

Rust

Rust ESP Book was referenced for most of this writing.

Some context was added for deeper understanding of high-level requirements of a rust esp project.

Project Types

There are two project types to consider when creating esp32 rust projects. It mostly depends on your hardware or application requirements, but there may be other factors to consider as well.

Using std

These projects include Rust's standard library and depend on the ESP-IDF framework as an underlying OS. They support features like heap allocation, file systems, networking, and standard error handling. Because of the range of features and implementations available in `std`, this is better for applications that have more complex logic and more resources available to run. While probably not technically required in all applications, you could expect an application like this to require AC power connection to run.

Using no std

`no_std` projects exclude the standard library and operate without an operating system. They use only the Rust core library and sometimes the `alloc` crate for heap-like behavior. This approach is ideal for low-level, resource-constrained, or timing-critical applications where full control over the hardware is required. For example this would be best for an application that is going to run on battery power with tight resources.

Dependencies

There are a few things we'll need to install before we can work on ESP projects with rust. The following sections will cover each required step.

Installing espup

This section was in [RISCV and Xtensa](#) requirements in the docs, but from my attempts locally I could not proceed even with ESP32 without it.

This tool will install the needed libraries for working with ESP32 microcontrollers with Rust.

```
# This is required if you are working with esp-idf std projects. If using no std this is not required.  
cargo install ldproxy
```

```
# This is required for all applications.  
cargo install espup --locked
```

Running `espup` will give us this output that is important to read.

```
espup install  
[info]: Installing the Espressif Rust ecosystem  
[info]: Checking Rust installation  
[info]: Installing Xtensa Rust 1.87.0.0 toolchain  
[info]: Installing RISC-V Rust targets ('riscv32imc-unknown-none-elf', 'riscv32imac-unknown-none-elf' and  
'riscv32imafc-unknown-none-elf') for 'stable' toolchain  
[info]: Installing Xtensa LLVM  
[info]: Installing GCC (xtensa-esp-elf)  
[info]: Creating symlink between '/home/shaun/.rustup/toolchains/esp/xtensa-esp32-elf-clang/esp-  
19.1.2_20250225/esp-clang/lib' and '/home/shaun/.espup/esp-clang'  
[info]: All downloads complete  
[info]: Installing 'rust' component for Xtensa Rust toolchain  
[info]: Installing 'rust-src' component for Xtensa Rust toolchain  
[info]: Installation successfully completed!
```

To get started, you need to set up some environment variables by running: `./home/shaun/export-esp.sh`

This step must be done every time you open a new terminal.

See other methods for setting the environment in <https://esp-rs.github.io/book/installation/riscv-and-xtensa.html#3-set-up-the-environment-variables>

So every time we want to work on esp-idf things with rust we need to run this script to set some variables.

```
./export-esp.sh
```

Project Generators

This section covers using generators to create a new ESP32 rust project. There are two options, so the two sections that follow will describe each one. You only need to pick one.

Installing cargo-generate

Installed with

```
cargo install cargo-generate
```

This is more configurable and can be used with `cargo-generate.toml` configurations stored on a remote git repository. This is not just for ESP project but can be used with a range of rust projects. For example to use a cargo generate project template for esp-idf (using `std`) we can run this command

```
cargo generate --git https://github.com/esp-rs/esp-idf-template.git --name esp-idf-rust-std -d mcu=esp32 -d std=true
```

To generate a project with no `std`

```
cargo generate --git https://github.com/esp-rs/esp-idf-template.git --name esp-idf-rust-no-std -d mcu=esp32 -d std=false
```

Installing esp-generate

Understanding esp-generate

Installed with

```
cargo install esp-generate
```

To generate a ESP32 rust project using `std` run the following command

```
esp-generate new esp-idf-rust-std --mcu esp32 --std
```

To generate a project with no `std`

```
esp-generate new esp-idf-rust-no-std --mcu esp32 --no-std
```

Build

Regardless of the method you chose for generating your project, your build steps will be the same.

Don't forget to source the script `espup` told us we needed when we were installing dependencies.

```
cd esp-rust-project

# Export the script that espup informed us we needed to build esp rust projects before you build
. ~/export-esp.sh

cargo build
```

Now we've built the project, and we just need to flash it to an ESP32 device to test it.

Flash

Install `espflash` and configure your user to the required `dialout` group.

```
cargo install espflash
sudo usermod -aG dialout $USER
newgrp dialout
```

Pay careful attention to the path passed to `espflash` below. If you are using a different microcontroller or project name it will need to be changed to match.

```
espflash flash target/xtensa-esp32-espidf/debug/esp-idf-rust-std
```

You can also just use cargo to flash to the device automatically. If you have more than one device this may not work.

```
cargo run
  Finished `dev` profile [optimized + debuginfo] target(s) in 0.07s
  Running `espflash flash --monitor target/xtensa-esp32-espidf/debug/esp-idf-rust-std`
[2025-07-06T16:56:15Z INFO ] Serial port: '/dev/ttyUSB0'
[2025-07-06T16:56:15Z INFO ] Connecting...
[2025-07-06T16:56:15Z INFO ] Using flash stub
Chip type:      esp32 (revision v3.1)
Crystal frequency: 40 MHz
Flash size:     4MB
Features:       WiFi, BT, Dual Core, 240MHz, VRef calibration in efuse, Coding Scheme None
MAC address:    3c:8a:1f:0c:53:60
App/part. size: 561,504/4,128,768 bytes, 13.60%
...
```

Then open the serial monitor for your device. If there is only one detected this will use it automatically, otherwise you will receive a prompt to select the device. Below my device was automatically selected, so what you see is the connection being established.

```
esflash monitor
[2025-07-06T16:24:40Z INFO ] Serial port: '/dev/ttyUSB0'
[2025-07-06T16:24:40Z INFO ] Connecting...
[2025-07-06T16:24:40Z INFO ] Using flash stub
Commands:
  CTRL+R  Reset chip
  CTRL+C  Exit
```

Press `CTRL+R` to reset the chip and you'll see the `Hello, world!` output in the [template project we used for generation](#)

```
rst:0x1 (POWERON_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:2
load:0x3fff0030,len:6276
load:0x40078000,len:15748
load:0x40080400,len:4
ho 8 tail 4 room 4
load:0x40080404,len:3860
entry 0x4008063c
I (31) boot: ESP-IDF v5.4.1-426-g3ad36321ea 2nd stage bootloader
I (31) boot: compile time Apr 24 2025 15:52:44
I (31) boot: Multicore bootloader
I (34) boot: chip revision: v3.1
I (37) boot.esp32: SPI Speed      : 40MHz
I (40) boot.esp32: SPI Mode      : DIO
I (44) boot.esp32: SPI Flash Size : 4MB
I (48) boot: Enabling RNG early entropy source...
I (52) boot: Partition Table:
I (55) boot: ## Label          Usage          Type ST Offset  Length
I (61) boot: 0 nvs             WiFi data      01 02 00009000 00006000
I (67) boot: 1 phy_init         RF data        01 01 0000f000 00001000
I (74) boot: 2 factory          factory app     00 00 00010000 003f0000
I (81) boot: End of partition table
I (84) esp_image: segment 0: paddr=00010020 vaddr=3f400020 size=27dcch (163276) map
I (147) esp_image: segment 1: paddr=00037df4 vaddr=3ffb0000 size=026a8h ( 9896) load
```

```
I (151) esp_image: segment 2: paddr=0003a4a4 vaddr=40080000 size=05b74h ( 23412) load
I (160) esp_image: segment 3: paddr=00040020 vaddr=400d0020 size=4e97ch (321916) map
I (271) esp_image: segment 4: paddr=0008e9a4 vaddr=40085b74 size=05a5ch ( 23132) load
I (285) boot: Loaded app from partition at offset 0x10000
I (285) boot: Disabling RNG early entropy source...
I (295) cpu_start: Multicore app
I (304) cpu_start: Pro cpu start user code
I (304) cpu_start: cpu freq: 160000000 Hz
I (304) cpu_start: Application information:
I (307) cpu_start: Project name:   libespidf
I (312) cpu_start: App version:   1
I (317) cpu_start: Compile time:   Jul  6 2025 11:41:07
I (323) cpu_start: ELF file SHA256: 000000000...
I (328) cpu_start: ESP-IDF:       v5.2.3
I (333) cpu_start: Min chip rev:   v0.0
I (338) cpu_start: Max chip rev:   v3.99
I (342) cpu_start: Chip rev:       v3.1
I (347) heap_init: Initializing. RAM available for dynamic allocation:
I (355) heap_init: At 3FFAE6E0 len 00001920 (6 KiB): DRAM
I (360) heap_init: At 3FFB3018 len 0002CFE8 (179 KiB): DRAM
I (367) heap_init: At 3FFE0440 len 00003AE0 (14 KiB): D/IRAM
I (373) heap_init: At 3FFE4350 len 0001BCB0 (111 KiB): D/IRAM
I (380) heap_init: At 4008B5D0 len 00014A30 (82 KiB): IRAM
I (387) spi_flash: detected chip: generic
I (390) spi_flash: flash io: dio
W (395) pcnt(legacy): legacy driver is deprecated, please migrate to `driver/pulse_cnt.h`
W (403) i2c: This driver is an old driver, please migrate your application code to adapt `driver/i2c_master.h`
W (414) timer_group: legacy driver is deprecated, please migrate to `driver/gptimer.h`
I (423) main_task: Started on CPU0
I (433) main_task: Calling app_main()
I (433) esp_idf_rust_std: Hello, world!
```

Revision #7

Created 20 April 2025 13:28:59 by Shaun Reed

Updated 6 July 2025 17:10:30 by Shaun Reed