

Architectures for Embedded Systems

Introduction to ESP32, kit,
toolchain and examples
Laboratory assignments

Arnaldo S. R. Oliveira

16/02/2025 - Academic year 2024/25

Universidade de Aveiro – Dep. de Eletrónica, Telecomunicações e Informática

Outline

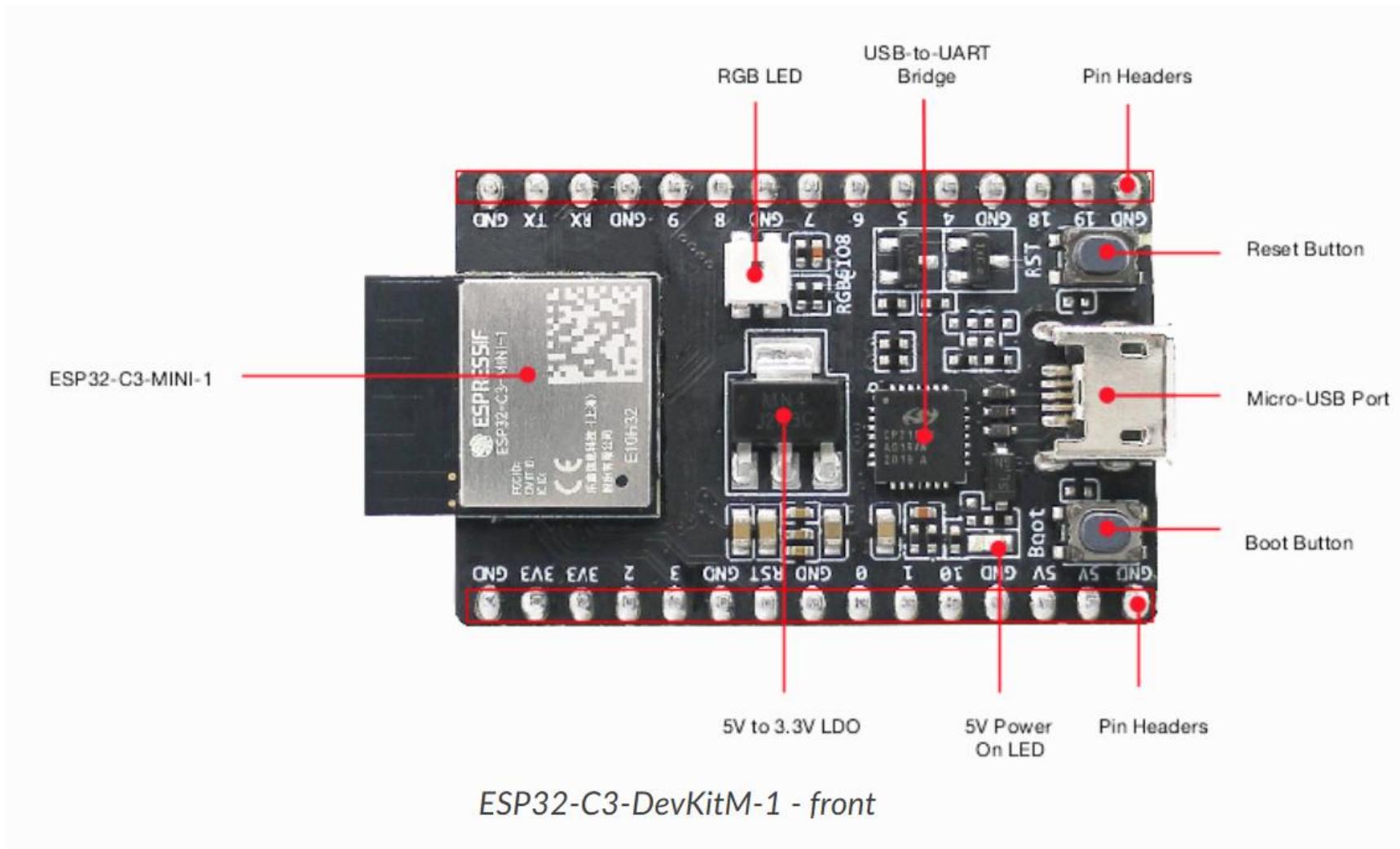
ESP32-C3 - the kit, the module and the SoC

Compilation toolchain

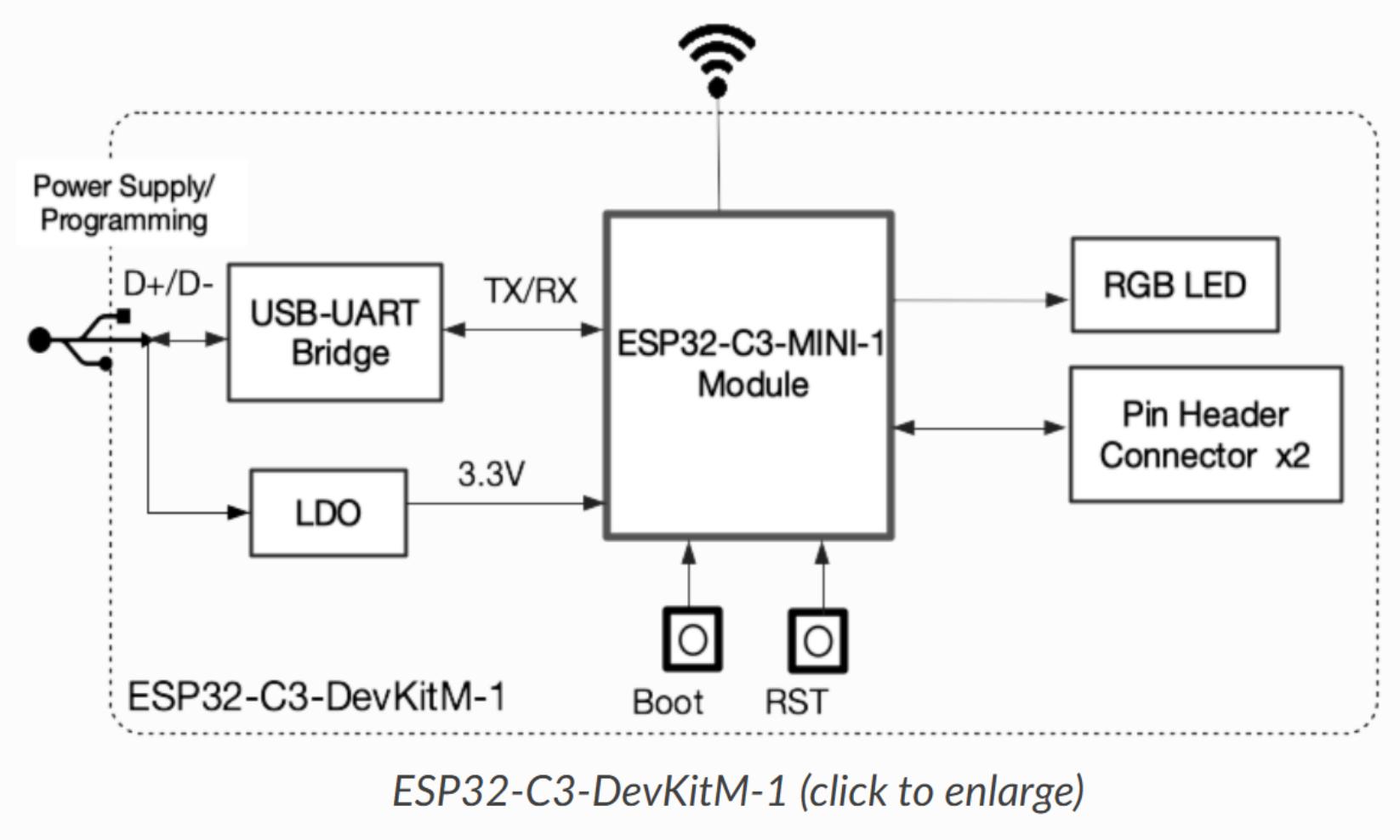
Get started examples

Lab assignments

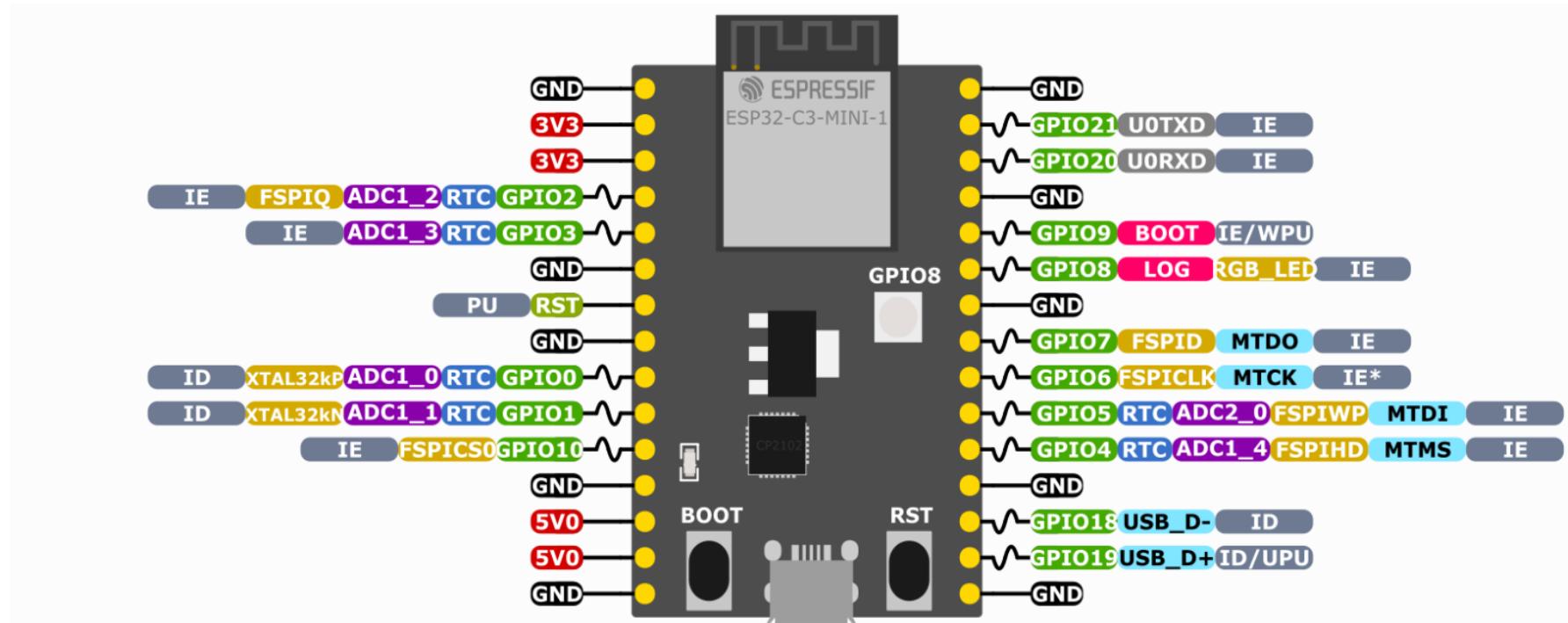
ESP32-C3-DevKitM-1 - Kit front view



ESP32-C3-DevKitM-1 - Kit block diagram



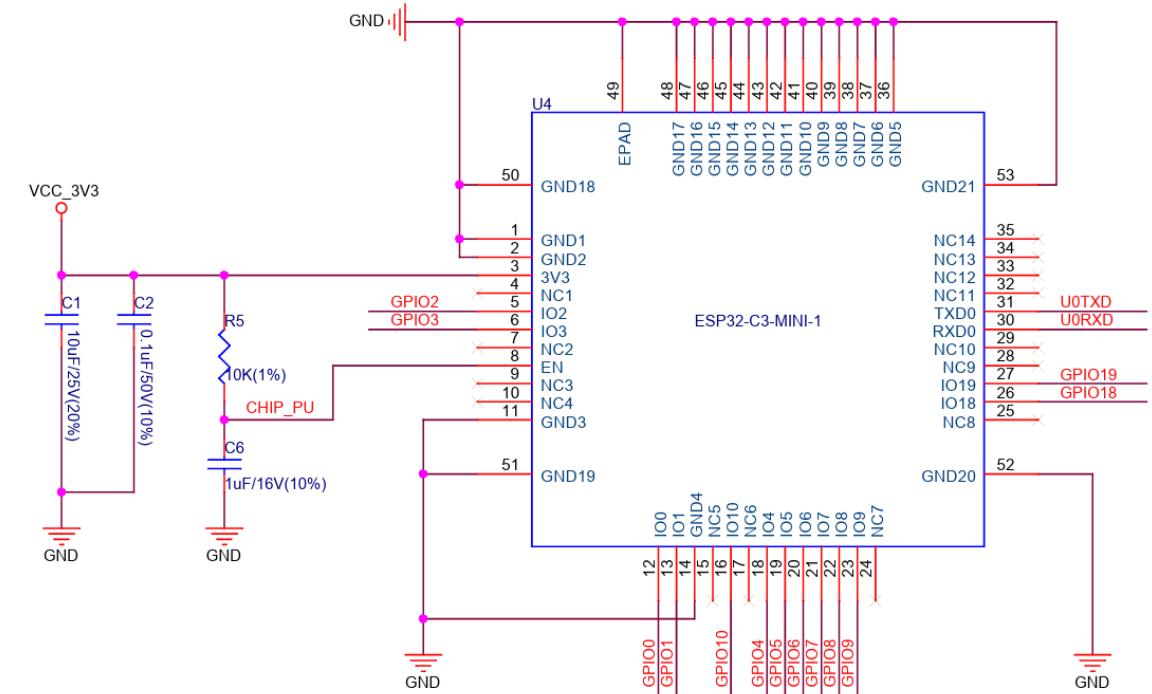
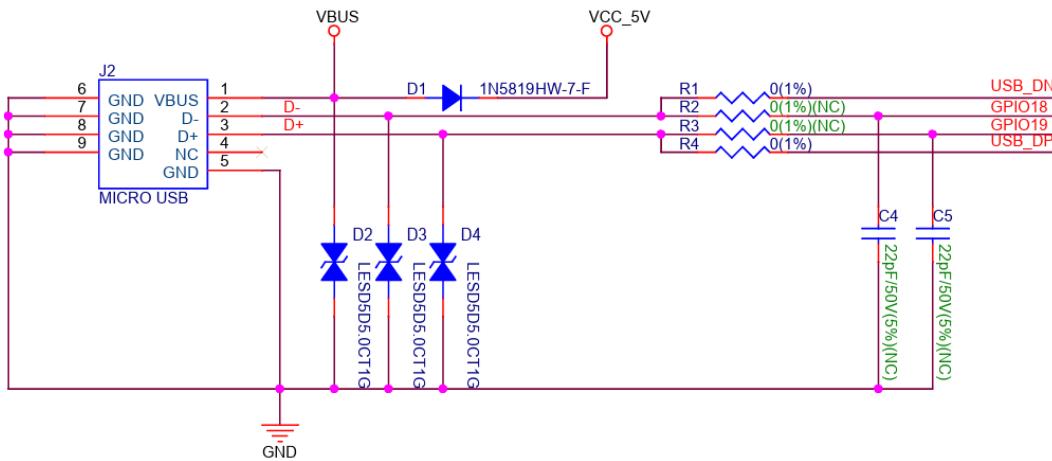
ESP32-C3-DevKitM-1 - Mux'ed pin functions



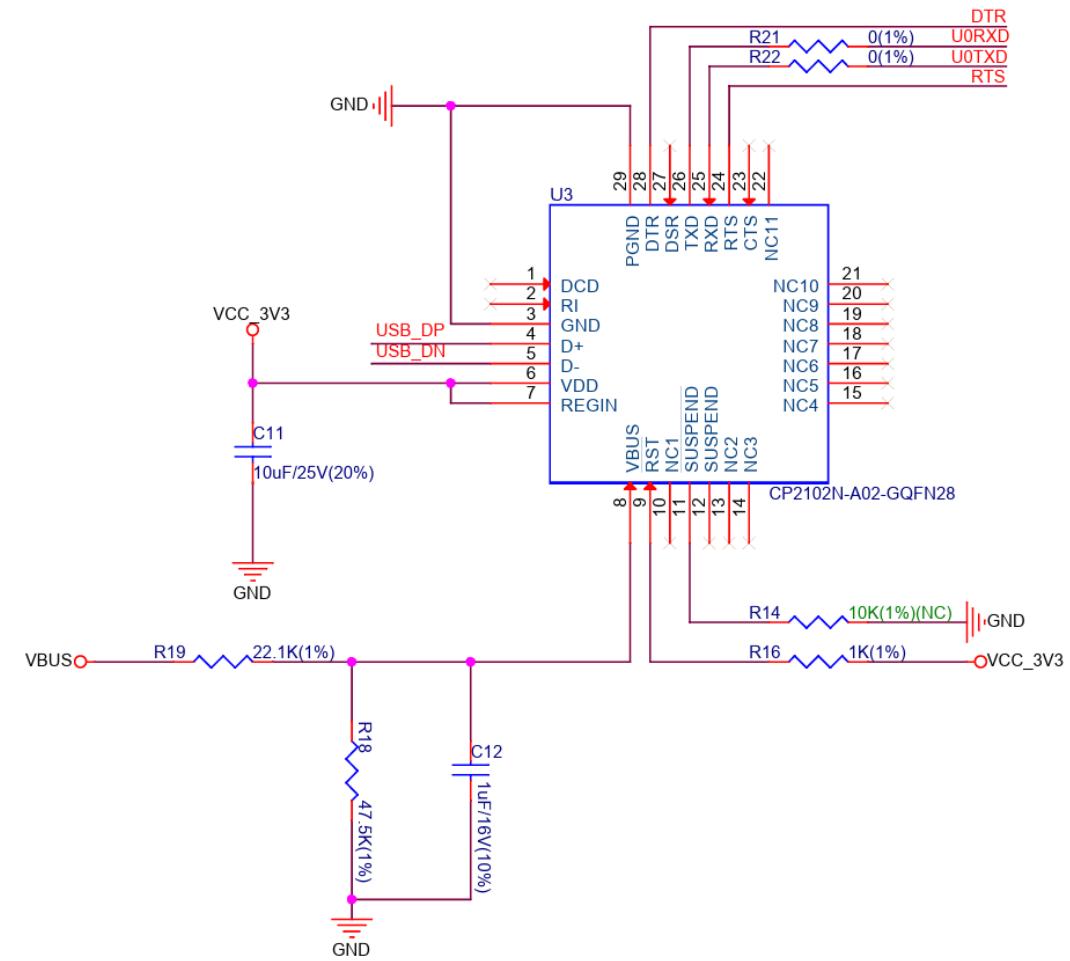
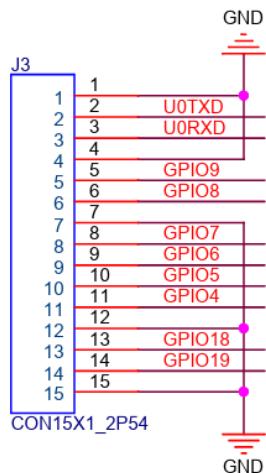
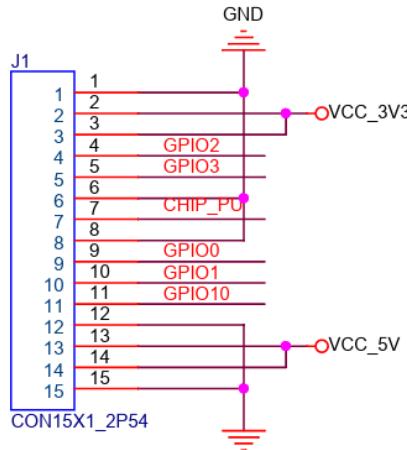
ESP32-C3 Specs

32-bit RISC-V single-core @160MHz
Wi-Fi IEEE 802.11 b/g/n 2.4GHz
Bluetooth LE 5
400 KB SRAM (16 KB for cache)
384 KB ROM
22 GPIOs, 3x SPI, 2x UART, I2C,
I2S, RMT, LED PWM, USB Serial/JTAG,
GDMA, TWAI®, 12-bit ADC

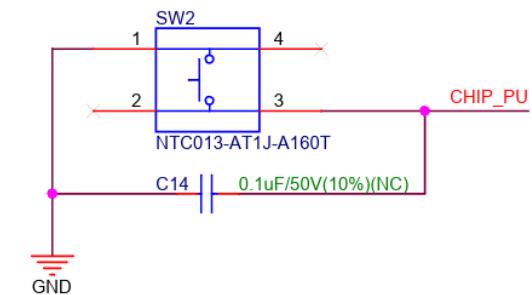
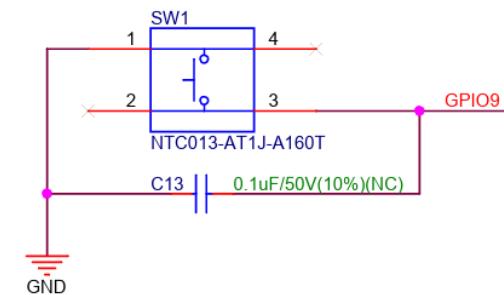
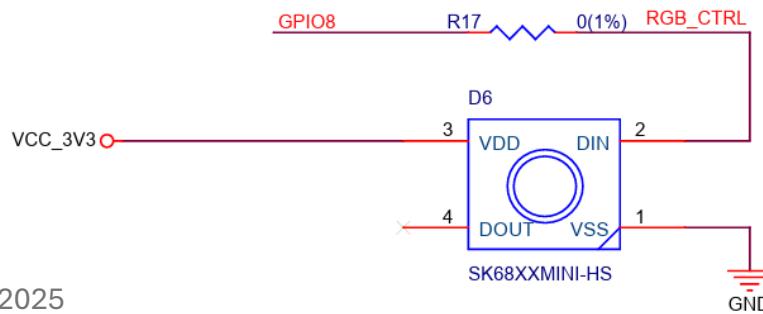
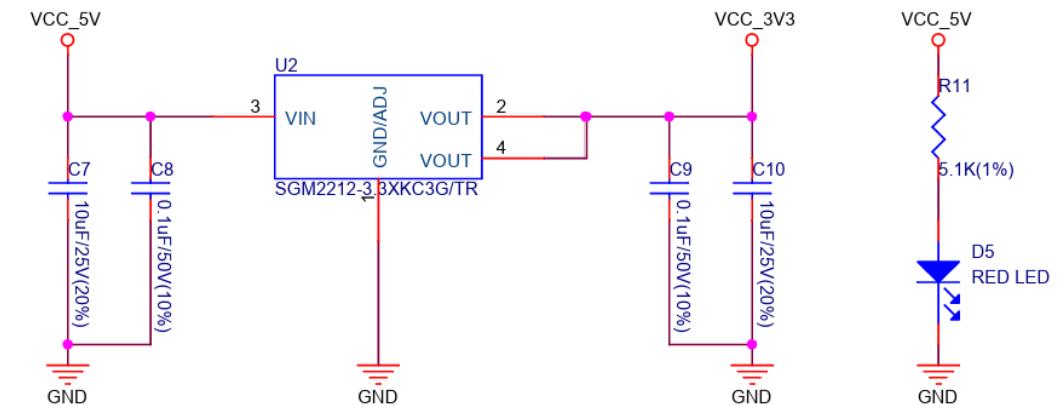
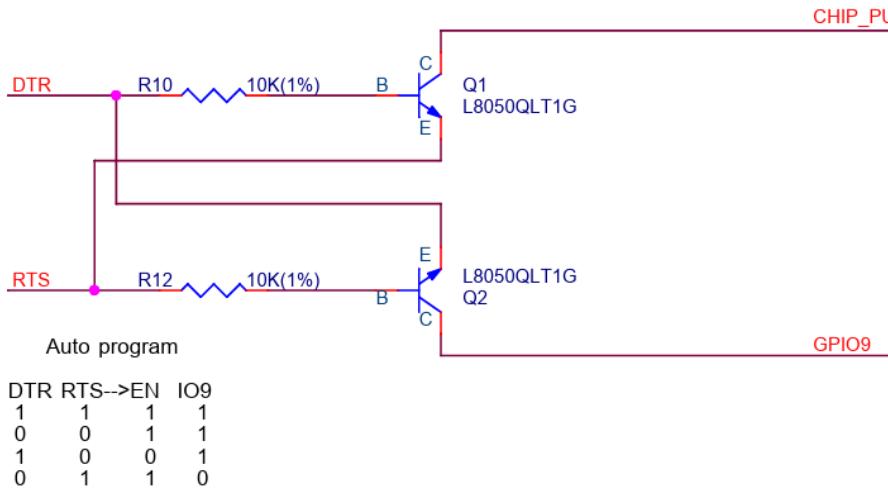
ESP32-C3-DevKitM-1 – Kit schematics (1/3)



ESP32-C3-DevKitM-1 – Kit schematics (2/3)



ESP32-C3-DevKitM-1 – Kit schematics (3/3)



ESP32-C3-MINI-1 - Module block diagram

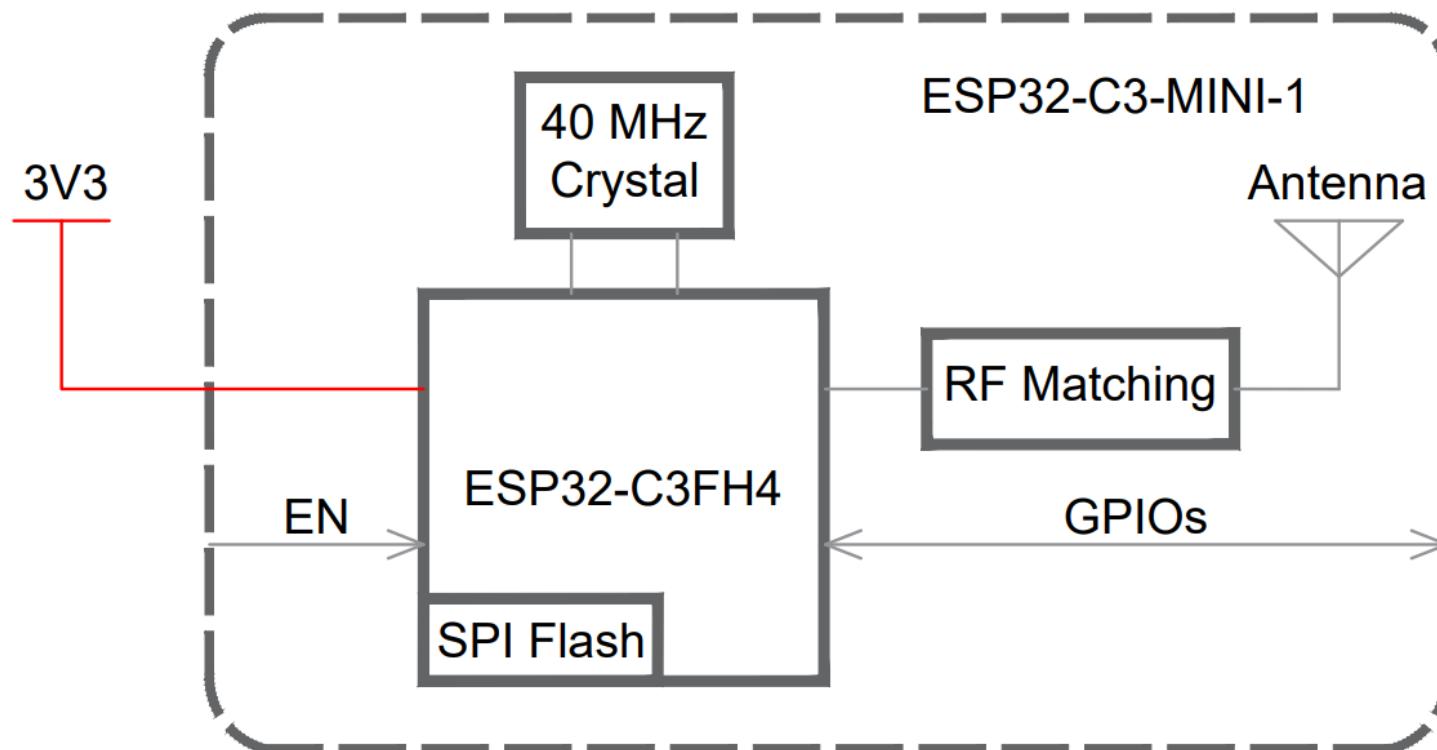
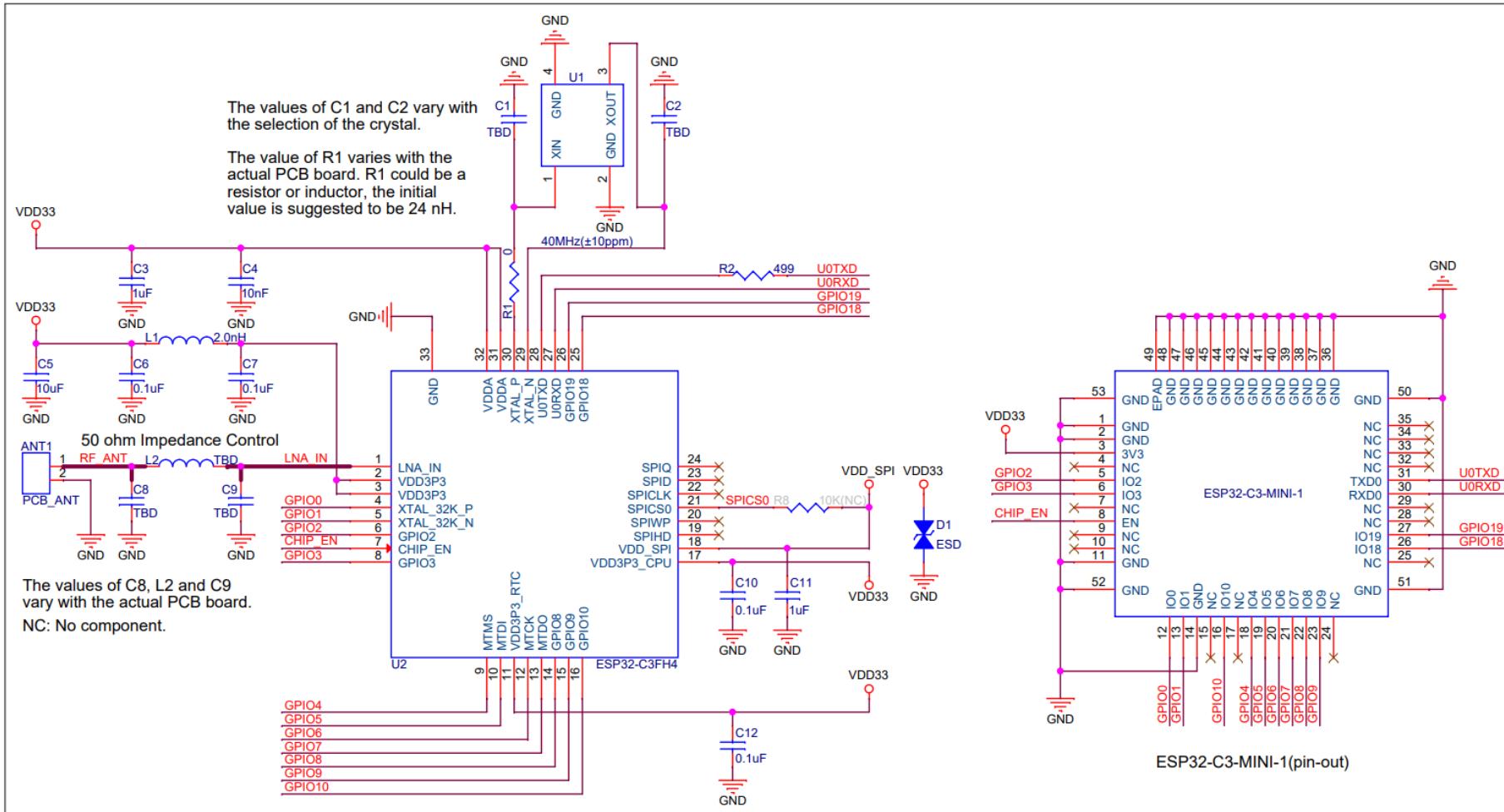
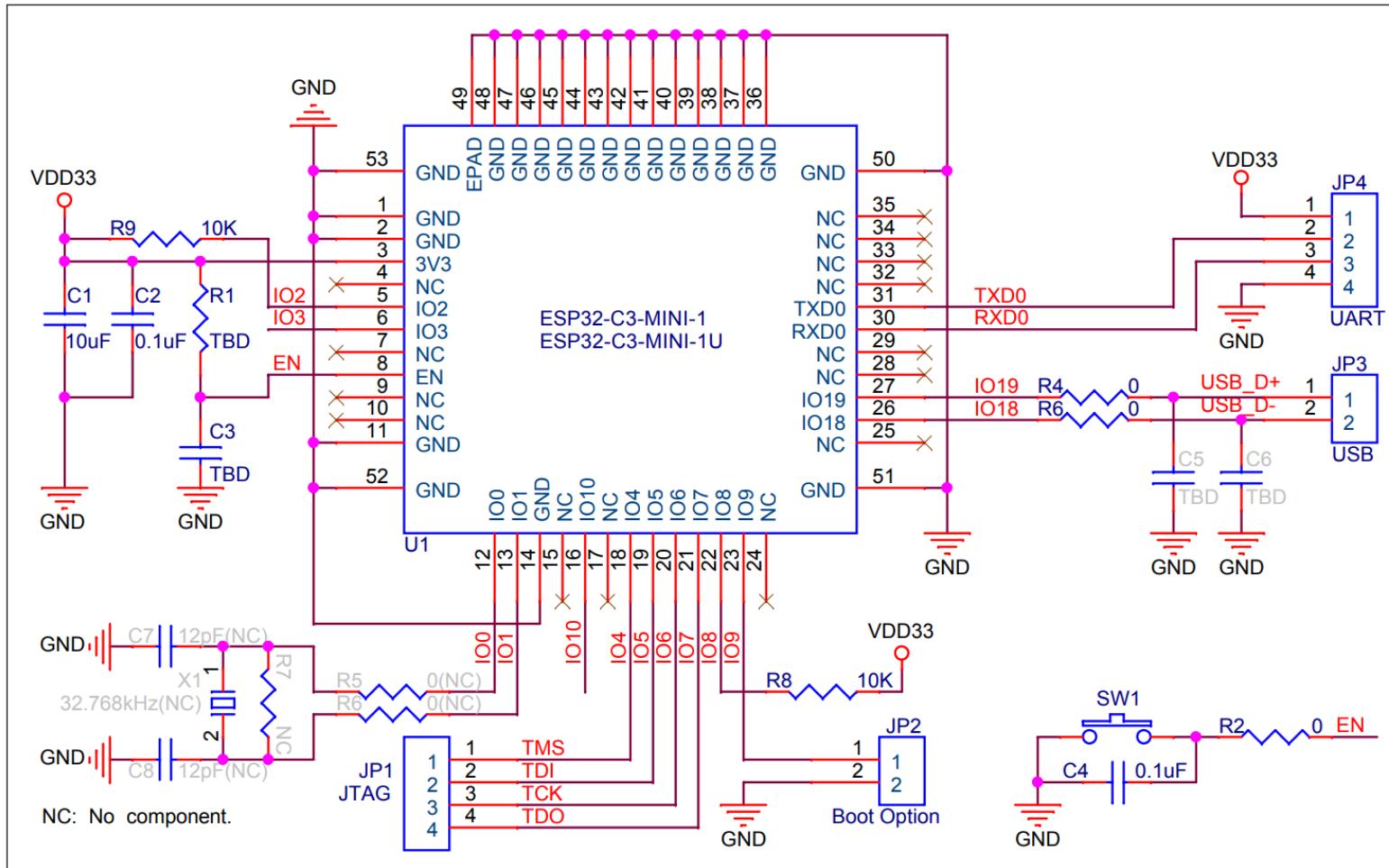


Figure 1: ESP32-C3-MINI-1 Block Diagram

ESP32-C3-MINI-1 - Module schematics (1/2)



ESP32-C3-MINI-1 - Module schematics (2/2)



ESP32-C3 SoC Components

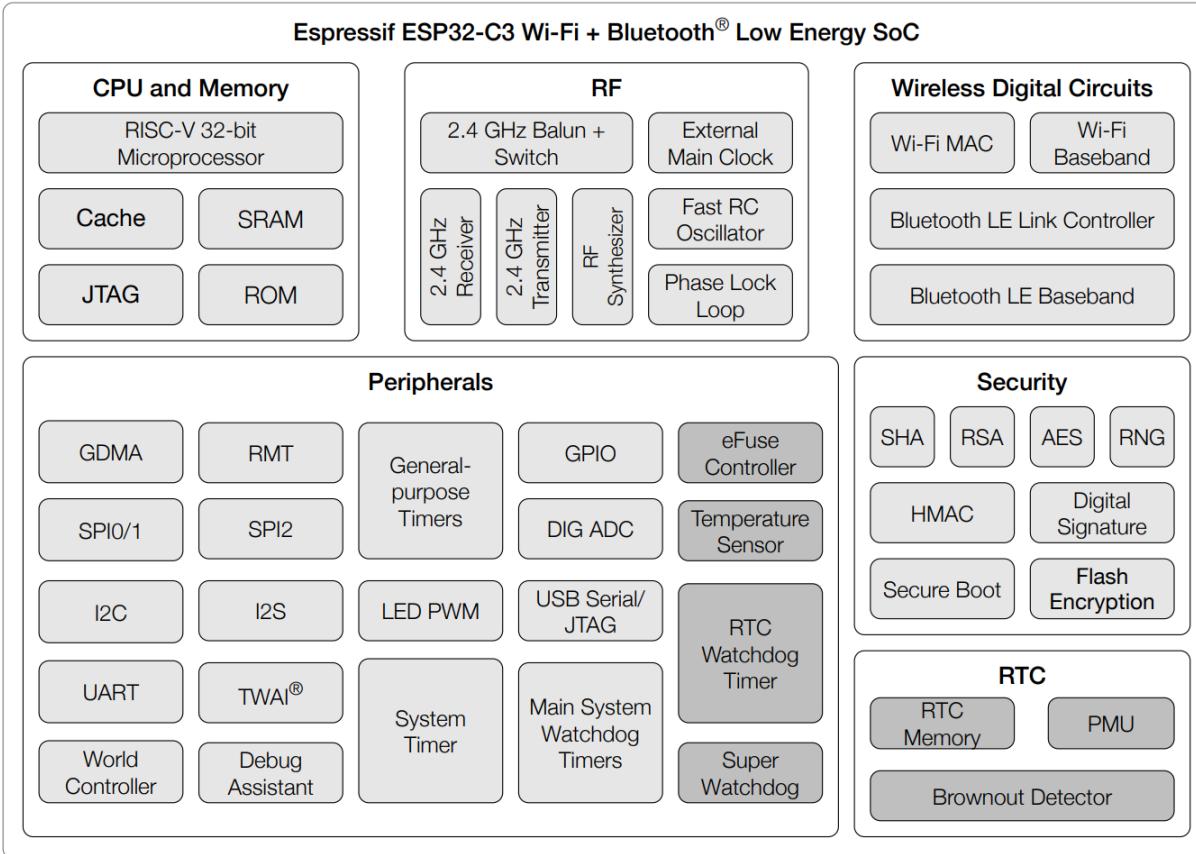


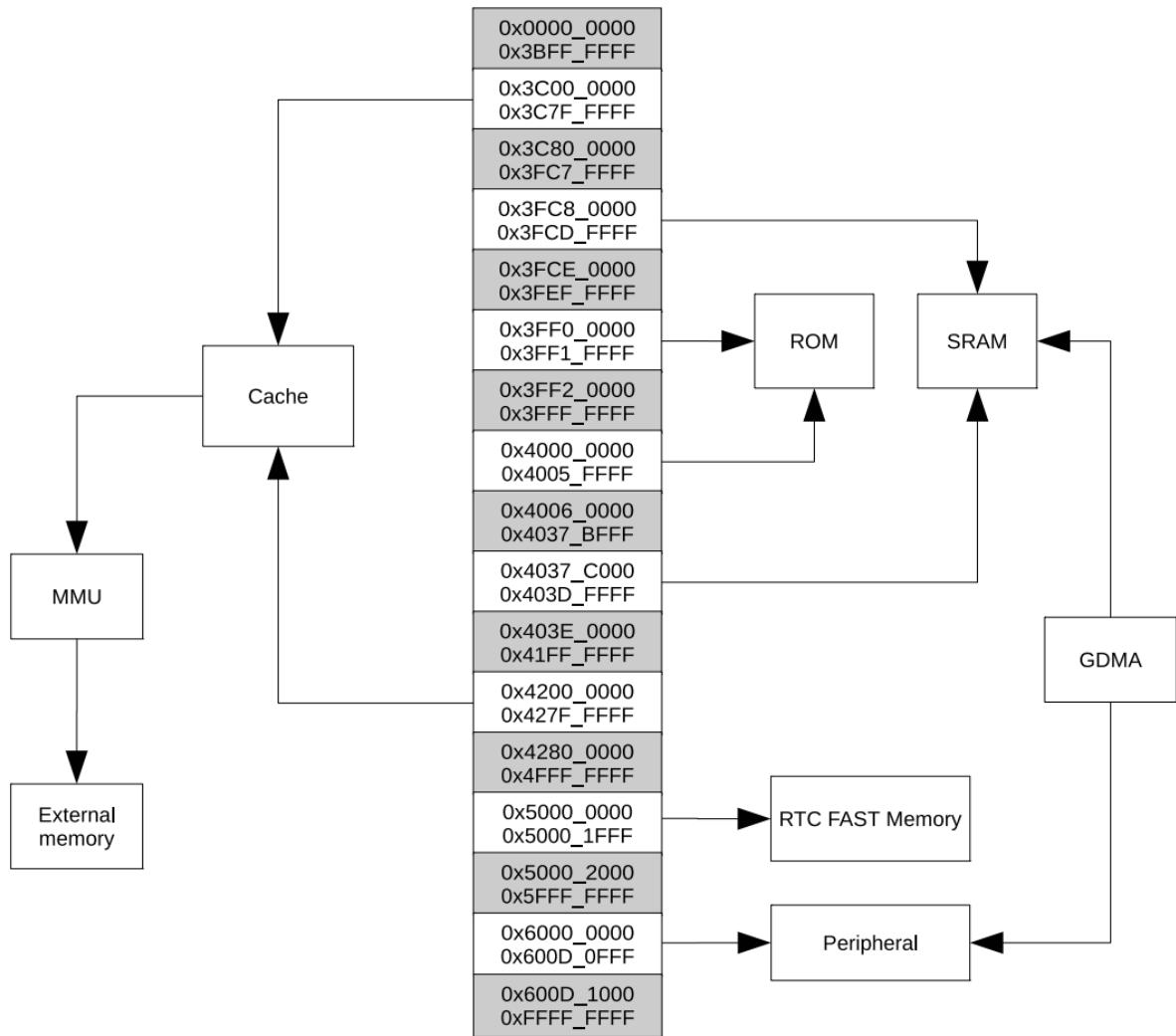
Table 4-1. Components and Power Domains

Power Domain \ Power Mode	RTC	Digital			Optional Digital Periph	Wireless Digital Circuits	Analog			
		CPU					FOSC_CLK	XTAL_CLK	PLL	RF Circuits
Active	ON	ON	ON	ON	ON	ON	ON	ON	ON	ON
Modem-sleep	ON	ON	ON	ON	ON ¹	ON	ON	ON	ON	OFF ²
Light-sleep	ON	ON	OFF ¹	ON ¹	OFF ¹	ON	OFF	OFF	OFF	OFF ²
Deep-sleep	ON	OFF	OFF	OFF	OFF	ON	OFF	OFF	OFF	OFF

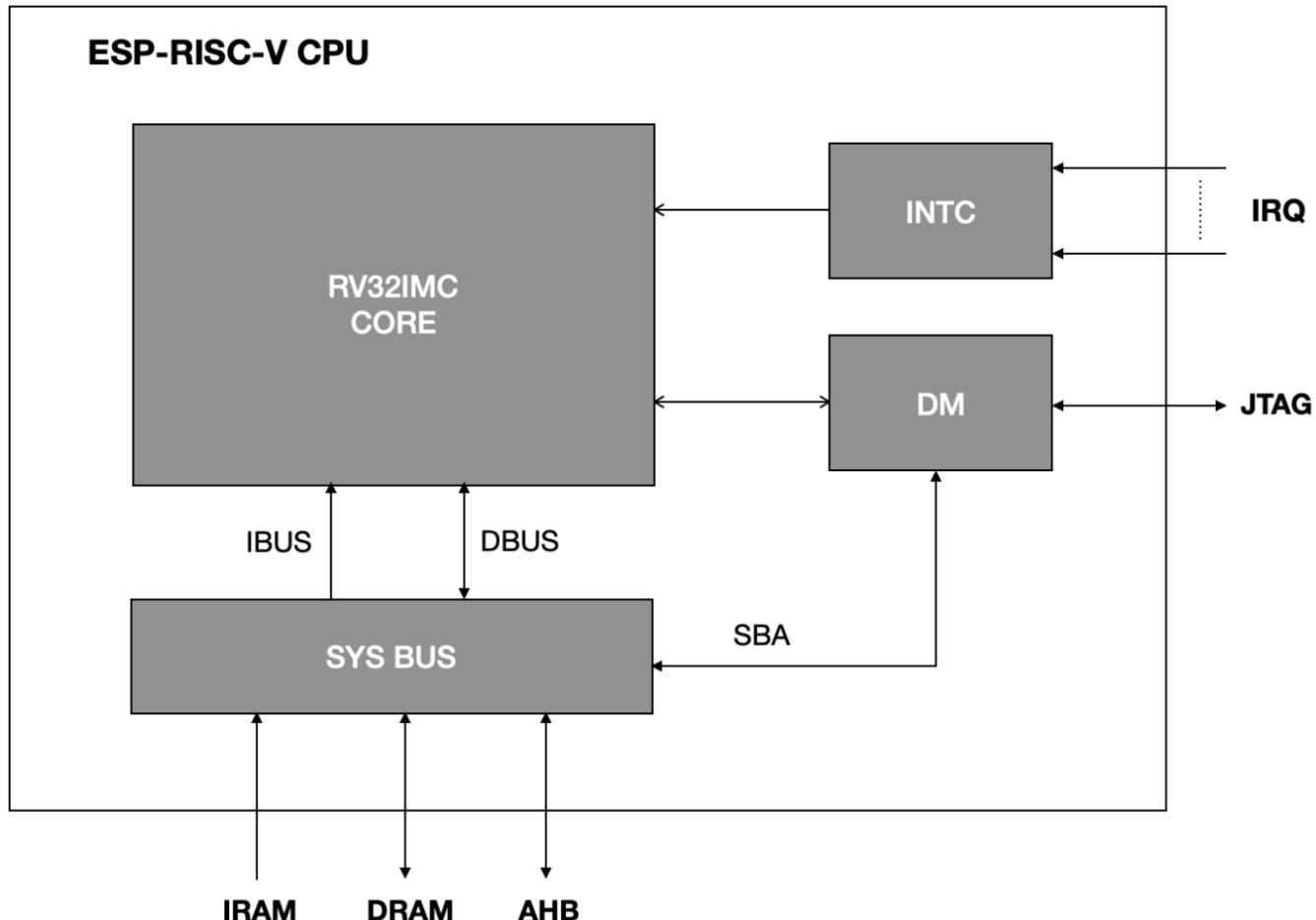
¹ Configurable, see the TRM.

² If Wireless Digital Circuits are on, RF circuits are periodically switched on when required by internal operation to keep active wireless connections running.

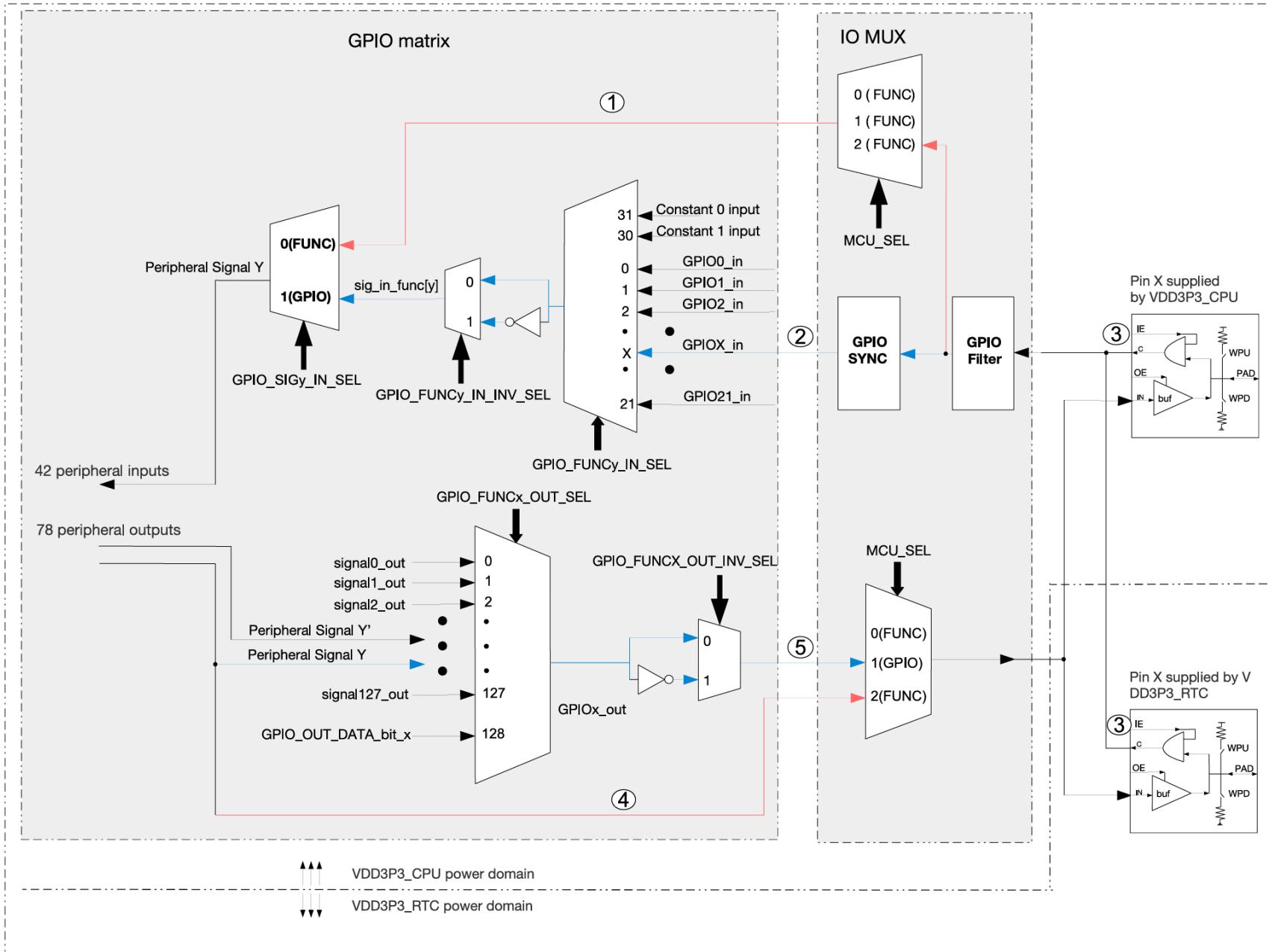
ESP32-C3 Memory Address Space



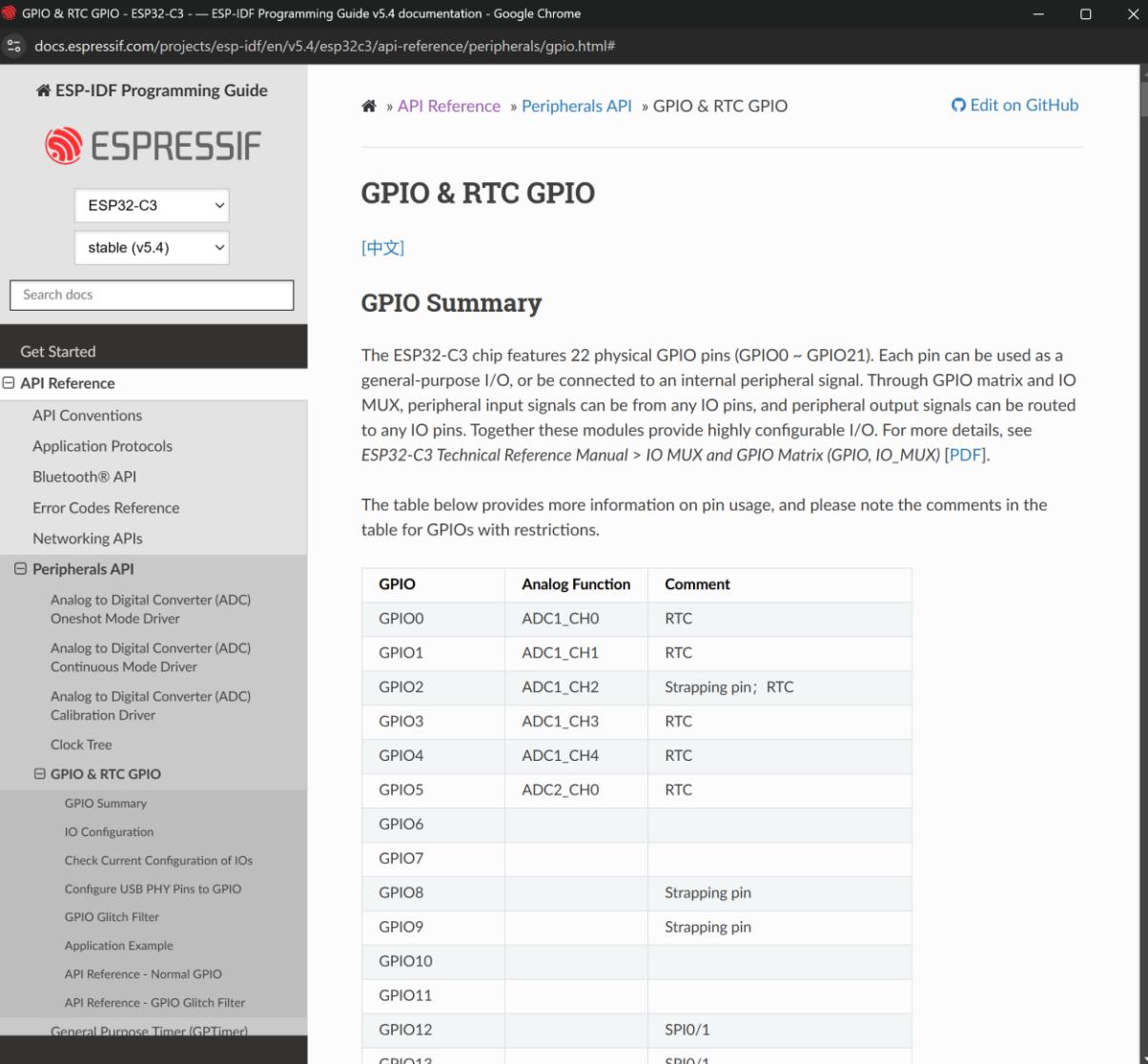
ESP32-C3 RISC-V CPU core



ESP32 I/O Pin Structure



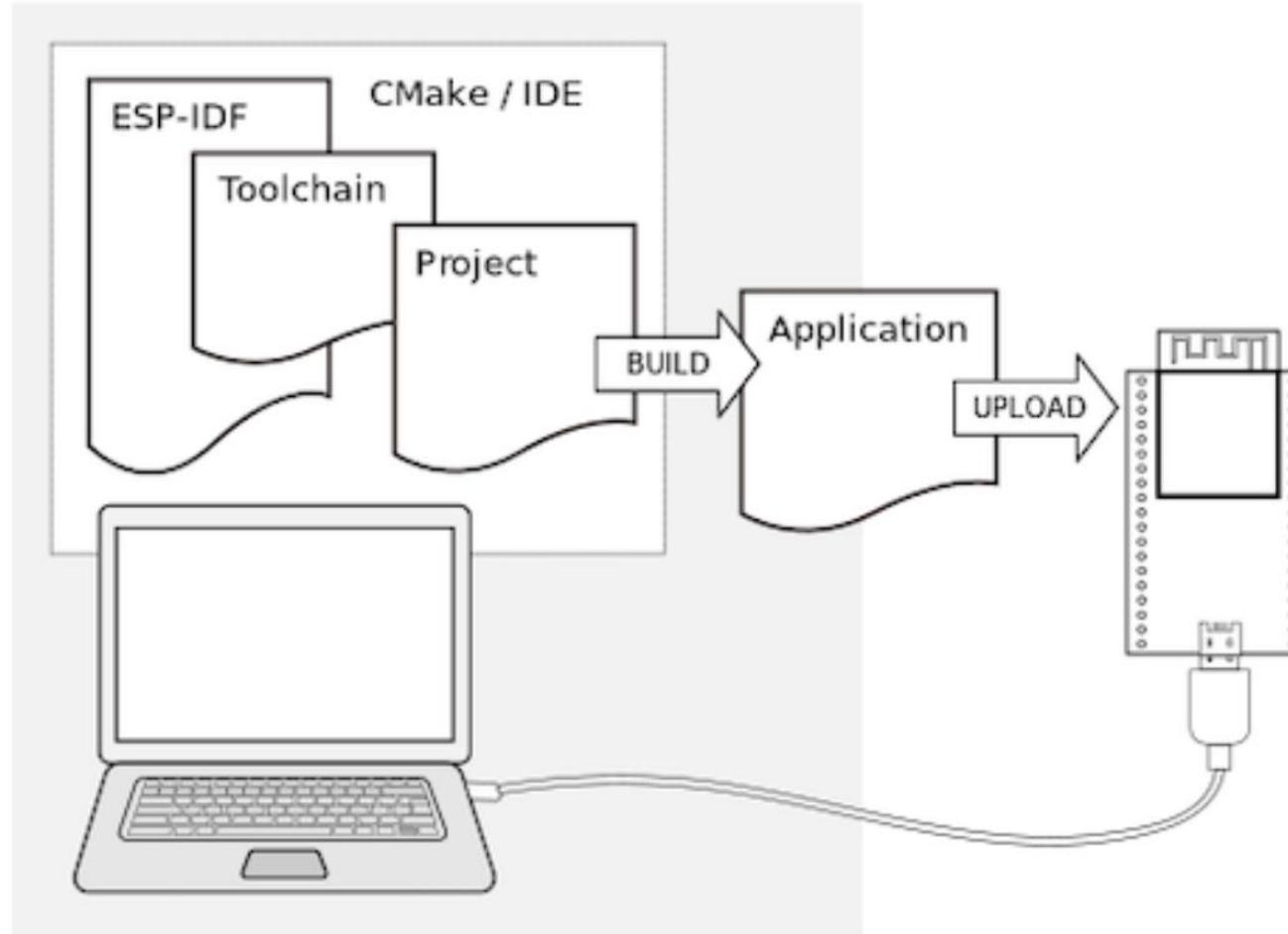
API Reference – General Purpose I/O



The screenshot shows a browser window displaying the [ESP-IDF Programming Guide v5.4 documentation](https://docs.espressif.com/projects/esp-idf/en/v5.4/esp32c3/api-reference/peripherals/gpio.html#). The page title is "GPIO & RTC GPIO". The left sidebar shows the navigation menu with "API Reference" expanded, and "GPIO & RTC GPIO" selected. The main content area contains a "GPIO Summary" section explaining the chip's 22 physical GPIO pins and their routing capabilities. Below this is a table mapping GPIO pins to their analog functions and comments.

GPIO	Analog Function	Comment
GPIO0	ADC1_CH0	RTC
GPIO1	ADC1_CH1	RTC
GPIO2	ADC1_CH2	Strapping pin; RTC
GPIO3	ADC1_CH3	RTC
GPIO4	ADC1_CH4	RTC
GPIO5	ADC2_CH0	RTC
GPIO6		
GPIO7		
GPIO8		Strapping pin
GPIO9		Strapping pin
GPIO10		
GPIO11		
GPIO12		SPI0/1
GPIO13		SPI0/1

ESP32-C3 Compilation Flow



ESP-IDF (command line tools)

Mostly used commands

idf.py set-target esp32c3

idf.py menuconfig

idf.py build

idf.py -p PORT flash

idf.py -p PORT flash monitor



The screenshot shows a Windows PowerShell window titled "ESP-IDF 5.4 PowerShell". The command entered is "idf.py". The output displays the usage information and a detailed list of available commands and their descriptions. The commands are grouped into sections such as Options, Commands, and Secure. The Options section includes flags like --version, --list-targets, --project-dir, --build-dir, --cmake-warn-uninitialized, -v, --verbose, --preview, --ccache, -G, --generator, -D, --define-cache-entry, -P, --port, -b, --baud, and --help. The Commands section lists numerous commands like add-dependency, all, app, app-flash, bootloader, bootloader-flash, build-system-targets, clang-check, clang-html-report, clean, conserver, coredump-debug, coredump-info, create-component, create-manifest, create-project, create-project-from-example, docs, efuse-burn, efuse-burn-key, efuse-common-table, efuse-custom-table, efuse-disable, efuse-read-protect, efuse-summary, efuse-write-protect, encrypted-app-flash, encrypted-flash, erase-flash, erase-otadata, flash, fullclean, gdb, gdbgui, gdbtui, menuconfig, merge-bin, mod-sig, opencoc, partition-table, partition-table-flash, post-debug, python-clean, qemu, read-otadata, reconfigure, save-defconfig, secure-decrypt-flash-data, secure-digest-secure-bootloader, secure-encrypt-flash-data, secure-generate-flash-encryption-key, secure-generate-signing-key, secure-sign-data, secure-verify-signature, set-target, show-efuse-table, size, size-components, size-files, uf2, uf2-apps, update-dependencies, and update-project. The Secure section contains commands for generating private keys and signing boot images.

```
PS C:\Espressif\frameworks\esp-idf-v5.4> idf.py
Usage: idf.py [OPTIONS] COMMAND1 [ARGS]... [COMMAND2 [ARGS]...]...
ESP-IDF CLT build management tool. For commands that are not known to idf.py an attempt to execute it as a build system target will be made. Selected target: None

Options:
  --version                         Show IDF version and exit.
  --list-targets                    Print list of supported targets and exit.
  -C, --project-dir PATH           Project directory.
  -B, --build-dir PATH             Build directory.
  -W, --cmake-warn-uninitialized  Enable CMake uninitialized variable warnings for CMake files inside the project directory. (--no-warning is now the default, and doesn't need to be specified.) The default value can be set with the IDF_CMAKE_WARN_UNINITIALIZED environment variable.
  -n, --no-warnings                Disable CMake uninitialized variable warnings.
  -v, --verbose                     Verbose build output.
  --preview                        Enable IDF features that are still in preview.
  --ccache / --no-ccache           Use ccache in build. Disabled by default. The default value can be set with the IDF_CCACHE_ENABLE environment variable.
  -G, --generator [Ninja]          CMakel generator.
  --no-hints                       Disable hints on how to resolve errors and logging.
  -D, --define-cache-entry TEXT   Create a cmake cache entry. This option can be used at most once either globally, or for one subcommand.
  -P, --port PATH                  Serial port. The default value can be set with the ESPPORT environment variable. This option can be used at most once either globally, or for one subcommand.
  -b, --baud INTEGER              Global baud rate for all idf.py subcommands if they don't overwrite it locally. It can imply monitor baud rate as well if it hasn't been defined locally. The default value can be set with the ESPBAUD environment variable. This option can be used at most once either globally, or for one subcommand.
  --help                           Show this message and exit.

Commands:
  add-dependency                   Add dependency to the manifest file.
  all                            Aliases: build. Build the project.
  app                            Build only the app.
  app-flash                       Flash the app only.
  bootloader                      Build only bootloader.
  bootloader-flash                Flash bootloader only.
  build-system-targets            Print list of build system targets.
  clang-check                      run clang-tidy check under current folder, write the output into "warnings.txt"
  clang-html-report               generate html report to "html_report" folder by reading "warnings.txt" (may take a few minutes). This feature requires extra dependency "codereport". Please install this by running "pip install codereport"
  clean                           Delete build artifacts in the build directory.
  conserver                       Run JSON configuration server.
  coredump-debug                  Create core dump ELF file and run GDB debug session with this file.
  coredump-info                   Print crashed task's registers, callstack, list of available tasks in the system, memory regions and contents of memory stored in core dump (TCBs and stacks)
  create-component                Create a new component.
  create-manifest                 Create manifest for specified component.
  create-project                  Create a new project.
  create-project-from-example    Create a project from an example in the ESP Component Registry.
  docs                            Open web browser with documentation for ESP-IDF
  efuse-burn                      Burn the efuse with the specified name.
  efuse-burn-key                 Burn a 256-bit key to EFUSE: BLOCK1, flash_encryption, BLOCK2, secure_boot_v1, secure_boot_v2, BLOCK3.
  efuse-common-table              Generate C-source for IDF's efuse fields.
  efuse-custom-table              Generate C-source for user's efuse fields.
  efuse-disable                   Disable writing to all efuses.
  efuse-read-protect              Disable writing to the efuse with the specified name.
  efuse-summary                  Get the summary of the efuses.
  efuse-write-protect             Disable writing to the efuse with the specified name.
  encrypted-app-flash            Flash the encrypted app only.
  encrypted-flash                Flash the encrypted project.
  erase-flash                     Erase entire flash chip.
  erase-otadata                  Erase otadata partition.
  flash                           Flash the project.
  fullclean                       Delete the entire build directory contents.
  gdb                            Run the GDB.
  gdbgui                          GDB UI in default browser.
  gdbtui                          GDB TUI mode.
  menuconfig                      Run "menuconfig" project configuration tool.

  Display serial output.
  Run opencoc from current path
  Build only partition table.
  Flash partition table only.
  Utility target to read the output of async debug action and stop them.
  Delete generated Python byte code from the IDF directory
  Run QEMU.
  Read otadata partition.
  Re-run CMake.
  Generate a sdkconfig.defaults with options different from the default ones

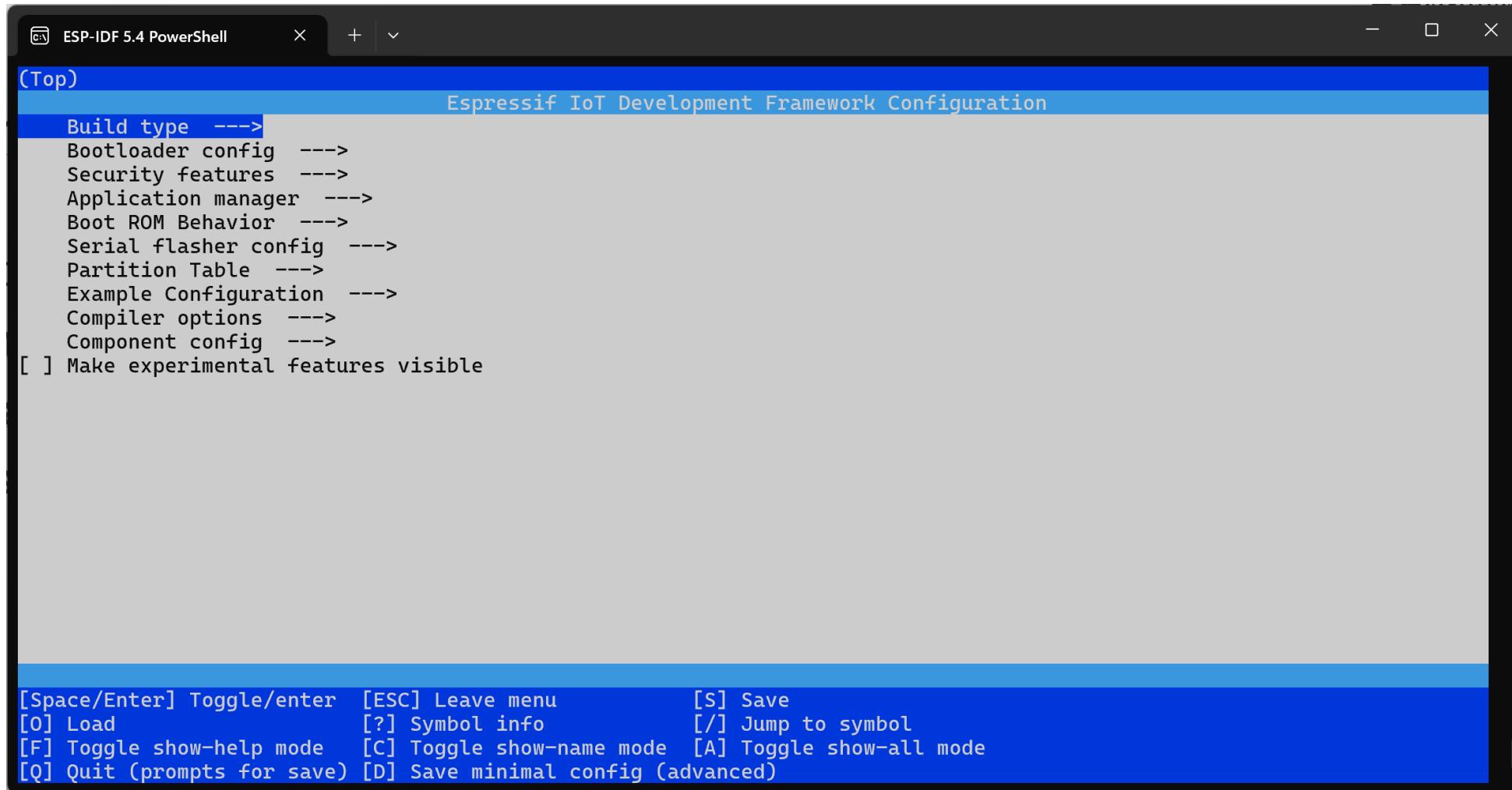
  Take a bootloader binary image and a secure boot key, and output a combined digest+binary suitable for flashing along with the precalculated secure boot key.
  Encrypt some data suitable for encrypted flash (using known key).

  Generate a private key for signing secure boot images as per the secure boot version. Key file is generated in PEMFormat, Secure Boot V1 - ECDSA NIST256 private key. Secure Boot V2 - RSA 3072, ECDSA NIST256p, ECDSA NIST192p private key.
  Sign a data file for use with secure boot. Signing algorithm is deterministic ECDSA w/ SHA-512 (V1) or either RSA-PSS or ECDSA w/ SHA-256 (V2).
  Verify a previously signed binary image, using the ECDSA (V1) or either RSA or ECDSA (V2) public key.

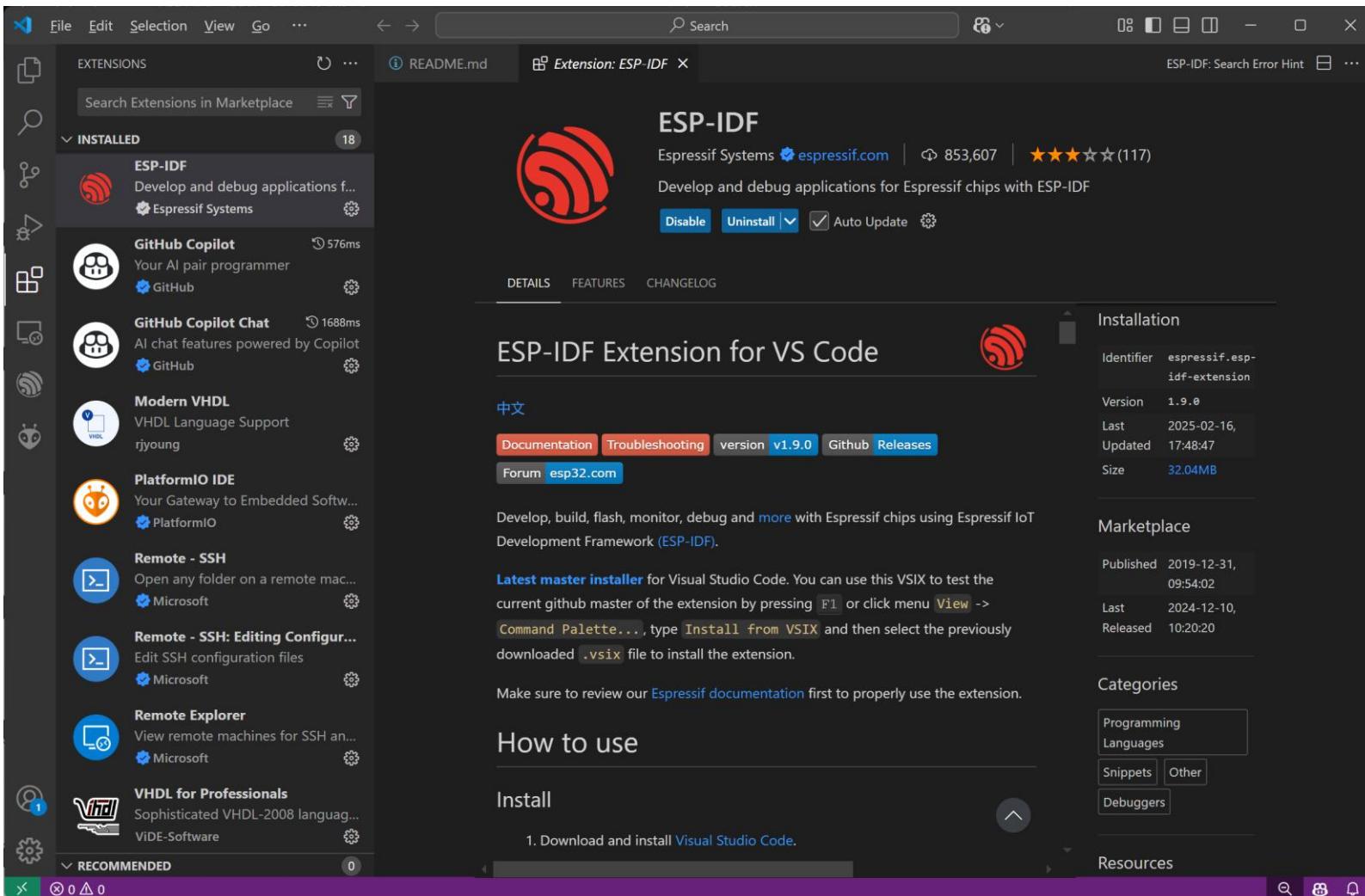
  Set the chip target to build.
  Print efuse table.
  Print basic size information about the app.
  Print per-component size information.
  Print per-source-file size information.
  Generate the UF2 binary with all the binaries included.
  Generate an UF2 binary for the application only.
  Update dependencies of the project

PS C:\Espressif\frameworks\esp-idf-v5.4>
```

ESP-IDF (idf.py menuconfig)



Visual Studio Code - based environment



What is the purpose of the CMake tool? (1/2)

CMake is an open-source tool designed to manage the build process of software in a platform-independent manner. It is primarily used for generating build files that can be used with different native build systems such as **Makefiles**, **Ninja**, **Visual Studio project files**, **Xcode project files**, and others.

Purpose of CMake:

1. Cross-Platform Build Configuration

- CMake allows developers to write build scripts that work across different operating systems (Windows, macOS, Linux) and compilers (GCC, Clang, MSVC).

2. Automatic Build System Generation

- Instead of manually writing complex Makefiles or project files, CMake generates them based on a simple CMake configuration file (`CMakeLists.txt`).

3. Dependency Management

- It can find and configure external libraries, ensuring that all required dependencies are correctly located and linked.

4. Out-of-Source Builds

- Supports building software in a separate directory from the source code, keeping the source clean and preventing conflicts.

5. Integration with IDEs

- Works with popular Integrated Development Environments (IDEs) like Visual Studio, CLion, and Xcode.

6. Build Customization

- Allows defining custom build options and configurations using `CMakeLists.txt` and `CMakeCache.txt`.

7. Support for Multiple Build Types

- Easily configure different build types such as Debug, Release, RelWithDebInfo, and MinSizeRel.

What is the purpose of the CMake tool? (2/2)

Basic Workflow of CMake

1. Write a `CMakeLists.txt` file that defines the project, source files, dependencies, and build settings.
2. Run `cmake` to generate the build system files:

```
css
```

 Copy  Edit

```
cmake -S . -B build
```

3. Use the generated build system to compile the project:

```
css
```

 Copy  Edit

```
cmake --build build
```

CMake is widely used in modern C++ projects due to its flexibility, portability, and ease of integration with various tools and compilers.

What is the purpose of the Ninja tool? (1/2)

Purpose of the Ninja Build System

Ninja is a small, fast build system designed to efficiently handle incremental builds. It is primarily used to speed up the compilation of large projects by minimizing unnecessary work and maximizing parallelism.

Key Features and Purpose of Ninja:

1. Optimized for Speed

- Ninja is significantly faster than traditional build systems like **Make** because it avoids unnecessary file checks and only rebuilds what is required.

2. Incremental Builds

- It tracks dependencies efficiently, ensuring that only modified files are recompiled, reducing build times dramatically.

3. Parallel Execution

- Ninja automatically maximizes CPU usage, making it much faster than `make -j` in large projects.

4. Minimal Overhead

- The Ninja build system is lightweight and has fewer features than Make, focusing only on executing build commands as quickly as possible.

5. Integration with CMake and Other Generators

- Unlike Make, Ninja does not have its own build configuration language. Instead, **CMake**, **GN** (Google's build system), or **Meson** are often used to generate Ninja build files.

6. Cross-Platform Support

- Works on Windows, Linux, and macOS, making it a portable solution for fast builds.

Ninja vs. Other Build Systems

- **Make:** Ninja is significantly faster and better at handling dependencies.
- **CMake:** CMake is not a build system but a **build system generator**. It can generate Ninja files for fast compilation.
- **Bazel/Meson:** Ninja is simpler but does not include higher-level dependency management.

What is the purpose of the Ninja tool? (2/2)

Basic Usage of Ninja

1. Generating Ninja Build Files using CMake

```
css
cmake -G Ninja -S . -B build
```
2. Building the Project with Ninja

```
mathematica
ninja -C build
```

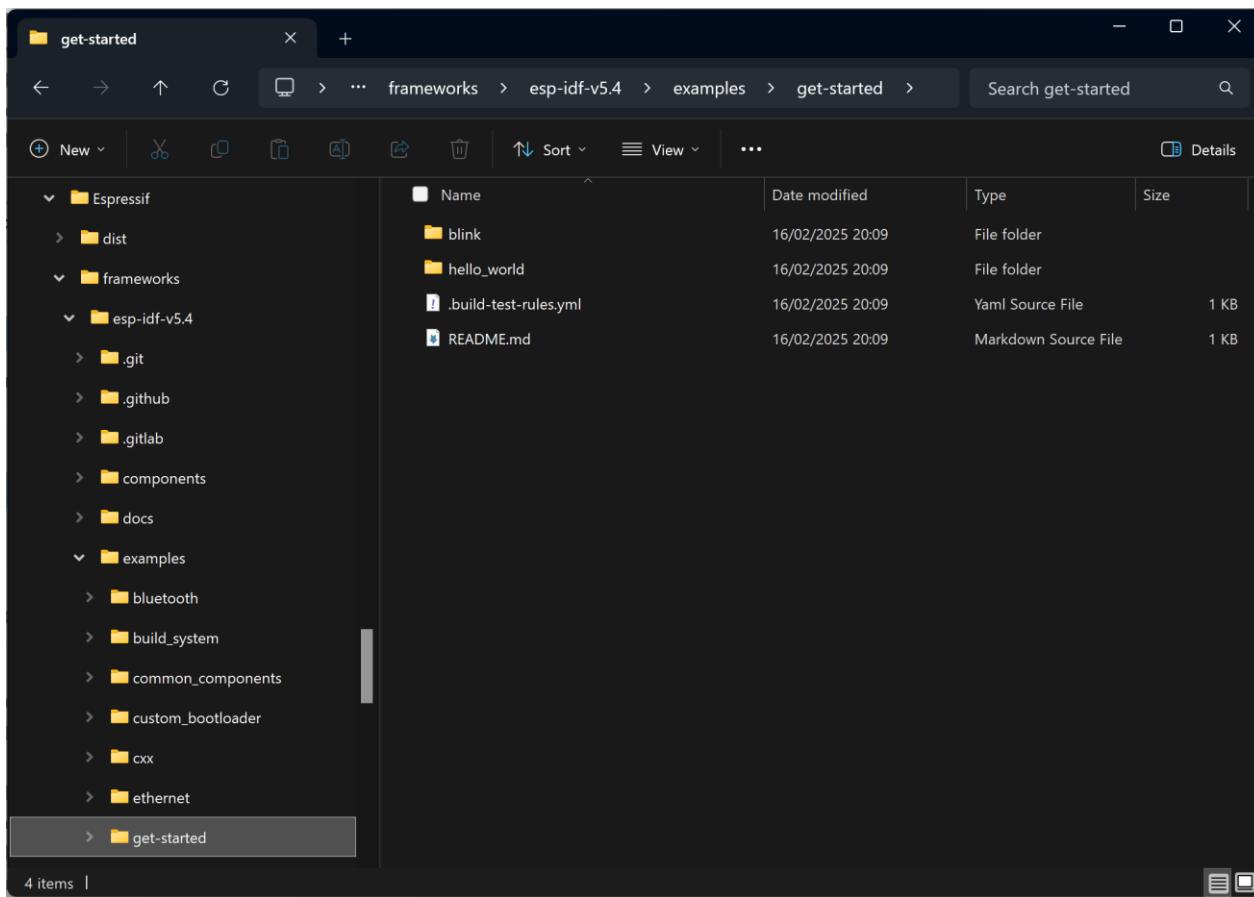
or

```
graphql
cmake --build build -- -j8 # Using CMake with Ninja backend
```

Conclusion

Ninja is widely used in modern development workflows, especially in projects where **fast incremental builds** are essential, such as **Chromium, LLVM, and Android development**. If your project is large and relies on CMake, switching to Ninja can significantly improve build performance. 🚀

Get Started Examples (hello_world and blink)



A screenshot of the 'ESP-IDF 5.4 PowerShell' window. The title bar says '(Top) → Example Configuration'. The main area displays configuration options for the 'blink' example:

```
Blink LED type (LED strip) --->
LED strip backend peripheral (RMT) --->
(8) Blink GPIO number
(1000) Blink period in ms
```

At the bottom, there is a blue footer bar with keyboard shortcuts:

```
[Space/Enter] Toggle/enter [ESC] Leave menu [S] Save
[O] Load [?] Symbol info [/] Jump to symbol
[F] Toggle show-help mode [C] Toggle show-name mode [A] Toggle show-all mode
[Q] Quit (prompts for save) [D] Save minimal config (advanced)
```

Laboratory Assignment 1 – GPIO LED

- Create a copy of the “blink” project to your profile directory
- Connect a LED (+current limiting resistor) in the breadboard to a kit’s GPIO pin of your choice
- Configure the project settings accordingly
- Compile and test the project

Laboratory Assignment 2 – Read in / Write out

- Select a GPIO to be used as an input
- Create a copy of the previous project to your profile directory
- Develop a program that reads the GPIO input and writes the corresponding value to the GPIO output associated with the LED
- Compile and test the project

Laboratory Assignment 3 – Brightness control

- Create a copy of the previous project to your profile directory
- Develop a program that allows to set the brightness of the LED, based on a software-generated PWM signal with a varying duty-cycle controlled by the user
- Compile and test the project

Final Remarks

- You must be acquainted with the previous topics and complete the 3 lab assignments this week
- Always bring with you
 - Breadboard with the kit inserted and connected to the required components
 - USB-A to USB-micro cable
 - Cutting pliers
 - Wires (not jumpers!)
 - Oscilloscope probes (2 un.)