle timer events.
   - Global Context Assignment: The configured timer is moved to the global context within a critical section to ensure thread-safe access.

8.0.0.5 Reading Timers/Counters by Polling Timers can be read by continuously polling their current value. This method involves checking the timer's count at regular intervals to determine the elapsed time or to trigger specific actions based on the timer's state.

```
// Activate Counter to Start Counting
timer0.start();

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();
    // Print Timer Elapsed Time (from 0)
    println!("Elapsed Timer Duration in Seconds is {}", dur);
}
```

In this example, `timer0` is started and then continuously reset within a loop. The elapsed time since the epoch is calculated and printed, providing real-time feedback on the timer's state. This polling approach ensures that the application can monitor the timer's progress and react accordingly.

9 PWM (std)

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 (Ton) 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.

9.0.1 Duty Cycle and Its Importance

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.

9.0.2 PWM in Embedded Systems

In embedded systems, PWM is widely used for current control tasks. By adjusting the duty cycle, PWM can control the amount of electricity flowing through a load. This is analogous to adjusting a faucet to control water flow—turning it on more frequently increases the flow, while turning it off more often decreases it. PWM achieves this by altering the proportion of time the signal is high (on) versus low (off) within each cycle. This technique is essential for applications like dimming LEDs, regulating motor speeds, positioning servos, and managing the intensity of heating elements.

9.0.3 PWM Generation in ESP32-C3

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:

1. 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.
2. 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.

Pulse Width Modulation is a versatile and essential technique in embedded systems for controlling the average voltage and current delivered to electronic components by varying the duty cycle of square wave signals. In the ESP32-C3 microcontroller, PWM is efficiently handled by dedicated peripherals like the LEDC, which offers multiple channels and timers for generating precise PWM signals. This capability enables developers to implement a variety of applications, including LED dimming, motor speed control, and more, by leveraging the adjustable duty cycle to achieve the desired power levels and performance.

9.0.4 Configuring Pulse Width Modulation (PWM)

Configuring PWM on the ESP32-C3 involves a series of methodical steps to set up the LED PWM Controller (LEDC) peripheral for generating precise PWM signals. This process ensures that PWM

channels are correctly initialized and configured to control various applications such as LED brightness, motor speed, and more.

9.0.4.1 Take the Peripherals The initial step in configuring PWM mirrors the peripheral initialization process used in previous configurations, such as GPIO setup. This involves acquiring and setting up the necessary peripherals required for PWM operation, ensuring that all components are ready for subsequent configuration stages.

9.0.4.2 Configure an LEDC Timer Instance The LEDC peripheral is divided into two main parts: timers and PWM generators/channels. Timers drive the PWM channels and are independently configurable from the PWM generators. To configure an LEDC timer, the `LedcTimerDriver` struct's `new` method is used, which requires two arguments: an LEDC timer peripheral instance (e.g., `timer0`) and a `TimerConfig` configuration instance.

```
// The Timer Configuration Struct
pub struct TimerConfig {
    pub frequency: Hertz,
    pub resolution: Resolution,
    pub speed_mode: SpeedMode,
}
```

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, with the ESP32-C3 offering a single option.

```
// Configure Timer0 with a clock of 50Hz and a resolution of 14 bits
let timer_driver = LedcTimerDriver::new(
    peripherals.ledc.timer0,
    &TimerConfig::default()
        .frequency(50.Hz())
        .resolution(Resolution::Bits14),
)
.unwrap();
```

In this example, `timer0` is configured with a 50Hz clock frequency and a 14-bit resolution. The ESP32-C3 LEDC peripheral includes four timers (`timer0` to `timer3`), each of which can be independently selected based on application requirements.

9.0.4.3 Configure an LEDC PWM Channel Instance After configuring the timer, the next step is to set up the PWM channel. This is done using the `LedcDriver` struct's `new` method, which requires three parameters: a PWM channel instance (e.g., `channel0`), the previously configured timer driver instance, and a GPIO pin instance where the PWM signal will be output.

```
// Creating an LEDC PWM Channel Instance
let mut driver = LedcDriver::new(
    peripherals.ledc.channel0,
    timer_driver,
    peripherals.pins.gpio7,
)
.unwrap();
```

In this example, `channel0` of the LEDC peripheral is associated with `gpio7` for PWM signal output. The `LedcDriver` abstraction handles the necessary pin configurations internally, simplifying the setup process.

9.0.5 Interacting with PWM

9.0.5.1 Controlling the LEDC PWM Peripheral Once the PWM channel is configured, controlling the PWM signal involves setting the desired duty cycle and enabling the PWM output. The duty cycle determines the proportion of time the signal is in the "on" state within each period.

```
// Configuring Duty Cycle and Enabling PWM Output
// Set Desired Duty Cycle
some_ledc_driver_inst.set_duty(1000_u32).unwrap();
// Enable PWM Output
some_ledc_driver_inst.enable().unwrap();
```

Here, the `set_duty` method sets the duty cycle based on a value that corresponds to the PWM resolution. After setting the duty cycle, the `enable` method activates the PWM signal on the configured GPIO pin.

9.0.5.2 Reading from the LEDC PWM Peripheral While generating PWM signals typically doesn't require reading events, it's often useful to read the current duty cycle or determine the maximum possible duty value for dynamic adjustments.

```
// Configuring 20% Duty Cycle and Enabling PWM Output
// Get Maximum Possible Duty Value
let max_duty = some_ledc_driver_inst.get_max_duty();
// Set Duty Cycle to 20%
some_ledc_driver_inst.set_duty(max_duty * 20 / 100).unwrap();
// Enable PWM Output
some_ledc_driver_inst.enable().unwrap();
```

In this example, the `get_max_duty` method retrieves the maximum duty cycle value based on the configured resolution. The duty cycle is then set to 20% by calculating 20% of the maximum duty value, and the PWM output is enabled accordingly.

Configuring PWM on the ESP32-C3 involves initializing the necessary peripherals, setting up LEDC timers, and configuring PWM channels to generate precise PWM signals. By following these steps and utilizing the provided code examples, developers can effectively control various applications such as LED brightness, motor speeds, and more. The LEDC peripheral's flexibility, with multiple timers and channels, allows for simultaneous PWM signal generation tailored to diverse embedded system requirements.

10 PWM (no-std)

10.0.0.1 Initialize the ESP & Gain Access to Peripherals The initial step involves configuring the ESP device and gaining access to its peripherals. This is achieved using the `esp_hal::init` function, which sets up the device clocks and initializes peripheral instances using the singleton pattern. This ensures that only one instance of each peripheral is accessed throughout the application, promoting safe and efficient hardware resource management.

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

The `init` function takes an `esp_hal::Config` struct as an argument and returns instances of the peripherals and system clocks. Using the default configuration simplifies the setup process by applying standard settings.

10.0.0.2 Create an IO Driver

With peripherals initialized, the next step is to create an IO driver, which provides control over individual IO pins. The `esp_hal::gpio` module offers the `Io` struct for this purpose.

```
let io = Io::new(peripherals.GPIO, peripherals.IO_MUX);
```

This code initializes the IO driver by passing the GPIO and IO_MUX peripherals obtained during initialization. The IO driver facilitates further configuration and management of specific GPIO pins.

10.0.0.3 Configure the PWM Pin into Output

Before using a pin for PWM output, it must be configured as a push-pull output. This configuration allows the microcontroller to actively drive the pin high or low, enabling precise control over connected devices.

```
let some_output_pin = Output::new(io.pins.gpio3, Level::Low);
```

In this example, GPIO3 is set as an output with an initial low level. The push-pull configuration ensures that the pin can actively drive both high and low states, which is essential for generating PWM signals.

10.0.0.4 Create an LEDC Peripheral Driver

The LED Controller (LEDC) peripheral manages PWM signal generation. Creating an LEDC driver involves instantiating the `Ledc` struct and setting the global clock source.

```
pub fn new(
    _instance: impl Peripheral<P = LEDC> + 'd
) -> Self
```

```
let mut ledc = Ledc::new(peripherals.LEDC);
```

```
ledc.set_global_slow_clock(LSGlobalClkSource::APBClk);
```

This code initializes the LEDC driver with the LEDC peripheral and sets the global slow clock source to `APBClk`, which is necessary for timing accuracy in PWM signal generation.

10.0.0.5 Configure the LEDC Timer

Timers are integral to PWM signal generation, determining the frequency and resolution of the PWM signal. Configuring a timer involves associating it with the LEDC instance and setting its parameters such as duty resolution and frequency.

```
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();
```

In this example, Timer0 is configured with a 12-bit duty resolution and a frequency of 4 kHz. The `configure` method sets these parameters, ensuring that the PWM signal operates at the desired specifications.

10.0.0.6 Configure a PWM Channel Instance

After configuring the timer, a PWM channel must be set up to generate the PWM signal on a specific pin. This involves associating the output pin with the PWM channel, linking it to the configured timer, and setting the duty cycle.

```
let mut channel =
    ledc.get_channel(channel::Number::Channel0, some_output_pin);
```

```
pub struct Config<'a, S>
where
    S: TimerSpeed,
{
    pub timer: &'a dyn TimerIFace<S>,
    pub duty_pct: u8,
    pub pin_config: PinConfig,
}
```

```
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();
```

This configuration associates Channel0 with the previously configured Timer0 and sets the duty cycle to 10%. The `PinConfig::PushPull` ensures that the pin can actively drive the PWM signal.

10.0.1 Interacting with PWM

Once PWM is configured, it can be controlled and monitored through various methods provided by the LEDC driver.

10.0.1.1 Controlling the LEDC PWM Peripheral PWM signals are primarily controlled by adjusting the duty cycle, which determines the proportion of time the signal stays high versus low within each cycle. This is managed using the `set_duty` method.

```
// Set the Desired Duty Cycle
fn set_duty(&self, duty_pct: u8) -> Result<(), Error>
```

This method allows the application to dynamically adjust the duty cycle, enabling effects such as LED fading by smoothly transitioning between different brightness levels.

10.0.1.2 Reading from the LEDC PWM Peripheral While PWM generation typically does not require reading from the peripheral, certain applications may benefit from monitoring the current state. The `max_duty_cycle` method can be used to retrieve the maximum duty cycle value supported by the configuration.

11 Serial Communication (std)

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.

11.0.1 UART (Universal Asynchronous Receiver/Transmitter)

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 `println!()` 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.

11.0.2 UART Configuration Parameters

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.

11.0.3 Asynchronous vs. Synchronous Serial Communication

Serial communication can be categorized into asynchronous and synchronous modes:

1. 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.

2. 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.

11.0.4 Common Serial Communication Protocols in Embedded Systems

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.

Serial communication is an essential method for data transmission in embedded systems, offering various protocols tailored to different application needs. UART, an asynchronous protocol, is widely used for its simplicity and reliability in point-to-point communication, particularly for debugging and interfacing with serial devices. Ensuring that both transmitting and receiving devices share matching configurations—such as baud rate, data bits, parity, and stop bits—is crucial for successful UART communication. Additionally, understanding the differences between asynchronous and synchronous serial communication helps in selecting the appropriate protocol for specific embedded applications, whether it's the straightforward UART for simple tasks or the high-speed SPI for more demanding data transfer requirements.

11.0.5 Configuring UART

11.0.5.1 Take the Peripherals The initial step in configuring UART is identical to previous peripheral setups. This involves acquiring and initializing the necessary peripherals required for UART operation to ensure they are ready for configuration.

11.0.5.2 Configure a UART Instance To configure UART on the ESP32-C3, the `UartDriver` abstraction from the `esp-idf-hal` library is utilized. The `new` method of `UartDriver` is responsible for creating a UART instance and requires six parameters:

1. uart: An instance of a UART peripheral.
2. tx: An output pin for transmitting serial bits.
3. rx: An input pin for receiving serial bits.
4. cts: (Optional) A pin for Clear To Send control flow.
5. rts: (Optional) A pin for Request To Send control flow.
6. config: A reference to a UART configuration.

UartDriver `new` Method Signature

```
pub fn new<UART: Uart>(
    uart: impl Peripheral<P = UART> + 'd,
    tx: impl Peripheral<P = impl OutputPin> + 'd,
```

```
    rx: impl Peripheral<P = impl InputPin> + 'd,
    cts: Option<impl Peripheral<P = impl InputPin> + 'd>,
    rts: Option<impl Peripheral<P = impl OutputPin> + 'd>,
    config: &Config
) -> Result<Self, EspError>
```

The `Config` struct for UART resides in the `uart::config` module and includes several members to fine-tune UART settings. While not all members need to be explicitly configured, commonly used configurations like baud rate can be set using the provided methods.

The `uart::config::Config` Configuration Struct

```
pub struct Config {
    pub baudrate: Hertz,
    pub data_bits: DataBits,
    pub parity: Parity,
    pub stop_bits: StopBits,
    pub flow_control: FlowControl,
    pub flow_control_rts_threshold: u8,
    pub source_clock: SourceClock,
    pub intr_flags: EnumSet<InterruptType>,
    pub event_config: EventConfig,
    pub rx_fifo_size: usize,
    pub tx_fifo_size: usize,
    pub queue_size: usize,
    /* private fields */
}
```

Instantiating and Configuring `uart1` as 8N1 with 115200 Hz Baud and No Flow Control

```
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();
```

In this example, `uart1` is instantiated with an 8N1 configuration (8 data bits, no parity, 1 stop bit) and a baud rate of 115200 Hz. Hardware flow control is disabled by setting `cts` and `rts` to `None`. The `turbofish` syntax (`::<gpio::Gpio0>`) assists the compiler in inferring the correct types for the optional flow control pins.

11.0.6 Interacting with UART

11.0.6.1 Writing to the UART Peripheral To send data over UART, the `write` method of the `UartDriver` is used. This method takes a slice of `u8` data and transmits each byte sequentially.

Signature of `UartDriver` `write` Method

```rust
pub fn write(&self, bytes: &[u8]) -> Result<usize, EspError>
```

Example of Sending a Single Byte Over UART

```rust
some_uart_instance.write(&[25_u8]).unwrap();
```

This example sends a single byte with the value 25 over the configured UART channel. The `write` method handles the conversion of parallel data to serial format and manages the transmission of bits.

11.0.6.2 Blocking Read from the UART Peripheral Receiving data over UART involves using the `read` method, which performs a blocking read operation. This means the code will wait (block) until the specified number of bytes are received or the timeout is reached.

Signature of `UartDriver` `read` Method

```rust
pub fn read(
    &self,
    buf: &mut [u8],
    timeout: TickType_t
) -> Result<usize, EspError>
```

Example of Receiving a Single Byte Over UART

```rust
let mut buf = [0_u8; 1];
// BLOCK is a constant containing the largest possible u32 value
some_uart_instance.read(&mut buf, BLOCK).unwrap();
```

In this example, the `read` method attempts to receive a single byte and stores it in `buf`. The `BLOCK` constant ensures that the method waits indefinitely until data is received.

11.0.7 Configuring I2C

11.0.7.1 Take the Peripherals Similar to UART configuration, the first step in setting up I2C involves taking and initializing the necessary peripherals as demonstrated before.

11.0.7.2 Create and Configure an I2C Instance The `I2cDriver` abstraction from the `esp-idf-hal` library is used to create and configure an I2C instance. The `new` method requires four parameters:

1. i2c: An instance of an I2C peripheral.
2. sda: A bidirectional pin instance for the Serial Data Line.
3. scl: A bidirectional pin instance for the Serial Clock Line.
4. config: A reference to an I2C configuration.

I2cDriver `new` Method Signature

```rust
pub fn new<I2C: I2c>(
    _i2c: impl Peripheral<P = I2C> + 'd,
    sda: impl Peripheral<P = impl InputPin + OutputPin> + 'd,
    scl: impl Peripheral<P = impl InputPin + OutputPin> + 'd,
    config: &Config
) -> Result<Self, EspError>
```

The `Config` struct for I2C is defined in the `i2c::config` module and includes various parameters to tailor the I2C communication settings.

The `i2c::config::Config` Configuration Struct

```
pub struct Config {
    pub baudrate: Hertz,
    pub sda_pullup_enabled: bool,
    pub scl_pullup_enabled: bool,
    pub timeout: Option<APBTickType>,
    pub intr_flags: EnumSet<InterruptType>,
}
```

*Example of Instantiating and Configuring an I2C Channel Using the `i2c0` Peripheral

```
let i2c_instance = peripherals.i2c0;
let config = I2cConfig::new().baudrate(100.kHz().into());
let i2c_driver = I2cDriver::new(i2c, sda, scl, &config).unwrap();
```

This example configures the `i2c0` peripheral with a baud rate of 100 kHz. The `I2cDriver::new` method initializes the I2C instance with the specified SDA and SCL pins and the provided configuration.

11.0.8 Interacting with I2C

11.0.8.1 Writing to the I2C Peripheral To send data over I2C, the `write` method of the `I2cDriver` is utilized. This method requires the slave address, a slice of `u8` data to send, and a timeout value.

Signature of `I2cDriver` `write` Method

```
pub fn write(
    &mut self,
    addr: u8,
    bytes: &[u8],
    timeout: TickType_t
) -> Result<(), EspError>
```

Example of Sending a Single Byte Over I2C to Address 0x65

```
// BLOCK is a constant containing the largest possible u32 value
some_i2c_instance.write(0x65, &[25], BLOCK).unwrap();
```

In this example, a single byte with the value `25` is sent to the I2C slave device with the address `0x65`. The `BLOCK` constant ensures that the method waits until the write operation is completed.

11.0.8.2 Reading from the I2C Peripheral Receiving data over I2C involves using the `read` method, which retrieves data from a specified slave address into a provided buffer within a given timeout period.

Signature of `I2cDriver` `read` Method

```
pub fn read(
    &mut self,
    addr: u8,
    buffer: &mut [u8],
    timeout: TickType_t
) -> Result<(), EspError>
```

Example of Receiving a Single Byte Over I2C from Address 0x65

```
let mut buf = [0_u8; 1];
// BLOCK is a constant containing the largest possible u32 value
some_i2c_instance.read(0x65, &mut buf, BLOCK).unwrap();
```

This example demonstrates how to receive a single byte from the I2C slave device at address `0x65`. The received byte is stored in the `buf` array, and the operation blocks until the data is received or the timeout is reached.

Configuring UART and I2C on the ESP32-C3 involves initializing the necessary peripherals, creating and configuring driver instances with appropriate settings, and utilizing provided methods to handle data transmission and reception. For UART, this includes setting parameters like baud rate, data bits, parity, and stop bits, and managing transmit and receive operations using the `write` and `read` methods. For I2C, configuration involves setting the baud rate, enabling pull-ups, and handling communication with slave devices through `write` and `read` methods that require specifying slave addresses and managing timeouts. By following the provided steps and utilizing the code examples, developers can effectively implement robust serial communication in their embedded applications.

12 Serial Communication (no-std)

12.0.0.1 Initialize the ESP & Gain Access to Peripherals Before utilizing any device peripherals, it's crucial to configure the ESP device itself. This involves setting up the device clocks and gaining access to peripheral instances using the singleton pattern. The `esp-hal` crate provides a streamlined method for initializing the device with default configurations.

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

The `init` function takes an `esp_hal::Config` struct as an argument and returns instances of the peripherals and system clocks. Using the default configuration simplifies the setup process by applying standard settings, ensuring that the device is ready for peripheral configuration.

```
pub struct Config {
    pub cpu_clock: CpuClock,
    pub watchdog: WatchdogConfig,
}
```

The `Config` struct allows customization of system parameters such as CPU clock speed and watchdog settings. However, for basic applications, the default values are typically sufficient, as demonstrated in the initialization step.

12.0.0.2 Create an IO Driver With peripherals initialized, the next step is to create an IO driver. The IO driver provides control over individual IO pins, enabling their configuration for various functions, including UART communication.

```
let io = Io::new(peripherals.GPIO, peripherals.IO_MUX);
```

This code initializes the IO driver by passing the GPIO and IO_MUX peripherals obtained during initialization. The IO driver facilitates further configuration and management of specific GPIO pins required for UART operations.

12.0.0.3 Instantiate UART Pins Before creating a UART instance, the pins designated for UART communication must be instantiated. Typically, the transmit (TX) pin is configured as an output, and the receive (RX) pin is configured as an input. This configuration ensures proper data transmission and reception.

```
// Configure GPIO21 as UART TX (Output)
let uart_tx = Output::new(io.pins.gpio21, Level::Low);

// Configure GPIO20 as UART RX (Input with Pull-Up)
let uart_rx = Input::new(io.pins.gpio20, Pull::Up);
```

In this example: - GPIO21 is configured as an output pin for transmitting data. - GPIO20 is configured as an input pin with a pull-up resistor for receiving data.

12.0.0.4 Configure a UART Instance Configuring UART involves creating a UART instance with specific settings such as baud rate, data bits, parity, stop bits, and clock source. The `esp_hal::uart::Uart` abstraction provides methods to instantiate and configure UART peripherals effectively.

```
pub fn new_with_config<TX: OutputPin, RX: InputPin>(
    uart: impl Peripheral<P = T> + 'd,
    config: Config,
    tx: impl Peripheral<P = TX> + 'd,
    rx: impl Peripheral<P = RX> + 'd,
) -> Result<Self, Error>
```

```
pub struct Config {
    pub baudrate: u32,
    pub data_bits: DataBits,
    pub parity: Parity,
    pub stop_bits: StopBits,
    pub clock_source: ClockSource,
    pub rx_fifo_full_threshold: u16,
    pub rx_timeout: Option<u8>,
}
```

```
// Create a UART Configuration
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 log = Uart::new_with_config(
    peripherals.UART0,
    uart_config,
    io.pins.gpio21,
    io.pins.gpio20,
)
.unwrap();
```

In this example: 1. UART Configuration: An instance of `Config` is created with the following settings: - Baud Rate: 115200 - Data Bits: 8 - Parity: None - Stop Bits: 1 - Clock Source: APB Clock 2. UART Instantiation: A new UART instance (`log`) is created using the `UART0` peripheral, the defined configuration, and the instantiated TX (`gpio21`) and RX (`gpio20`) pins. The `unwrap()` method is

used to handle any potential errors during instantiation, assuming successful configuration.

Note: `UART0` is typically used for logging and firmware communication on ESP32-C3 development boards. For UART operations intended for other purposes, it's advisable to consult the device's reference manual to ensure correct peripheral usage and pin assignments.

12.0.0.5 Interacting with UART Once the UART instance is configured, it can be used to send and receive data. The `Uart` type offers several methods to facilitate standard write and read operations.

12.0.0.5.1 Writing to the UART Peripheral Sending data over UART is accomplished through write operations. The `write_bytes` method allows sending a slice of bytes over the UART channel.

```
pub fn write_bytes(&mut self, data: &[u8]) -> Result<usize, Error>
```

```
some_uart_instance.write_bytes(&[25_u8]).unwrap();
```

In this example, a single byte with the value `25` is sent over the UART channel. The `unwrap()` method ensures that the operation succeeds, and any errors during the write process will cause a panic. For robust applications, consider handling errors gracefully instead of using `unwrap()`.

12.0.0.5.2 Blocking Read from the UART Peripheral Receiving data over UART involves read operations. The blocking read approach waits until data is available before proceeding, ensuring that the application receives the intended data.

```
fn read(&mut self, buf: &mut [u8]) -> Result<usize, Self::Error>
```

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

In this example: 1. Buffer Initialization: A mutable buffer `buf` is created to store the received byte. 2. Read Operation: The `read` method is called on the UART instance, attempting to read one byte into the buffer. The `unwrap()` method is used to handle any potential errors, assuming successful data reception.

Note: The blocking read approach may halt the execution of other tasks until data is received. For applications requiring concurrent operations, consider implementing non-blocking reads or utilizing asynchronous programming techniques.

12.0.1 Configuring I2C on ESP Devices

Configuring the Inter-Integrated Circuit (I2C) interface on ESP devices using Rust involves a systematic process that ensures reliable serial communication with various I2C peripherals. I2C is widely used for connecting low-speed peripherals like sensors, displays, and EEPROMs to microcontrollers. This guide outlines the necessary steps to initialize the ESP device, set up the IO driver, configure I2C pins, instantiate and configure an I2C instance, and interact with the I2C peripheral for data transmission and reception.

12.0.1.1 Initialize the ESP & Gain Access to Peripherals The initial step involves configuring the ESP device and gaining access to its peripherals. This is achieved using the `esp_hal::init` function, which sets up the device clocks and initializes peripheral instances using the singleton pattern. This ensures that only one instance of each peripheral is accessed throughout the application, promoting safe and efficient hardware resource management.

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

The `init` function takes an `esp_hal::Config` struct as an argument and returns instances of the peripherals and system clocks. Using the default configuration simplifies the setup process by applying standard settings, ensuring that the device is ready for peripheral configuration.

12.0.1.2 Create an IO Driver With peripherals initialized, the next step is to create an IO driver. The IO driver provides control over individual IO pins, enabling their configuration for various functions, including I2C communication.

```
let io = Io::new(peripherals.GPIO, peripherals.IO_MUX);
```

This code initializes the IO driver by passing the GPIO and IO_MUX peripherals obtained during initialization. The IO driver facilitates further configuration and management of specific GPIO pins required for I2C operations.

12.0.1.3 Create and Configure an I2C Instance Configuring I2C involves creating an I2C instance with specific settings such as the operating frequency and associating it with designated SDA (Serial Data) and SCL (Serial Clock) pins. The `esp_hal::i2c::I2c` abstraction provides methods to instantiate and configure I2C peripherals effectively.

```
pub fn new<SDA: OutputPin + InputPin, SCL: OutputPin + InputPin>(
    i2c: impl Peripheral<P = T> + 'd,
    sda: impl Peripheral<P = SDA> + 'd,
    scl: impl Peripheral<P = SCL> + 'd,
    frequency: HertzU32,
) -> Self
```

```
let i2c = I2c::new(
    peripherals.I2C0,
    io.pins.gpio1,
    io.pins.gpio2,
    100u32.kHz(),
);
```

In this example: 1. I2C Peripheral Instance: `peripherals.I2C0` refers to the I2C0 peripheral instance obtained during initialization. 2. SDA and SCL Pins: `io.pins.gpio1` and `io.pins.gpio2` are instantiated as bidirectional pins for SDA and SCL respectively. 3. Frequency: The I2C operation frequency is set to 100 kHz, which is a standard speed for I2C communication.

12.0.1.4 Instantiate UART Pins Before creating a UART instance, the pins designated for UART communication must be instantiated. Typically, the transmit (TX) pin is configured as an output, and the receive (RX) pin is configured as an input. This configuration ensures proper data transmission and reception.

```
// Configure GPIO21 as UART TX (Output)
let uart_tx = Output::new(io.pins.gpio21, Level::Low);

// Configure GPIO20 as UART RX (Input with Pull-Up)
let uart_rx = Input::new(io.pins.gpio20, Pull::Up);
```

In this example: - GPIO21 is configured as an output pin for transmitting data. - GPIO20 is configured as an input pin with a pull-up resistor for receiving data.

12.0.1.5 Configure an I2C Instance

Configuring I2C involves creating an I2C instance with specific settings such as the operating frequency and associating it with designated SDA (Serial Data) and SCL (Serial Clock) pins.

```
pub fn new<SDA: OutputPin + InputPin, SCL: OutputPin + InputPin>(
    i2c: impl Peripheral<P = T> + 'd,
    sda: impl Peripheral<P = SDA> + 'd,
    scl: impl Peripheral<P = SCL> + 'd,
    frequency: HertzU32,
) -> Self
```

```
let i2c = I2c::new(
    peripherals.I2C0,
    io.pins.gpio1,
    io.pins.gpio2,
    100u32.kHz(),
);
```

In this example: 1. I2C Peripheral Instance: `peripherals.I2C0` refers to the I2C0 peripheral instance obtained during initialization. 2. SDA and SCL Pins: `io.pins.gpio1` and `io.pins.gpio2` are instantiated as bidirectional pins for SDA and SCL respectively. 3. Frequency: The I2C operation frequency is set to 100 kHz, which is a standard speed for I2C communication.

12.0.1.6 Interacting with I2C

Once the I2C instance is configured, it can be used to send and receive data to and from I2C devices. The `I2c` type implements the `embedded_io::Write` and `embedded_io::Read` traits, facilitating standard write and read operations.

12.0.1.6.1 Writing to the I2C Peripheral

Sending data over I2C is accomplished through write operations. The `write` method allows sending a slice of bytes to a specific I2C slave address.

```
pub fn write(&mut self, addr: u8, bytes: &[u8]) -> Result<(), Error>
```

```
some_i2c_instance.write(0x65, &[25]).unwrap();
```

In this example, a single byte with the value `25` is sent to the I2C slave device with the address `0x65`. The `unwrap()` method is used to handle any potential errors during the write process, assuming successful transmission.

12.0.1.6.2 Reading from the I2C Peripheral

Receiving data over I2C involves read operations. The `read` method allows reading a specified number of bytes from a particular I2C slave address into a buffer.

```
pub fn read(&mut self, address: u8, buffer: &mut [u8]) -> Result<(), Error>
```

```
let mut buf = [0_u8; 1];
some_i2c_instance.read(0x65, &mut buf).unwrap();
```

In this example: 1. Buffer Initialization: A mutable buffer `buf` is created to store the received byte. 2. Read Operation: The `read` method is called on the I2C instance, attempting to read one byte from the I2C slave device at address `0x65` into the buffer. The `unwrap()` method is used to handle any potential errors, assuming successful data reception.

13 IoT & Networking Services (std)

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.

13.0.1 WiFi

WiFi, short for Wireless Fidelity, enables wireless connections between devices within a local network using radio waves. Devices can function as either clients or access points, with access points acting as central hubs connected to wired networks to extend wireless connectivity. WiFi operates primarily in the 2.4 GHz and 5 GHz frequency bands. The 2.4 GHz band offers wider coverage and better obstacle penetration but is more prone to congestion. In contrast, the 5 GHz band provides faster data rates and more reliable connections in crowded environments, albeit with a shorter range and reduced penetration capabilities. Security in WiFi networks is maintained through encryption protocols like WPA2 and WPA3, ensuring data privacy and preventing unauthorized access. Once connected, devices can communicate seamlessly and access network resources without the limitations of physical cables. The ESP32-C3 microcontroller is equipped with advanced WiFi functionalities, offering four virtual interfaces: Station (STA), Access Point (AP), Sniffer, and a reserved interface. It supports multiple operational modes, including Station-only, AP-only, and coexistence modes, adhering to IEEE 802.11 b/g/n standards. Security features include support for WPA2 and WPA3 protocols. Additionally, the ESP32-C3 uniquely incorporates the ESP-NOW protocol and a Long Range mode, enabling extended data transmission distances of up to 1 kilometer. These capabilities make the ESP32-C3 a versatile choice for a wide range of IoT applications requiring reliable and secure wireless communication.

13.0.2 Configuring WiFi

This section outlines the process of configuring WiFi for ESP devices using Rust. It emphasizes the importance of establishing network access as a foundational step for any IoT service, with WiFi being a primary protocol for connectivity. The focus is on creating a basic WiFi connection without delving into complex configuration details to maintain simplicity and avoid excessive code complexity. The section also highlights the use of the `esp-idf-svc` and `embedded-svc` crates for managing various networking services.

13.0.3 The AnyHow Crate

The AnyHow crate is introduced as a solution to improve error handling in Rust applications for ESP devices. Traditionally, the `unwrap` method was used to extract values from `Result` types, which can lead to panics if an error occurs. The AnyHow crate integrates ESP-IDF error codes, providing more informative error messages and better context for debugging. To use AnyHow, developers need to declare it as a dependency, import it, change the main function's return type to `anyhow::Result`, and replace `unwrap` calls with the `?` operator. This approach enhances the robustness and maintainability of the code, especially in wireless implementations.

13.0.4 Take the Peripherals

The first step in configuring WiFi involves taking control of the device's peripherals. This is done using the `Peripherals::take().unwrap()` method, which initializes the necessary hardware components

required for WiFi functionality. This step is consistent with earlier sections of the documentation, ensuring that the peripherals are properly initialized before proceeding to create and configure the WiFi driver instance.

13.0.5 Create a WiFi Driver Instance

Creating a WiFi driver instance involves using the `esp-idf-svc` crate, which offers multiple structures such as `EspWifi` and `WifiDriver`. The `EspWifi` struct provides a higher-level abstraction, simplifying the creation of networking examples by encapsulating a `WifiDriver`. To instantiate `EspWifi`, developers use the `new` method, which requires a WiFi peripheral instance, a system event loop, and an optional non-volatile storage (NVS) partition. This setup follows a singleton pattern, ensuring that only one instance of each component is created.

13.0.6 Configure the WiFi Driver Instance

Once the WiFi driver instance is created, it needs to be configured to operate either in station mode or access point mode. Station mode allows the device to connect to an existing WiFi network as a client, while access point mode enables the device to act as a hotspot for other clients to connect. The `set_configuration` method of `EspWifi` is used to apply the desired configuration by passing a `Configuration` enum. For example, configuring the device as a client involves specifying the SSID, password, and authentication method. This step is crucial for establishing the desired network behavior of the ESP device.

13.0.7 Interacting with WiFi

Interacting with the WiFi subsystem involves managing connections and monitoring the status of the network interface. The ESP-IDF framework supports both blocking and non-blocking operations, with the example focusing on a blocking approach for simplicity. By wrapping the `EspWifi` instance with a `BlockingWifi` abstraction, developers can perform operations such as starting the WiFi, connecting to a network, and waiting for the network interface to become active. Additionally, the framework provides methods like `is_connected` and `get_configuration` to check the connection status and retrieve current network settings, respectively.

13.0.8 Connecting to WiFi

Connecting to a WiFi network involves a sequence of method calls on the `BlockingWifi` instance. First, the `start` method initializes the WiFi peripheral. Next, the `connect` method attempts to establish a connection to the specified network using the previously set configuration. Finally, the `wait_netif_up` method blocks the execution until the network interface is fully up and running. Each of these methods returns a `Result`, ensuring that any issues during the connection process are appropriately handled and propagated.

13.0.9 Reading WiFi Status

Monitoring the WiFi status is essential for ensuring a stable and reliable network connection. The `EspWifi` struct provides methods such as `is_connected` to check if the device is currently connected to a WiFi network and `get_configuration` to retrieve the current network settings. These methods return results that can be used to verify the connection status and debug any issues related to network configuration. While `EspWifi` offers a wide range of methods for controlling and monitoring the WiFi instance, this section focuses on the essential functions necessary for achieving and maintaining network connectivity.

13.0.10 HTTP Client

HTTP (Hypertext Transfer Protocol) is the cornerstone of internet communication, operating on a client-server model that enables data exchange between web browsers and servers. When a user enters a website URL, the browser initiates an HTTP request—commonly a GET request—to the server hosting the desired website. This request specifies actions such as fetching a webpage or submitting data and utilizes methods like GET, POST, PUT, and DELETE to define its purpose. HTTP relies on the TCP protocol for data transmission and includes headers that provide additional information about the request and response. Upon receiving a request, the server processes it and responds with a status code (e.g., 200 for success, 404 for not found) and the requested data. HTTP is inherently stateless, treating each request independently, which can make data vulnerable to interception. To enhance security, HTTPS encrypts the data transmitted between clients and servers, ensuring secure communication. A typical GET request workflow involves establishing a TCP connection, sending the GET request with necessary headers, the server processing and responding with the appropriate status code and data, the client rendering the response, and finally closing the connection. Understanding HTTP client interactions is essential for developing robust and secure IoT applications that communicate effectively over the internet.

13.0.11 Configuring an HTTP Client

Configuring an HTTP client for ESP devices using Rust involves a systematic approach that ensures secure and efficient communication over the internet. This process leverages the `esp-idf-svc` crate, which provides essential abstractions for handling HTTP client functionalities within the ESP-IDF framework. The configuration process is divided into several key steps, each building upon the previous to establish a robust HTTP client setup.

13.0.11.1 Take the Peripherals The initial step in configuring an HTTP client mirrors the process used in setting up WiFi. It involves taking control of the device's peripherals using the `Peripherals::take().unwrap()` method. This ensures that all necessary hardware components are properly initialized and ready for subsequent configuration steps. By securing access to the peripherals, the device can effectively manage networking tasks required for HTTP communication.

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

13.0.11.2 Connect to WiFi Before establishing an HTTP connection, the device must be connected to a WiFi network. This step is identical to the WiFi configuration process outlined earlier, where the device is set up either as a station or an access point. Successfully connecting to WiFi provides the necessary network access for the HTTP client to send and receive data over the internet.

```
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()?;
```

13.0.11.3 Configure an HTTP Connection

Configuring an HTTP connection involves creating an instance of `EspHttpConnection` from the `esp_idf_svc::http::client` module. This abstraction requires a reference to a `http::client::Configuration` struct during instantiation. The configuration typically sets essential parameters such as `use_global_ca_store` and `crt_bundle_attach` to enable secure HTTPS connections. These settings ensure that the HTTP client uses a global certificate authority store and attaches the necessary certificate bundle for encryption, which is crucial for secure data transmission.

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

13.0.12 Interacting with HTTP

Once the HTTP client is configured, interacting with it involves initiating requests, handling responses, and processing the received data. The `EspHttpConnection` abstraction provides methods to manage these interactions seamlessly.

13.0.12.1 Initiating Requests & Responses

To initiate an HTTP request, the `initiate_request` method of `EspHttpConnection` is used. This method requires specifying the request type (e.g., GET, POST), the target URL, and any headers that need to accompany the request. For example, initiating a GET request to "https://httpbin.org/get" can be done as follows:

```
let url = "https://httpbin.org/get";
let _request = httpconnection.initiate_request(Method::Get, url, &[])?;
```

After sending the request, the `initiate_response` method is called to handle the incoming response from the server. This structured approach ensures that requests and responses are managed effectively.

```
httpconnection.initiate_response()?;
```

13.0.12.2 Processing HTTP Responses

Handling the server's response involves retrieving and interpreting various components of the HTTP response. The `EspHttpConnection` provides several methods for this purpose:

- `status`: Retrieves the HTTP status code (e.g., 200 for success).
- `status_message`: Provides the corresponding status message.
- `header`: Allows retrieval of specific headers from the response.

For instance, to check if the request was successful and to retrieve a specific header:

```
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
}
```

These methods return results that help verify the success of the request and access any additional information provided by the server. While comprehensive response handling, such as reading and parsing the response body, requires additional code, these methods offer essential tools for managing basic HTTP interactions and ensuring reliable network communication.

13.0.13 HTTP Server Overview

An HTTP server is a fundamental component in web architecture, responsible for handling client requests and serving the appropriate resources. Unlike client-side interactions where the client initiates communication, the server operates continuously, listening for incoming connections and responding to requests such as fetching HTML pages or other resources. When a client, like a web browser, requests a webpage, the server processes the request, retrieves the necessary resources, and sends back an HTTP response containing the requested data and a status code indicating the result of the request (e.g., `200 OK` for success or `404 Not Found` if the resource is unavailable). Understanding the server-side request handling process is essential for developing robust web applications and IoT solutions that require reliable communication between devices and servers.

13.0.14 Configuring an HTTP Server

Configuring an HTTP server on ESP devices using Rust involves several key steps to ensure that the server can handle client requests effectively. This process leverages the `esp-idf-svc` crate, which provides abstractions for managing HTTP server functionalities within the ESP-IDF framework. The configuration process is methodical, starting from initializing peripherals to defining response behaviors for different HTTP methods and endpoints.

13.0.14.1 Take the Peripherals The first step in setting up an HTTP server is to take control of the device's peripherals. This is achieved using the `Peripherals::take().unwrap()` method, which initializes the necessary hardware components required for networking and server operations. Ensuring that peripherals are properly initialized is crucial for the subsequent configuration steps and for the reliable functioning of the HTTP server.

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

13.0.14.2 Connect to WiFi Before the HTTP server can handle incoming requests, the device must be connected to a WiFi network. This step is identical to the WiFi configuration process previously outlined, where the device is set up either as a station or an access point. Successfully connecting to WiFi provides the server with the necessary network access to listen for and respond to client requests.

```
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()?;
```

13.0.14.3 Create and Configure an HTTP Server Instance Creating and configuring an HTTP server involves instantiating the `EspHttpServer` abstraction from the `esp_idf_svc::http::server` module. This is done using the `new` method, which takes a reference to a `http::server::Configuration` struct. The default configuration is typically sufficient for

basic server operations, but it can be customized as needed based on specific networking or protocol requirements.

```
// Create Server Connection Handle
let httpserver = EspHttpServer::new(&Configuration::default())?;
```

13.0.15 Interacting with an HTTP Server

Once the HTTP server is configured and running, it needs to handle incoming requests and respond appropriately. This involves defining response behaviors for different HTTP methods and endpoints using the `fn_handler` method provided by the `EspHttpServer` abstraction.

13.0.15.1 Defining Response Behavior The `fn_handler` method allows developers to register handler functions that define how the server should respond to specific requests. Each handler is associated with a particular URL and HTTP method. For example, to handle GET requests to the root URL (`"/"`), a handler can be defined as follows:

```
// Define Server RequestHandler Behavior for GET on Root URL
httpserver.fn_handler("/", Method::Get, |request| {
    // Retrieve HTML String
    let html = index_html();

    // Respond with OK status
    let mut response = request.into_ok_response()?;

    // Return Requested Object (IndexPage)
    response.write(html.as_bytes())?;

    Ok::<(), anyhow::Error>(())
})?;
```

In this example:

1. Define Handler: The handler is registered for the root URL (`"/"`) and the GET method.
2. Retrieve Content: The `index_html()` function generates or retrieves the HTML content to be served.
3. Create Response: The `into_ok_response()` method creates an HTTP response with a `200 OK` status.
4. Write Content: The HTML content is written to the response body.
5. Finalize: The handler completes successfully, allowing the server to send the response to the client.

13.0.16 SNTP Overview

SNTP, or Simple Network Time Protocol, is a protocol designed to synchronize the clocks of devices over a network. It ensures that devices such as computers, routers, and servers maintain accurate and consistent time references by communicating with dedicated time servers. Unlike its more complex counterpart, NTP (Network Time Protocol), SNTP offers a lightweight solution suitable for most basic time synchronization needs where high precision is not critical.

13.0.17 How SNTP Works

SNTP operates through a straightforward four-step process to maintain accurate time across devices:

1. Requesting the Time: A device, such as an ESP microcontroller, sends a request to a network time server to obtain the current time.
2. Time Server Responds: The time server receives the request and replies with the current time, typically precise to the millisecond.
3. Adjusting the Clock: The device receives the time data from the server and adjusts its internal clock to match the received time, ensuring consistency across the network.
4. Regular Updates: To maintain accuracy, SNTP can be configured to periodically send time synchronization requests at regular intervals, allowing devices to correct any drift in their internal clocks.

13.0.18 Key Features of SNTP

- Simplicity: SNTP is a simplified version of NTP, making it easier to implement for devices that do not require the advanced features and high precision offered by NTP.
- Lightweight Protocol: Operating over UDP (User Datagram Protocol), SNTP benefits from a connectionless and low-overhead communication method, which is ideal for sending small packets of data with minimal delay.
- Reliability: By relying on Coordinated Universal Time (UTC), SNTP provides a standardized and accurate time reference that ensures all synchronized devices operate on the same time basis.

13.0.19 Why Use SNTP

SNTP is particularly advantageous in scenarios where:

- Resource Constraints: Devices with limited processing power and memory, such as microcontrollers in IoT applications, can efficiently handle SNTP without the complexity of full NTP implementations.
- Basic Synchronization Needs: Applications that require consistent timekeeping without the necessity for millisecond-level precision benefit from the simplicity and efficiency of SNTP.
- Network Efficiency: The use of UDP allows for faster communication with lower overhead, making SNTP suitable for networks where bandwidth and latency are critical factors.

Coordinated Universal Time (UTC) serves as the global time standard for SNTP. By synchronizing to UTC, SNTP ensures that all devices across different regions and networks adhere to a uniform time reference. This uniformity is essential for applications that rely on accurate timestamps, logging, and coordinated operations across multiple devices.

13.0.20 Configuring an SNTP Instance

Configuring SNTP on ESP devices using Rust involves creating and setting up an SNTP client instance. This process leverages the `esp-idf-svc` crate, which provides the `EspSntp` abstraction for managing SNTP functionalities within the ESP-IDF framework.

13.0.20.1 Take the Peripherals The initial step involves taking control of the device's peripherals to ensure that all necessary hardware components are initialized and ready for network operations.

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

13.0.20.2 Connect to WiFi Before the device can synchronize its time, it must be connected to a WiFi network. This step is identical to the WiFi configuration process outlined earlier.

```
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()?;
```

13.0.20.3 Create and Configure an SNTP Instance

Instantiating and configuring SNTP is straightforward, especially with the default configuration. The `EspSntp::new_default()` method initializes an SNTP client with default settings. Custom configurations are also possible, allowing the selection of different SNTP servers, operating modes, or synchronization modes as needed.

```
let ntp = EspSntp::new_default().unwrap();
```


```
let ntp = EspSntp::new_default().unwrap();
```


13.0.21 Interacting with SNTP

Once the SNTP instance is created and configured, the device can interact with it to manage time synchronization.

13.0.21.1 Synchronizing Time

The `EspSntp` abstraction provides the `get_sync_status` method to check the synchronization status with the NTP server. This method returns a `SyncStatus` enum, indicating whether the synchronization process is completed or still in progress.

```
pub enum SyncStatus {
    Reset,
    // Other variants...
}
```

To ensure that the system time has been synchronized, the following loop can be used:

```
while ntp.get_sync_status() != SyncStatus::Completed {
    // Wait or perform other tasks
}
```


```
while ntp.get_sync_status() != SyncStatus::Completed {
    // Wait or perform other tasks
}
```


Once synchronization is completed, the device's system time can be retrieved using Rust's standard library:

```
use std::time::SystemTime;

let current_time = SystemTime::now();
println!("Synchronized System Time: {:?}", current_time);
```

14 The Embassy Framework (no-std)

In a blocking approach, the processor sits idle, busy waiting (continuously polling) for a result. This not only wastes processing power but also prevents other code from executing concurrently. A potential solution to this issue is using interrupts, which can notify the application when an event occurs. However, configuring interrupts can be daunting and increases code verbosity.

14.0.0.1 Introducing Asynchronous Programming Asynchronous programming offers an elegant alternative and has gained significant popularity in Rust circles. Asynchronous (often abbreviated as "async") programming in Rust allows you to run multiple tasks concurrently while preserving the synchronous, readable nature of regular Rust code. Async programming in Rust involves three main components:

1. Futures: Represent work that may complete in the future.
2. Async/Await Syntax: Handles asynchronous tasks in a non-blocking manner.
3. Runtime: Executes the asynchronous tasks.

Rust's asynchronous operation is based on futures, which are polled in the background until they signal completion. The `async` keyword transforms a block of code into a future, and the `await` keyword is used to wait for the future to resolve without blocking the entire thread. The `await` keyword also implies deferred execution, allowing the program to handle other tasks while waiting for the future to complete.

Unlike some other languages, Rust does not include a built-in runtime for asynchronous operations. Instead, developers must choose a runtime, such as Tokio, async-std, smol, etc., and include it as a dependency. In embedded Rust development, the Embassy executor serves this purpose.

14.0.0.2 The Embassy Executor The Embassy executor is part of the Embassy framework, which offers more than just a runtime. Embassy provides an efficient and easy-to-use multitasking environment using Rust's `async/await` syntax. Additionally, Embassy offers Hardware Abstraction Layers (HALs) for select hardware, providing safe, idiomatic Rust APIs for hardware capabilities. It also includes crates for time management, connectivity, and networking features, among others. Embassy is poised to become a significant tool in embedded development by offering a lightweight runtime that facilitates writing multithreaded, efficient, and safe code without the overhead of a Real-Time Operating System (RTOS).

Notable Embassy Crates:

- embassy-time: Provides timekeeping, delays, and timeout abstractions.
- embassy-net: Offers an async network stack.
- embassy-sync: Supplies synchronization primitives and data structures with async support.
- embassy-usb: Delivers an async USB device stack.
- embassy-executor: Contains async/await executor abstractions.

Asynchronous programming is powerful and has been prevalent in areas like web development. In embedded development, it is relatively new and may take some time for wide-scale adoption. However, Embassy's stable and efficient abstractions make it a promising candidate for the future of embedded Rust development.

14.0.0.3 Getting Started with Embassy Getting started with the Embassy framework is straightforward. Technically, you need to:

1. Import the Embassy crates.
2. Initialize the background executor.
3. Use async abstractions.

Condition: The underlying HAL used with Embassy must support async development. This means the HAL should implement or provide access to async functions. ESP devices support the Embassy framework by leveraging community-driven HALs, such as `embedded-hal-async` and `embedded-io-async`, maintained by the embedded Rust workgroup. These HALs establish common behavior among devices, enabling portability across a wider range of hardware platforms.

Note: Currently, no-std Embassy support with ESPs is limited to peripherals supported by the `embedded-hal-async` and `embedded-io-async` crates. Not all peripherals have async abstractions, so consult the documentation for supported peripherals.

Below is a basic template demonstrating the use of Embassy with the `esp-hal`. This template includes two tasks: the main task and an `embassy_task` spawned by main. Key differences from earlier templates include:

1. `main` Function Declared as `async`: Allows deferring execution to the background executor and awaiting futures.
2. Spawner Handle: Passed to `main` to spawn additional tasks.
3. Initialization of the Embassy Executor: Using `embassy::init` with an async timer instance.
4. Spawning Tasks: Using `spawner.spawn` to add tasks to the executor.
5. Delay Creation with `Timer::after`: Creates a future that resolves after a specified duration.

```rust
#![no_std]
#![no_main]

use embassy_executor::Spawner;
use embassy_time::{Duration, Timer};
use esp_backtrace as _;
use esp_hal::{prelude::*, timer::timg::TimerGroup};
use esp_hal_embassy;
use esp_println::println;

#[embassy_executor::task]
async fn embassy_task() {
    loop {
        // Task Loop Code
        println!("Print from an embassy task");
        Timer::after(Duration::from_millis(1_000)).await;
    }
}

#[main]
async fn main(spawner: Spawner) {
    println!("Init!");
    let peripherals = esp_hal::init(esp_hal::Config::default());

    // Initialize Embassy executor
    let timg0 = TimerGroup::new(peripherals.TIMG0);
    esp_hal_embassy::init(timg0.timer0);

    spawner.spawn(embassy_task()).unwrap();

    loop {
        // Main loop code
```

```
        println!("Print from the main task");
        Timer::after(Duration::from_millis(5_000)).await;
    }
}
```

```
[dependencies]
esp-backtrace = { version = "0.14.1", features = [
    "esp32c3",
    "exception-handler",
    "panic-handler",
    "println",
] }
esp-hal = { version = "0.21.0", features = ["esp32c3"] }
esp-println = { version = "0.12.0", features = ["esp32c3", "log"] }
log = { version = "0.4.20" }
esp-hal-embassy = { version = "0.4.0", features = [
    "esp32c3",
    "integrated-timers",
    "log",
] }
embassy-executor = { version = "0.6.0", features = ["task-arena-size-40960"] }
embassy-futures = "0.1.1"
embassy-sync = "0.6.0"
embassy-time = "0.3.2"
embedded-hal-async = "1.0.0"
embedded-io-async = "0.6.1"
```

Explanation of `Cargo.toml` Dependencies:

- esp-backtrace: Handles backtraces for debugging.
- esp-hal: Provides Hardware Abstraction Layer for ESP devices.
- esp-println: Facilitates printing/logging capabilities.
- log: Logging facade.
- esp-hal-embassy: Integrates Embassy with the ESP HAL.
- embassy-executor: Async executor for Embassy.
- embassy-futures: Future abstractions for Embassy.
- embassy-sync: Synchronization primitives for async tasks.
- embassy-time: Time management utilities for Embassy.
- embedded-hal-async: Async traits for embedded HAL.
- embedded-io-async: Async traits for embedded IO.

14.0.0.4 Synchronization Primitives

In the following sections, we will explore some of Embassy's synchronization primitives that are particularly useful when sharing data among tasks or threads. These primitives help manage access to shared resources without introducing race conditions, ensuring thread-safe operations in an asynchronous environment.

14.0.1 Synchronization Primitives

When introducing interrupts earlier, dealing with global variables presented significant challenges. The primary issue stems from ensuring that variables are shared safely among threads to prevent synchronization problems like data races. However, using the Embassy framework, this experience is greatly improved through several synchronization primitives provided by the `embassy-sync` crate.

But with several primitives available, how do we decide which one to use? The answer lies in how you plan to share the data. Specifically, consider whether you want to:

1. Share data among tasks in a blocking manner
2. Require async support for shared data
3. Need the primitive to notify a task when the data it holds changes

The `embassy-sync` crate offers the following primitives to cater to these scenarios:

- Channel: A Multiple Producer Multiple Consumer (MPMC) channel where each message sent is received by a single consumer.
- PubSubChannel: A broadcast (publish-subscribe) channel where each message sent is received by all consumers.
- Signal: Signals the latest value to a single consumer.
- Mutex: Synchronizes state between asynchronous tasks.
- Pipe: A byte stream that implements `embedded-io` traits.

Additionally, there are Waker primitives, which are utilities to signal the executor to poll a Future. These include:

- WakerRegistration: Utility to register and wake a Waker.
- AtomicWaker: A variant of `WakerRegistration` accessible using a non-mut API.
- MultiWakerRegistration: Utility for registering and waking multiple Wakers.

14.0.1.1 Use Cases

The use of different synchronization primitives can be categorized into three main cases:

1. Reading/Writing from/to Multiple Tasks: Sharing simple data among multiple tasks.
2. Reading/Writing Across Async Tasks: Sharing data across asynchronous tasks, requiring safe mutation while awaiting.
3. Wait for Value Change: Scenarios where a receiving task waits for a change in a value.

In the following sections, we will explore the constructs available under each category.

14.0.1.2 The `AtomicU32` Type

While `AtomicU32` is not explicitly listed among the Embassy synchronization primitives, it is a valuable tool available in Rust's `core::sync::atomic` module. It is especially handy when you need to share a simple value among tasks without the overhead of more complex synchronization mechanisms. However, `AtomicU32` works only for types that are `u32` or smaller in size. For larger types, you should use a global blocking `Mutex`.

```
#![no_std]
#![no_main]

use core::sync::atomic::{AtomicU32, Ordering};
use embassy_executor::Spawner;
use embassy_time::{Duration, Timer};
use esp_backtrace as _;
use esp_hal::{prelude::*, timer::timg::TimerGroup};
use esp_println::println;

// Shared AtomicU32 variable initialized to 0
static SHARED: AtomicU32 = AtomicU32::new(0);

#[embassy_executor::task]
```

```rust
async fn async_task() {
    loop {
        // Load the current value, increment, and store it back
        let shared_var = SHARED.load(Ordering::Relaxed);
        SHARED.store(shared_var.wrapping_add(1), Ordering::Relaxed);

        // Wait for 1 second
        Timer::after(Duration::from_millis(1000)).await;
    }
}

#[main]
async fn main(spawner: Spawner) {
    // Initialize and create handle for device peripherals
    let peripherals = esp_hal::init(esp_hal::Config::default());

    // Initialize Embassy executor
    let timg0 = TimerGroup::new(peripherals.TIMG0);
    esp_hal_embassy::init(timg0.timer0);

    // Spawn the asynchronous task
    spawner.spawn(async_task()).unwrap();

    loop {
        // Load the current shared value
        let shared = SHARED.load(Ordering::Relaxed);

        // Wait for 1 second
        Timer::after(Duration::from_millis(1000)).await;

        // Print the shared value
        println!("{}", shared);
    }
}
```

Explanation:

1. Global Variable Setup: A global `SHARED` variable of type `AtomicU32` is defined and initialized to `0`.
2. Asynchronous Task (`async_task`):
   - Continuously increments the `SHARED` variable every second.
   - Uses `load` with `Ordering::Relaxed` to retrieve the current value.
   - Uses `store` with `Ordering::Relaxed` to update the value.
3. Main Function:
   - Initializes peripherals and the Embassy executor.
   - Spawns the `async_task`.
   - In the main loop, it reads and prints the `SHARED` value every second.

Note: `Ordering::Relaxed` is used here for simplicity, assuming that the exact ordering of operations is not critical. For more complex synchronization requirements, stronger memory ordering guarantees may be necessary.

14.0.1.3 The Blocking `Mutex` Type

For scenarios where you need to share data types larger than `u32` or require more complex synchronization, a blocking `Mutex` is the appropriate choice. The blocking `Mutex` ensures safe access to shared data by allowing only one task to access the data at a time, effectively preventing data races.

```
#![no_std]
#![no_main]

use core::cell::RefCell;
use embassy_executor::Spawner;
use embassy_sync::blocking_mutex::raw::CriticalSectionRawMutex;
use embassy_sync::blocking_mutex::Mutex;
use embassy_time::{Duration, Timer};
use esp_backtrace as _;
use esp_hal::{prelude::*, timer::timg::TimerGroup};
use esp_println::println;

// Shared Mutex-protected u32 variable initialized to 0
static SHARED: Mutex<CriticalSectionRawMutex, RefCell<u32>> = Mutex::new(RefCell::new(0));

#[embassy_executor::task]
async fn async_task() {
    loop {
        // Acquire the mutex lock and modify the shared value
        SHARED.lock(|f| {
            let val = f.borrow_mut().wrapping_add(1);
            f.replace(val);
        });

        // Wait for 1 second
        Timer::after(Duration::from_millis(1000)).await;
    }
}

#[main]
async fn main(spawner: Spawner) {
    // Initialize and create handle for device peripherals
    let peripherals = esp_hal::init(esp_hal::Config::default());

    // Initialize Embassy executor
    let timg0 = TimerGroup::new(peripherals.TIMG0);
    esp_hal_embassy::init(timg0.timer0);

    // Spawn the asynchronous task
    spawner.spawn(async_task()).unwrap();

    loop {
        // Wait for 1 second
        Timer::after(Duration::from_millis(1000)).await;

        // Acquire the mutex lock and read the shared value
        let shared = SHARED.lock(|f| f.borrow().clone());
```

```
        // Print the shared value
        println!("{}", shared);
    }
}
```

Explanation:

1. Global Variable Setup: A global `SHARED` variable is defined as a `Mutex` protecting a `RefCell<u32>`, initialized to `0`.
2. Asynchronous Task (`async_task`):
   - Continuously increments the `SHARED` variable every second.
   - Uses the `lock` method to safely access and modify the shared data.
3. Main Function:
   - Initializes peripherals and the Embassy executor.
   - Spawns the `async_task`.
   - In the main loop, it acquires the mutex lock to read the `SHARED` value every second and prints it.

Key Differences from `AtomicU32`:

- Data Type Flexibility: The `Mutex` allows for sharing data types larger than `u32`, providing greater flexibility.
- Locking Mechanism: The `Mutex` ensures that only one task can access the shared data at a time, preventing concurrent modifications and ensuring data integrity.
- No Atomic Operations: Unlike `AtomicU32`, which provides atomic operations, the `Mutex` relies on locking to synchronize access.

Note: The `CriticalSectionRawMutex` is chosen for this example, but depending on the context and requirements, other mutex types provided by Embassy may be more appropriate.

14.0.1.4 The `async Mutex` Type

While the blocking `Mutex` ensures safe access to shared data by allowing only one task to hold the lock at a time, it does not hold the lock across `await` points. In contrast, the `async Mutex` provided by the `embassy-sync` crate allows a task to `await` while holding the lock, enabling other tasks to proceed without being blocked indefinitely.

Key Differences: - Blocking `Mutex`: - Does not support holding the lock across `await` points. - Suitable for scenarios where tasks do not need to hold the lock while awaiting. - `async Mutex`: - Allows holding the lock across `await` points. - Suitable for scenarios where tasks need to hold the lock while performing asynchronous operations.

```
#![no_std]
#![no_main]

use embassy_executor::Spawner;
use embassy_sync::mutex::Mutex;
use embassy_time::{Duration, Timer};
use esp_backtrace as _;
use esp_hal::{prelude::*, timer::timg::TimerGroup};
use esp_hal_embassy;
use esp_println::println;

// Shared async Mutex-protected u32 variable initialized to 0
static SHARED: Mutex<embassy_sync::mutex::raw::CriticalSectionRawMutex, u32>
  = Mutex::new(0);
```

```rust
#[embassy_executor::task]
async fn async_task() {
    loop {
        {
            // Acquire the mutex lock and modify the shared value
            let mut shared = SHARED.lock().await;
            *shared = shared.wrapping_add(1);

            // Hold the lock while awaiting (simulated by a delay)
            Timer::after(Duration::from_millis(1000)).await;
        }
        // The lock is automatically released here
        Timer::after(Duration::from_millis(1000)).await;
    }
}

#[main]
async fn main(spawner: Spawner) {
    // Initialize and create handle for device peripherals
    let peripherals = esp_hal::init(esp_hal::Config::default());

    // Initialize Embassy executor
    let timg0 = TimerGroup::new(peripherals.TIMG0);
    esp_hal_embassy::init(timg0.timer0);

    // Spawn the asynchronous task
    spawner.spawn(async_task()).unwrap();

    loop {
        // Wait for 1 second
        Timer::after(Duration::from_millis(1000)).await;

        // Acquire the mutex lock and read the shared value
        let shared = SHARED.lock().await;

        // Print the shared value
        println!("{}", shared);
    }
}
```

Explanation:

1. Global Variable Setup:
   - A global `SHARED` variable is defined as an `async Mutex` protecting a `u32`, initialized to `0`.
2. Asynchronous Task (`async_task`):
   - Continuously increments the `SHARED` variable every second.
   - Uses the `lock().await` method to safely acquire and hold the mutex lock.
   - The lock is held while awaiting the timer, ensuring that no other task can access `SHARED` during this period.
3. Main Function:
   - Initializes peripherals and the Embassy executor.

- Spawns the `async_task`.
- In the main loop, it waits for one second, acquires the mutex lock to read the `SHARED` value, and prints it.

Behavior: - The `async_task` increments the shared value every second while holding the lock during the delay. - The main task attempts to read the shared value every second. If the `async_task` is holding the lock (during its `await`), the main task will wait until the lock becomes available. - This ensures synchronized access to the shared variable without data races.

Important Considerations: - Lock Scope: The lock is held within a scoped block `{ ... }`, ensuring it is released before the main task attempts to acquire it again. - Ordering: Unlike atomic types, mutexes ensure exclusive access, making them suitable for more complex data manipulation.

14.0.1.5 The `Signal` Type

The `Signal` primitive is ideal for scenarios where a task needs to be notified when a particular value changes. It provides a simple mechanism to buffer or send a new value to another task, effectively signaling that an update has occurred.

Use Case: - When one task needs to notify another task about a specific event or data change without continuous polling.

```
#![no_std]
#![no_main]

use embassy_executor::Spawner;
use embassy_sync::signal::Signal;
use embassy_time::{Duration, Timer};
use esp_backtrace as _;
use esp_hal::{prelude::*, timer::timg::TimerGroup};
use esp_hal_embassy;
use esp_println::println;

// Shared Signal-protected u32 variable
static SHARED: Signal<embassy_sync::mutex::raw::CriticalSectionRawMutex, u32>
  = Signal::new();

#[embassy_executor::task]
async fn async_task() {
    loop {
        // Signal the value 5
        SHARED.signal(5);

        // Wait for 1 second
        Timer::after(Duration::from_millis(1000)).await;
    }
}

#[main]
async fn main(spawner: Spawner) {
    // Initialize and create handle for device peripherals
    let peripherals = esp_hal::init(esp_hal::Config::default());

    // Initialize Embassy executor
```

```
    let timg0 = TimerGroup::new(peripherals.TIMG0);
    esp_hal_embassy::init(timg0.timer0);

    // Spawn the asynchronous task
    spawner.spawn(async_task()).unwrap();

    loop {
        // Wait for a signal and retrieve the value
        let val = SHARED.wait().await;

        // Print the received value
        println!("{}", val);
    }
}
```

Explanation:

1. Global Variable Setup:
   - A global `SHARED` variable is defined as a `Signal` protecting a `u32`.
2. Asynchronous Task (`async_task`):
   - Continuously sends (signals) the value `5` every second using the `signal` method.
3. Main Function:
   - Initializes peripherals and the Embassy executor.
   - Spawns the `async_task`.
   - In the main loop, it awaits a signal using the `wait().await` method and prints the received value.

Behavior: - The `async_task` sends a signal with the value `5` every second. - The main task waits for the signal and, upon receiving it, prints the value. - This setup eliminates the need for the main task to continuously poll for updates, making the communication more efficient.

Advantages Over `AtomicU32`: - Event-Driven: `Signal` allows tasks to react to specific events or changes rather than continuously checking for updates. - Buffering: Signals can buffer values, ensuring that important updates are not missed even if the receiving task is busy.

14.0.1.6 The `Channel` Type

The `Channel` primitive expands upon the capabilities of `Signal` by allowing multiple values to be buffered in a queue. It supports multiple producers and multiple consumers, making it suitable for scenarios where multiple tasks need to send and receive messages concurrently.

Use Case: - When you need to buffer multiple values sent by producers and have them processed by consumers in a first-come, first-served manner.

```
#![no_std]
#![no_main]

use embassy_executor::Spawner;
use embassy_sync::channel::Channel;
use embassy_time::{Duration, Timer};
use esp_backtrace as _;
use esp_hal::{prelude::*, timer::timg::TimerGroup};
use esp_hal_embassy;
```

```rust
use esp_println::println;

// Declare a channel with capacity of 2 u32s
static SHARED: Channel<embassy_sync::mutex::raw::CriticalSectionRawMutex, u32, 2>
  = Channel::new();

#[embassy_executor::task]
async fn async_task_one() {
    loop {
        // Send the value 1
        SHARED.send(1).await;

        // Wait for 0.5 seconds
        Timer::after(Duration::from_millis(500)).await;
    }
}

#[embassy_executor::task]
async fn async_task_two() {
    loop {
        // Send the value 2
        SHARED.send(2).await;

        // Wait for 1 second
        Timer::after(Duration::from_millis(1000)).await;
    }
}

#[main]
async fn main(spawner: Spawner) {
    // Initialize and create handle for device peripherals
    let peripherals = esp_hal::init(esp_hal::Config::default());

    // Initialize Embassy executor
    let timg0 = TimerGroup::new(peripherals.TIMG0);
    esp_hal_embassy::init(timg0.timer0);

    // Spawn asynchronous tasks
    spawner.spawn(async_task_one()).unwrap();
    spawner.spawn(async_task_two()).unwrap();

    loop {
        // Receive a value from the channel
        let val = SHARED.receive().await;

        // Print the received value
        println!("{}", val);
    }
}
```

Explanation:

1. Global Variable Setup:
    - A global `SHARED` channel is defined with a capacity of `2` for `u32` values.
2. Asynchronous Tasks (`async_task_one` and `async_task_two`):
    - `async_task_one` sends the value `1` every 0.5 seconds.
    - `async_task_two` sends the value `2` every 1 second.
3. Main Function:
    - Initializes peripherals and the Embassy executor.
    - Spawns both asynchronous tasks.
    - In the main loop, it awaits messages from the `SHARED` channel and prints them as they are received.

Behavior: - The channel buffers up to two messages. - `async_task_one` sends `1` every 0.5 seconds, and `async_task_two` sends `2` every 1 second. - The main task receives and prints each value in the order they are sent. - If the channel is full, producers will wait (`await`) until there is space available.

Important Notes: - Capacity: The channel's capacity determines how many messages can be buffered before producers are blocked. - Multiple Producers: Both `async_task_one` and `async_task_two` act as producers, sending messages to the channel. - Single Consumer: The main task acts as the sole consumer, receiving and processing messages.

14.0.1.7 The `PubSubChannel` Type

The `PubSubChannel` is an extension of the `Channel` type that supports a publish-subscribe (pub-sub) communication model. Unlike `Channel`, where each message is consumed by a single consumer, `PubSubChannel` allows multiple subscribers to receive each published message.

Use Case: - When you need to broadcast messages to multiple consumers, ensuring that each subscriber receives every message.

```
#![no_std]
#![no_main]

use embassy_executor::Spawner;
use embassy_sync::pubsub::PubSubChannel;
use embassy_time::{Duration, Timer};
use esp_backtrace as _;
use esp_hal::{prelude::*, timer::timg::TimerGroup};
use esp_hal_embassy;
use esp_println::println;

// Declare a PubSub channel with capacity of 2 messages,
// 2 publishers, and 2 subscribers
static SHARED: PubSubChannel<
    embassy_sync::mutex::raw::CriticalSectionRawMutex,
    u32,
    2, // Capacity
    2, // Number of publishers
    2, // Number of subscribers
> = PubSubChannel::new();

#[embassy_executor::task]
async fn async_task_one() {
    // Obtain a publisher
```

```
    let pub1 = SHARED.publisher().unwrap();
    loop {
        // Publish the value 1 immediately
        pub1.publish_immediate(1);

        // Wait for 0.5 seconds
        Timer::after(Duration::from_millis(500)).await;
    }
}

#[embassy_executor::task]
async fn async_task_two() {
    // Obtain a publisher
    let pub2 = SHARED.publisher().unwrap();
    loop {
        // Publish the value 2 immediately
        pub2.publish_immediate(2);

        // Wait for 1 second
        Timer::after(Duration::from_millis(1000)).await;
    }
}

#[main]
async fn main(spawner: Spawner) {
    // Initialize and create handle for device peripherals
    let peripherals = esp_hal::init(esp_hal::Config::default());

    // Initialize Embassy executor
    let timg0 = TimerGroup::new(peripherals.TIMG0);
    esp_hal_embassy::init(timg0.timer0);

    // Spawn asynchronous tasks
    spawner.spawn(async_task_one()).unwrap();
    spawner.spawn(async_task_two()).unwrap();

    // Obtain a subscriber
    let mut sub = SHARED.subscriber().unwrap();

    loop {
        // Wait for the next message from the subscriber
        let val = sub.next_message_pure().await;

        // Print the received value
        println!("{}", val);
    }
}
```

Explanation:

1. Global Variable Setup:
    - A global `SHARED` `PubSubChannel` is defined with a capacity of 2, supporting 2 publishers

and 2 subscribers.

2. Asynchronous Tasks (`async_task_one` and `async_task_two`):
   - Both tasks obtain their own publishers (`pub1` and `pub2`).
   - `async_task_one` publishes the value `1` every 0.5 seconds.
   - `async_task_two` publishes the value `2` every 1 second.
3. Main Function:
   - Initializes peripherals and the Embassy executor.
   - Spawns both asynchronous tasks.
   - Obtains a subscriber (`sub`) to receive messages.
   - In the main loop, it waits for messages from the subscriber and prints them as they are received.

Behavior: - Each published message is broadcasted to all subscribers. - In this example, the main task acts as a single subscriber receiving messages from both publishers. - If multiple subscribers were present, each would receive every message published to the channel. - If a subscriber misses a message because it was busy or not ready, it will receive an error signaling that occurrence.

Important Considerations: - Message Delivery: Every message published is delivered to all active subscribers. - Error Handling: Subscribers may receive errors if they miss messages due to buffer overflows or other issues. - Capacity Management: The channel's capacity should be chosen based on the expected message rate and subscriber readiness to prevent message loss.