ESP-Docs User Guide



Release v2.1.4-64-ga216ae2e78
Espressif Systems
Nov 19, 2026

Table of contents

Table of contents i

1 Introduction 3
- 1.1 What Is ESP-Docs? . . . 3
  - 1.1.1 Features . . . 3
  - 1.1.2 Third-Party Extensions . . . 3
  - 1.1.3 Extensions Developed by Espressif . . . 4
- 1.2 reStructuredText v.s. Markdown . . . 6
  - 1.2.1 Extensibility . . . 7
  - 1.2.2 Features . . . 7
  - 1.2.3 Building and Deploying Documentation . . . 9
  - 1.2.4 Comparison table . . . 10
- 1.3 ESP-Docs & Espressif Server v.s. Sphinx & Read the Docs . . . 10

2 Writing Documentation 13
- 2.1 Basic Syntax . . . 13
  - 2.1.1 Paragraphs . . . 13
  - 2.1.2 Inline Formatting . . . 13
  - 2.1.3 Titles and Headings . . . 14
  - 2.1.4 Section Numbering . . . 14
  - 2.1.5 Lists . . . 16
  - 2.1.6 Code Blocks . . . 17
  - 2.1.7 Tables of Contents . . . 18
  - 2.1.8 Index Files . . . 19
  - 2.1.9 Substitutions . . . 19
  - 2.1.10 To-Do Notes . . . 20
- 2.2 Figures . . . 20
  - 2.2.1 Why Add Figures? . . . 20
  - 2.2.2 Adding Figures in ESP-Docs . . . 21
- 2.3 Tables . . . 28
  - 2.3.1 Simple Table . . . 28
  - 2.3.2 Grid Table . . . 29
  - 2.3.3 List Table . . . 30
  - 2.3.4 CSV Table . . . 31
  - 2.3.5 Comparison . . . 31
  - 2.3.6 Still No Good Fit? . . . 32
- 2.4 Links . . . 32
  - 2.4.1 Table of Contents . . . 32
  - 2.4.2 Linking to Language Versions . . . 33
  - 2.4.3 Linking to Other Sections Within the Document . . . 33
  - 2.4.4 Linking to Other Documents . . . 33
  - 2.4.5 Linking to a Specific Place of Other Documents in a Same Project . . . 34
  - 2.4.6 Linking to Kconfig References . . . 35
  - 2.4.7 Linking to Classes, Functions, Enumerations, etc . . . 35
  - 2.4.8 Linking to GitHub Files . . . 35
  - 2.4.9 Linking to External Pages . . . 36
  - 2.4.10 Linking to ESP TRMs and Datasheets . . . 36

2.4.11 Resources . . . 37
2.5 Creating a Glossary . . . 37
2.5.1 Create Glossary of Terms . . . 37
2.5.2 Link a Term to its Glossary Entry . . . 38
2.6 Writing for Multiple Targets . . . 39
2.6.1 Target-Specific Inline Text . . . 39
2.6.2 Target-Specific Paragraph . . . 40
2.6.3 Target-Specific Bullet Point . . . 41
2.6.4 Target-Specific Document . . . 41
2.7 Redirecting Documents . . . 42
2.8 Writing API Description . . . 43
2.8.1 Document Conventions . . . 43
2.8.2 Macro . . . 43
2.8.3 Type Definition . . . 44
2.8.4 Enumeration . . . 45
2.8.5 Structure . . . 45
2.8.6 Union . . . 47
2.8.7 Function . . . 48
2.9 Formatting and Generating API Descriptions . . . 51
2.9.1 Document API in Header Files . . . 51
2.9.2 Generate and Include API Descriptions . . . 58
2.9.3 Example . . . 59
2.10 Formatting Documents for Translation . . . 59
2.10.1 One Line per Paragraph . . . 60
2.10.2 Line Number Consistency . . . 60
2.11 Using Spellcheckers in VS Code . . . 61
2.11.1 Why Use Spellcheckers? . . . 61
2.11.2 Using Code Spell Checker in VS Code . . . 61

3 Building Documentation **63**
3.1 Previewing Documentation inside Your Text Editor . . . 63
3.1.1 Visual Studio Code . . . 63
3.1.2 Sublime Text . . . 63
3.2 Building Documentation Locally . . . 63
3.2.1 Building HTML Locally on Your PC . . . 64
3.2.2 Building PDF Documentation Locally on Your PC . . . 66
3.2.3 Using a Docker Container . . . 67
3.2.4 Troubleshooting . . . 68

4 Configuring ESP-Docs Projects **69**
4.1 Integrating ESP-Docs into Your Project . . . 69
4.1.1 Get Familiar with the Documentation Folder . . . 69
4.1.2 Prepare a Documentation Folder . . . 70
4.1.3 Update Build Configuration Files . . . 70
4.1.4 Update CI Configuration File . . . 71
4.1.5 What's Next? . . . 74
4.2 Adding Extensions . . . 74
4.2.1 Where to Add? . . . 74
4.2.2 Third-Party Extensions . . . 74
4.2.3 Self-Developed Extensions . . . 75
4.3 Adding the Link-check Function . . . 75
4.3.1 How to Integrate the Link-check Function . . . 75
4.3.2 Note . . . 77
4.4 Adding the Language Link Checker . . . 77
4.4.1 What the Language Link Checker Verifies . . . 78
4.4.2 How to Integrate the Language Link Checker . . . 78
4.4.3 Understanding the Check Results . . . 79
4.4.4 Excluding Files from Check . . . 79

4.5 Collecting User Analytics . . . 80
4.5.1 Enabling Google Analytics for Your Project . . . 80
4.5.2 Viewing Google Analytics Data or Reports . . . 80
4.6 Collecting Document Feedback . . . 80
4.6.1 How to Add the Feedback Section to Your Repo . . . 81
4.6.2 Configuration of Collection of User Reactions in Google Analytics . . . 81
4.6.3 References . . . 85
4.7 Configuring `codespell` in GitLab Repositories . . . 85
4.7.1 Using `codespell` as a Git Pre-Commit Hook (Local) . . . 85
4.7.2 Expected Behavior of the Local `pre-commit` Hook for `codespell` . . . 86
4.7.3 Implement `codespell` in GitLab CI/CD . . . 86
4.7.4 Summary . . . 89

5 Troubleshooting **91**
5.1 Troubleshooting Build Errors and Warnings . . . 91
5.1.1 Message Format . . . 91
5.1.2 Package-Related Errors and Warnings . . . 91
5.1.3 Syntax-Related Errors and Warnings . . . 93
5.1.4 Still Have Troubles? . . . 95

6 Contributing Guide **97**
6.1 Report a Bug . . . 97
6.2 Add a New Feature . . . 97
6.3 Make Minor Changes . . . 97
6.4 Use the ESP-IDF Documentation Agent Guide . . . 97
6.5 Commit Messages . . . 98
6.6 Ask a Question . . . 98
6.7 Release Process . . . 98

7 Related Resources **101**

8 Glossary **103**

Index **105**

Index **105**

ESP-Docs is a documentation-building system developed by Espressif based on Sphinx and Read the Docs. This guide provides information on how to use it as the documentation-building system in a project and how to write, build, configure, and deploy the documentation under this system.

It is primarily for developers, writers, and translators who work on Espressif software documentation. Others can also use it as a reference, such as for reStructuredText syntax, Sphinx extensions, and customizing your documentation-building system based on Sphinx. Note that some links in this guide point to Espressif's internal documentation, which is thus not accessible to external users.

The guide consists of the following major sections:



Introduction

Overview, features, extensions, supported markup language of ESP-Docs.



Writing Doc

ESP-Docs-specific syntax and generic Sphinx and restructuredText syntax, including *basic syntax* and *link syntax*.



Building Doc

How to preview documentation, build documentation from source to target (HTML, PDF) etc.



Configuring ESP-Docs Projects

Configuration to Git projects to use ESP-Docs, adding extensions, etc.

Chapter 1

Introduction

1.1 What Is ESP-Docs?

ESP-Docs is a documentation-building system developed by Espressif based on Sphinx and Read the Docs. It expands Sphinx functionality and extensions with the features needed for Espressif's documentation and bundles this into a single package. It takes text source files written in reStructuredText and builds them into target formats, including HTML and PDF.

ESP-Docs is an open-source and common project. You are always welcome to contribute any functionality! See *Contributing Guide* for more information.

ESP-Docs is available as a Python package.

1.1.1 Features

ESP-Docs has the following features:

- Generating documentation for multiple targets from the same source files
- Generating API documentation automatically for multiple targets from header files
- Page redirection
- Linking to a specific file and folder in the project
- All features already provided by Sphinx, such as:
  - Source text format: reStructuredText
  - Multiple languages: English, Chinese, etc.
  - Output format: HTML, PDF, etc.
  - Extensive cross-references
  - Extensions

1.1.2 Third-Party Extensions

Besides Sphinx, several other third-party applications (extensions) help to provide nicely formatted and easy-to-navigate documentation. These applications are listed together with the installed version numbers as the dependent packages to ESP-Docs in setup.cfg .

- docutils: open-source text processing system for processing plaintext in reStructuredText into HTML, LaTeX, etc.
- cairosvg: SVG converter based on Cairo 2D graphics library to export SVG files to PDF, EPS, PS, and PNG files.
- sphinx: documentation generator, which is the foundation for ESP-Docs.
- breathe: bridge between the Sphinx and Doxygen documentation systems, making it possible to include Doxygen information in a set of documentation generated by Sphinx.
- sphinx-copybutton: Sphinx extension to add a "copy" button to code blocks.

- sphinx-notfound-page: Sphinx extension to create custom 404 pages.
- sphinxcontrib-blockdiag: Sphinx extension to generate block diagrams from plaintext.
- sphinxcontrib-seqdiag: Sphinx extension to generate sequence diagrams from plaintext.
- sphinxcontrib-actdiag: Sphinx extension to generate activity diagrams from plaintext.
- sphinxcontrib-nwdiag: Sphinx extension to generate network-related diagrams from plaintext.
- sphinxcontrib-wavedrom: Sphinx extension to generate wavedrom diagrams from plaintext.
- sphinxcontrib-svg2pdfconverter: sphinx extension to convert SVG images to PDF in case the builder does not support SVG images natively.
- nwdiag: network diagram generator.
- recommonmark: a flavor of Markdown. With this package, Sphinx can build documents written in Markdown to target formats.
- sphinx_selective_exclude: Sphinx extension to make the "only::" directive provided by Sphinx work in an expected and intuitive manner.

1.1.3 Extensions Developed by Espressif

Espressif has created a couple of custom add-ons and extensions to help integrate documentation with underlying Espressif repositories and further improve navigation as well as maintenance of documentation.

The section provides a quick reference to these add-ons and extensions.

Generic Extensions

These Sphinx extensions are developed for Espressif but do not rely on any Espressif-docs-specific behavior or configuration.

Toctree Filter This Sphinx extension overrides the `:toctree:` directive to allow filtering entries based on whether a tag is set (similar to how `.. only::` does for paragraphs), as `:tagname: toctree_entry`. See the Python file for a more complete description.

See *Target-Specific Document* for an example.

List Filter This Sphinx extension provides a `.. list::` directive that allows filtering of entries in lists based on whether a tag is set, as `:tagname: - list content`. See the Python file for a more complete description.

See *Target-Specific Bullet Point* for an example.

HTML redirect During the documentation lifetime, some source files are moved between folders or renamed. This Sphinx extension adds a mechanism to redirect documentation pages that have changed URLs by generating in the Sphinx output static HTML redirect pages. The script is used together with a redirection list `html_redirect_pages`. `conf_common.py` builds this list from `docs/page_redirects.txt`.

See *Redirecting Documents* for how to redirect documents.

Add warnings In some cases, it might be useful to be able to add warnings to a list of documents. This is the case in ESP-IDF when we introduce a new target, which we build docs for, but not all docs are yet updated with useful information. This extension can then be used to give warnings to readers of documents that are not yet updated.

Configuration values:

- `add_warnings_content`: content of the warning which will be added to the top of the documents.
- `add_warnings_pages`: list of the documents which the warning will be added to.

See conf_commom.py and docs_not_updated of ESP-IDF Programming Guide for an example.

Espressif-Specific Extensions

Run Doxygen Subscribes to `defines-generated` event and runs Doxygen (`docs/doxygen/Doxyfile`) to generate XML files describing key headers, and then runs Breathe to convert these to `.inc` files which can be included directly into API reference pages.

Pushes a number of target-specific custom environment variables into Doxygen, including all macros defined in the project's default `sdkconfig.h` file and all macros defined in all `soc` component `xxx_caps.h` headers. This means that public API headers can depend on target-specific configuration options or `soc` capabilities headers options as `#ifdef` & `#if` preprocessor selections in the header.

This means we can generate different Doxygen files, depending on the target we are building docs for.

For headers with unique names the path to the generated `.inc` will be the header name itself, e.g.: `inc/my_header.inc`, while for headers with non-unique names the whole header path will be used, e.g.: `inc/component/folder/my_header.inc`.

See *Formatting and Generating API Descriptions* for how to generate API description from header files and include it in your documentation.

Exclude Docs The Sphinx extension updates the excluded documents according to the `conditional_include_dict {tag:documents}`. If the tag is set, the list of documents will be included.

It is also responsible for excluding documents when building with the config value `docs_to_build` set. In these cases, all documents not listed in `docs_to_build` will be excluded.

It subscribes to `defines-generated` as it relies on the Sphinx tags to determine which documents to exclude.

See *Target-Specific Document* for an example.

Format ESP Target This is an extension for replacing generic target-related names with the idf_target passed to the Sphinx command line. It supports markup for defining local (single .rst file) substitutions and it also overrides the default `.. include::` directive in order to format any included content using the same rules.

See *Target-Specific Inline Text* for an example.

Link Roles This is an implementation of a custom Sphinx Roles to help to link from documentation to specific files and folders in project repositories.

See *Links to files on GitHub* for an example.

Latex Builder This extension adds ESP-Docs-specific functionality to the LaTeX builder. It overrides the default Sphinx LaTeX builder.

It creates and adds the espidf.sty LaTeX package to the output directory, which contains some macros for run-time variables such as IDF-Target.

Include Build File The `include-build-file` directive is like the built-in `include-file` directive, but the file path is evaluated relative to `build_dir`.

IDF-Specific Extensions

Build System Integration This is a Python package implementing a Sphinx extension to pull IDF build system information into the documentation build process:

- Creates a dummy CMake IDF project and runs CMake to generate metadata.
- Registers some new configuration variables and emits a new Sphinx event, both of which are for use by other extensions.

Configuration Variables

- `docs_root` - The absolute path of the $IDF_PATH/docs directory.
- `idf_path` - The value of IDF_PATH variable, or the absolute path of IDF_PATH if environment unset.
- `build_dir` - The build directory passed in by `build_docs.py`, and the default will be like `_build/<lang>/<target>`.
- `idf_target` - The IDF_TARGET value. It is expected that `build_docs.py` set this on the Sphinx command line.

New Event `project-build-info` event is emitted early in the build, after the dummy project CMake run is complete.

Arguments are `(app, project_description)`, where `project_description` is a dict containing the values parsed from `project_description.json` in the CMake build directory.

Other IDF-specific extensions subscribe to this event and use it to set up some docs parameters based on build system info.

KConfig Reference This extension subscribes to `project-build-info` event and uses confgen to generate `kconfig.inc` from the components included in the default project build. This file is then included into `/api-reference/kconfig`.

See *Link to Kconfig Reference* for an example.

Error to Name Small wrapper extension that calls `gen_esp_err_to_name.py` and updates the included .rst file if it has changed.

Generate Toolchain Links There are a couple of places in documentation that provide links to download the toolchain. To provide one source of this information and reduce efforts to manually update several files, this script generates toolchain download links and toolchain unpacking code snippets based on information found in `tools/toolchain_versions.mk`. These links can be found in List of IDF Tools.

Generate Version-Specific Includes This extension automatically generates reStructuredText `.inc` snippets with version-based content for this ESP-IDF version, such as `git-clone-bash.inc`.

Generate Defines This extension integrates defines from IDF into the Sphinx build and runs after the IDF dummy project has been built.

It parses defines and adds them as Sphinx tags.

It emits the new `defines-generated` event which has a dictionary of raw text define values that other extensions can use to generate relevant data.

Sphinx-IDF-Theme

HTML/CSS theme for Sphinx based on ReadtheDocs' s Sphinx theme. For more information see the Sphinx-IDF-theme repository.

1.2 reStructuredText v.s. Markdown

reStructuredText and Markdown are two markup languages that are easy to read in plain-text format. Comparatively, Markdown is simpler than reStructuredText regarding syntax, formatting, and documentation build system, so many startup project documentation would use Markdown for its simplicity.

If your project is small, with a limited number of documents (for example, less than 5) and subfolders, then Markdown is your go-to language.

As your project evolves and becomes more systematic, you might consider switching to reStructuredText which ESP-Docs uses, given that reStructuredText offers more advanced formatting features and better experience but requires fewer manual edits.

This document compares reStructuredText and Markdown in the following aspects, so that you can better understand why reStructuredText is more suitable for complex projects.

- *Extensibility*
- *Features*
  - *API Reference*
  - *Tables*
  - *Links*
  - *Table of Contents*
- *Building and Deploying Documentation*
  - *Build Packages for reStructuredText*
  - *Build Packages for Markdown*
- *Comparison table*

1.2.1 Extensibility

Extensibility is a core design principle for reStructuredText. For this markup language, it is straightforward to add:

- Customized `roles` and `directives`, such as `:example:` defined in link_roles.py
- Extensions developed by others, such as `sphinxcontrib.blockdiag`
- Extensions developed by yourself, such as format_esp_target.py (see *Adding Extensions*)

In Markdown, there is no such built-in support for extensions, and people might use different extensions in their Markdown editors to do the same thing. For example, to draw a diagram in the same project, one might use UMLet in VS Code, others might use UmlSync in MacDown.

Because reStructuredText can be more easily extended, it has more features provided by various extensions as described in the following section.

1.2.2 Features

reStructuredText has more built-in and extended features for generating API reference, tables, links, and table of contents. These features can save your time to do manual edits, and make complex documents fancier.

API Reference

In reStructuredText, you can include API references generated from header files into your documentation (see *Formatting and Generating API Descriptions*). The generation process of API references can be integrated into the build process. For example, ESP-Docs has an extension called run_doxygen.py to generate API references from header files when building documentation. You may navigate to doxygen, and run `build_example.sh` to see the results.

In Markdown, generating API documentation is not that easy. You need to either write from scratch as shown below, or leverage some third-party API generators.

````
### *check_model* method

```
Calibrator.check_model(model_proto)
```
Checks the compatibility of your model.
````

(continues on next page)

(continued from previous page)

```
**Argument**
- **model_proto** _(ModelProto)_: An FP32 ONNX model.

**Return**
- **-1**: The model is incompatible.
```

Tables

Thanks to the various *table formats* supported by reStructuredText, you can create more complex tables with merged cells, bullet lists, and specified column width, etc.

Column 1	Column 2
• Bullet point 1 • Bullet point 2	Column 2 is set to be wider

Column 1	Column 2
Merged cell	

In Markdown, you can only adjust table alignment.

Links

In reStructuredText, there are many ways to avoid using raw URL links (see *Links*) when you:

- Link to a specific place of other documents in the same project
- Link to other documents in the same project without specifying document name

With ESP-Docs, you can even extend this functionality when you:

- Link to Kconfig references
- Link to classes, functions, enumerations, etc.
- Link to GitHub files of a certain commit

One advantage of using above link syntax is to avoid manual update when links change.

None of these features are supported in Markdown.

Table of Contents

In reStructuredText, you can use the `toctree` directive to generate a Table of Contents at a specified folder depth. Using a file path is sufficient, and when document headings change, the headings in toctree will be updated automatically.

```
.. toctree::
    :maxdepth: 2

    release-5.x/5.0/index
    release-5.x/5.1/index
```

Moreover, with the help of `toctree`, you can generate a sidebar that contains the table of contents for easy navigation. For example, see the sidebar of ESP-Docs User Guide.

In Markdown, inserting a table of contents with the same effect is also possible, but you need to manually insert each file's path and name, and specify folder structure when including more than one folder levels.

```
- [Migration from 4.4 to 5.0](./release-5.x/5.0/index)
    - [Bluetooth](./release-5.x/5.0/bluetooth)
    - [Wi-Fi](./release-5.x/5.0/wifi)
    - [Peripherals](./release-5.x/5.0/peripherals)
- [Migration from 5.0 to 5.1](./release-5.x/5.1/index)
    - [Peripherals](./release-5.x/5.0/peripherals)
```

Besides, in Markdown there is no sidebar to show the documents in this project and to help readers navigate. Take the ESP-DL repository as example. If you are reading Get Started, and want to check how to deploy a model, there is no way to know where to find this document until you explore almost every folder. Just imagine what a nightmare it would be if the project has 100 files.

1.2.3 Building and Deploying Documentation

Regardless of which source format you choose, both reStructuredText and Markdown rely on a static site generator (SSG) to convert source files into HTML/CSS/JavaScript. An SSG is a build tool that processes your markup files and generates a static website that can be deployed to various hosting platforms.

While the core build-and-deploy logic is similar for both formats, the specific packages, features, and processes vary depending on which toolchain you choose. Additionally, **access control (private/public)** is primarily determined by the hosting solution rather than the source format or build toolchain.

Build Packages for reStructuredText

For `.rst` sources in Espressif projects, the commonly used build packages are:

- **Sphinx**: A general-purpose documentation generator widely used in the Python ecosystem.
- **ESP-Docs**: A Sphinx-based package tailored for Espressif documentation, offering extensions for multi-target documentation, automated API generation, and more. See *What is ESP-Docs* for details.

Both tools generate static HTML (and optionally PDF) output. The generated static site can be deployed to any hosting platform. In Espressif projects, `.rst` documentation is commonly deployed to Espressif's internal server or Read the Docs. See *ESP-Docs & Espressif Server v.s. Sphinx & Read the Docs* for details on choosing the right combination.

Build Packages for Markdown

For `.md` sources, several SSG options are available, each offering distinct features and workflows:

- VitePress: A modern SSG built on Vite, commonly used for technical documentation. It provides sidebar navigation, search, versioning, and multi-language support. For example, Espressif's internal **CI Services Wiki** uses VitePress. To access the documentation, navigate to Documentation Team Site > ESP-Docs User Guide > Espressif's internal ci-services-wiki & ci-services-wiki documentation pages.
- Starlight: An Astro-based documentation framework used by the IDF team to generate the **Chip Support Wiki** page. To access this site, navigate to Documentation Team Site > ESP-Docs User Guide > Chip Support Wiki page.
- mdBook: A command-line tool to create books with Markdown, ideal for creating product or API documentation, tutorials and course materials. For example, it is used to generate the ESP32-C3 book.
- MkDocs: A popular Python-based SSG for project documentation, known for its simplicity and ease of configuration. For example, it is used to generate the **esp-api-check** and **Athena Python Client** documentation pages. To access these, navigate to Documentation Team Site > ESP-Docs User Guide > esp-api-check & Athena Python Client documentation pages.

The generated static sites can be deployed to any hosting platform. In Espressif projects, `.md` documentation is commonly deployed to GitLab Pages, GitHub Pages, or internal web servers.

1.2.4 Comparison table

Dimension	reStructuredText	Markdown
Learning curve	Higher; more syntax and concepts	Lower; simpler syntax to start writing
Extensibility	Built-in extensibility via Sphinx `roles`/`directives` and extensions	No standard extension system; different editors use different plugins
API reference	Mature automation via Sphinx ecosystem (e.g., Doxygen integration)	Possible, but often relies on some third-party API generators
Tables	Rich table formats (merged cells, lists, specified column widths)	Basic tables
Links & cross-references	Strong cross-references across docs; can avoid raw URLs	Typically relies on plain links
TOC & navigation	`toctree` can generate TOC and sidebar automatically	Manual TOC configuration; repo rendering lacks navigation
Build toolchain	Typically ESP-Docs or Sphinx	Various SSG options (e.g., VitePress, Starlight, mdBook, MkDocs)
Output & deployment	Static site (HTML; often PDF too); commonly deployed to Espressif Server or Read the Docs	Static site (HTML); commonly deployed to GitLab Pages/GitHub Pages/Espressif Server
Best for	Large, complex documentation sets that benefit from automation and cross-references	Teams that prefer Markdown and want a modern site with minimal setup via an SSG

1.3 ESP-Docs & Espressif Server v.s. Sphinx & Read the Docs

Among all Espressif software documentation, some are built with *ESP-Docs* and deployed to Espressif server (recommended), such as ESP-IDF Programming Guide, and some are built with Sphinx and deployed to Read the Docs (RTD), such as ESP-ADF Guide.

This document compares the above two ways of building and deploying Espressif documentation and explains why the former is recommended for new Espressif software documentation. If your documentation has already adopted the latter, you can choose whether to switch to ESP-Docs for building and Espressif server for hosting based on your needs.

Dimension	ESP-Docs & Espressif Server	Sphinx & Read the Docs
Feature	✓ More *features*, including 1) those provided by Sphinx 2) those provided by Sphinx third-party extensions, which are standardized to fixed versions to reduce build or deploy issues 3) those developed only for Espressif documentation, such as support for multiple targets. They are actively maintained and contribution to new or existing extensions is very welcome.	✗ Fewer features, including 1) those provided by Sphinx 2) those provided by Sphinx third-party extensions, some of which are not set to fixed versions, thus causing build or deploy issues from time to time.
Configuring deployment	✗ More workload. For deployment information, see *Update CI Configuration File*.	✓ Easier. For deployment information, see Documentation Team Site > Section ESP-Docs User Guide > Read the Docs Configuration Notes for Espressif doc.
Debugging deployment issues	✓ Independent debugging without engaging third parties, thus quicker.	✗ Needing support from RTD team, because RTD often breaks in ways we can not debug ourselves.
Debugging build issues caused by dependent packages	✓ The project or documentation owner can get help and support internally from Documentation Team and ESP-Docs developers.	✗ The project or documentation owner should fix them.
Access to documentation	✓ Quicker access	✗ Slow access to RTD servers from China. A caching reverse proxy at `docs.espressif.com` is provided to speed up the access, but if the cache is cold, the page load time can be high (>= 0.5 s).

Chapter 2

Writing Documentation

2.1 Basic Syntax

This document covers some basic reST syntax used in documentation built with ESP-Docs.

- *Paragraphs*
- *Inline Formatting*
  - *Italic*
  - *Bold*
  - *Literal*
- *Titles and Headings*
- *Section Numbering*
- *Lists*
  - *Bulleted Lists*
  - *Numbered Lists*
  - *Nested Lists*
- *Code Blocks*
  - *Simple Code Blocks*
  - *Bash Code Blocks*
  - *Python Code Blocks*
  - *none Code Blocks*
- *Tables of Contents*
- *Index Files*
- *Substitutions*
- *To-Do Notes*

2.1.1 Paragraphs

The paragraph is the most basic block in a reST document. Paragraphs are simply chunks of text separated by one or more blank lines. As in Python, indentation is significant in reST.

2.1.2 Inline Formatting

You can specify inline formatting through special symbols around the text you want to format.

Italic

Use single asterisks to show text as italic or emphasized.

Syntax:

```
*text*
```

Rendering result:

text

Bold

Use double asterisks to show text as bold or strong.

Syntax:

```
**text**
```

Rendering result:

text

Literal

Use double backquotes to show text as inline literal, to indicate code snippets, variable names, UI elements, etc.

Syntax:

```
``code``
```

Rendering result:

`code`

2.1.3 Titles and Headings

Normally, there are no heading levels assigned to certain characters as the structure is determined from the succession of headings. However, it is better to stick to the same convention throughout a project. For instance:

- # with overline, for parts
- * with overline, for chapters
- =, for sections
- −, for subsections
- ^, for subsubsections
- ", for paragraphs

2.1.4 Section Numbering

Section numbering is generally **not recommended**, particularly when done manually. However, if no alternative exists, it is advisable to use automatic methods.

To automatically number sections and subsections **across documents**, see *Index Files* > `numbered` option.

To automatically number sections and subsections **in one document**, use

Syntax:

```
.. sectnum::
  :depth: 3
  :prefix: 3.2.
  :start: 1
```

You may give the following options to the directive:

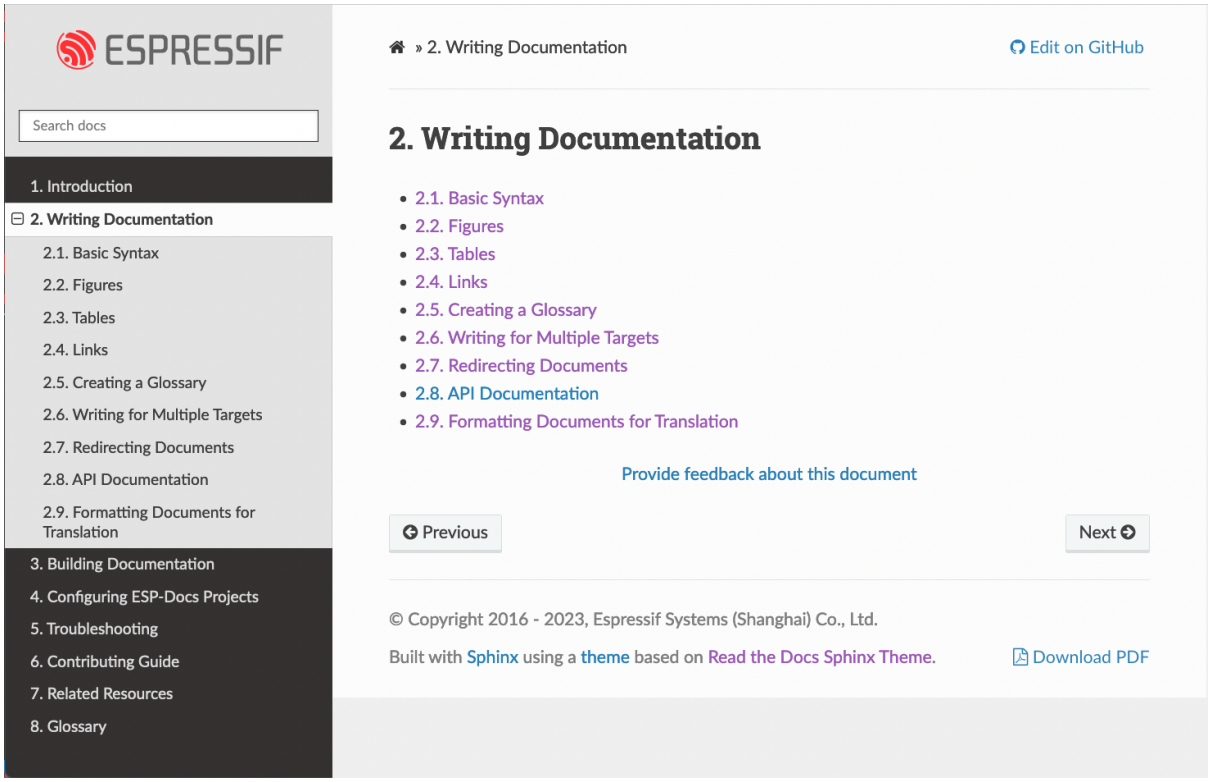


Fig. 1: Rendered Result - Numbering Across Documents (Click to Enlarge)

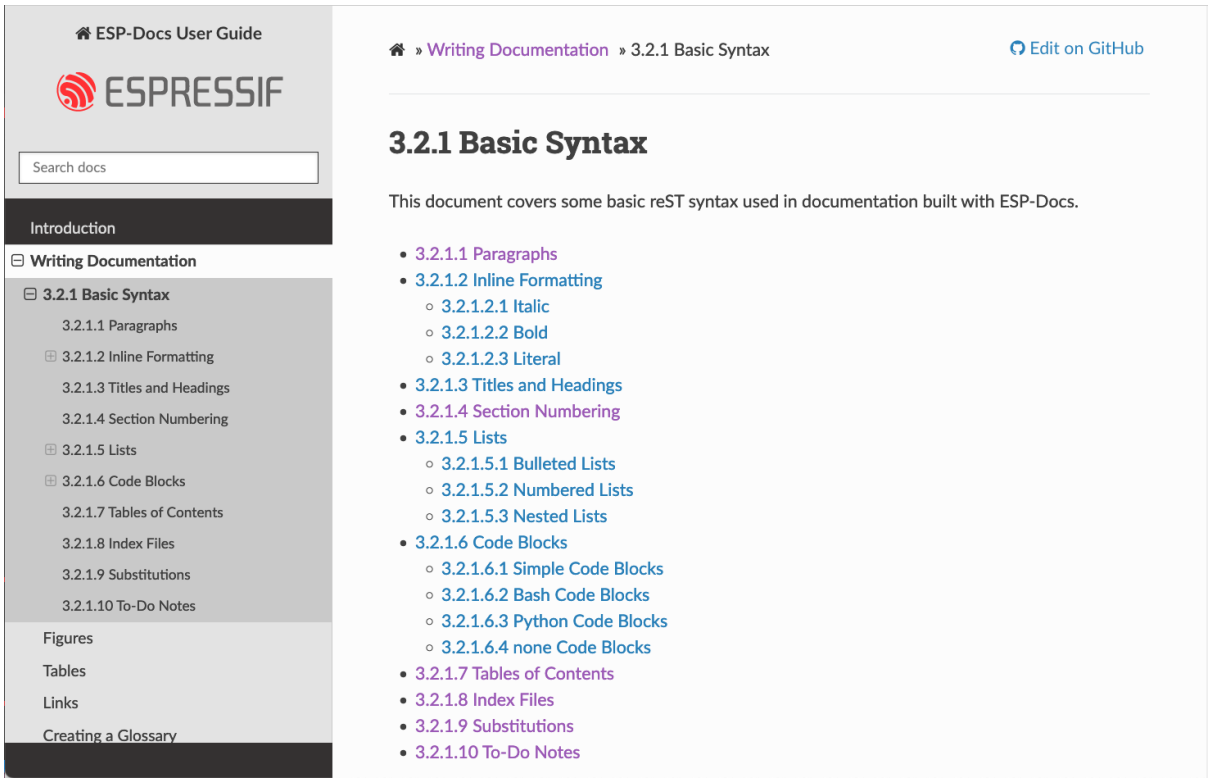


Fig. 2: Rendered Result - Numbering in One Document (Click to Enlarge)

- `:depth:` The number of section levels that are numbered by this directive. The default depth is unlimited.
- `:prefix:` An arbitrary string that is prefixed to the automatically generated section numbers. It may be something like "3.2.", which will produce "3.2.1", "3.2.2", and so on. The default is no prefix.
- `:start:` The value that will be used for the first section number. Combined with `prefix`, this may be used to force the right numbering for a document split over several source files. The default is 1.

However, the `sectnum` directive also needlessly numbers the title of the document itself. See invalid section numbering for reasons.

2.1.5 Lists

You can list items either in an ordered or unordered fashion.

Bulleted Lists

Syntax and example:

```
- Each bullet item starts with a symbol and a space.
- The symbol can be ``-``, ``*``, ``+``, etc.
```

Rendering result:

- Each bullet item starts with a symbol and a space.
- The symbol can be -, *, +, etc.

Numbered Lists

1. Common numbered lists

Syntax and example:

```
1. Each numbered list item starts with a symbol, a dot, and a space.
2. The symbol can be 1, A, i, (1) and so on.
```

Rendering result:

1. Each numbered list item starts with a symbol, a dot, and a space.
2. The symbol can be 1, A, i, (1) and so on.

2. Automatic numbered lists

Syntax and example:

```
#. Each automatic numbered list item starts with the number sign (#), a dot, and a␣
↪space.
#. The number sign is #.
```

Rendering result:

1. Each automatic numbered list item starts with the number sign (#), a dot, and a space.
2. The number sign is #.

Nested Lists

Example:

```
- This is the first item of the bulleted list.
- This is the second item of the bulleted list.

  1. This is the first item of the numbered list.
  2. This is the second item of the numbered list.

- This is the third item of the bulleted list.
```

Rendering result:

- This is the first item of the bulleted list.
- This is the second item of the bulleted list.
  1. This is the first item of the numbered list.
  2. This is the second item of the numbered list.
- This is the third item of the bulleted list.

Note:

1. Separate different levels of list items with a line.
2. The same level of list items should have the same indentation.

2.1.6 Code Blocks

A code block consists of the `code-block` directive and the actual code indented by four spaces for consistency with other code bases. For Python, C, Bash, and other programming languages, the keywords are highlighted by default.

Simple Code Blocks

Syntax and example:

```
::

    AT+GMR
```

Rendering result:

```
AT+GMR
```

Bash Code Blocks

Syntax and example:

```
.. code-block:: bash

    ls
    pwd
    touch a.txt
```

Rendering result:

```
ls
pwd
touch a.txt
```

Python Code Blocks

Syntax and example:

```
.. code-block:: python

    for i in range(10):
        print(i)
```

Rendering result:

```
for i in range(10):
    print(i)
```

none Code Blocks

If no other type applies, use "none". It can be useful for obscure languages or mixtures of languages like this mix of Bash and Python.

Syntax and example:

```
.. code-block:: none

    cat program.py

    for i in range(10):
        print(i)
```

Rendering result:

```
cat program.py

for i in range(10):
    print(i)
```

For more types, please refer to code blocks.

2.1.7 Tables of Contents

To create a table of contents (TOC), use

Syntax:

```
.. contents::
  :local:
  :depth: 1
```

You may give the following options to the directive:

- `:local:`: Generate a local table of contents. Entries will only include subsections of the section in which the directive is given. If no explicit title is given, the table of contents will not be titled.
- `:depth:`: The number of section levels that are collected in the table of contents. The default depth is unlimited.

To generate a TOC of the whole document, use

Syntax:

```
.. contents::
  :depth: 1
```

To generate a TOC of a section, use

Syntax:

```
.. contents::
  :local:
  :depth: 1
```

2.1.8 Index Files

Instead of using the `contents` directive to show a table of its own contents, the index file uses the `toctree` directive to create a table of contents **across** files.

Syntax and example:

```
.. toctree::
    :hidden:

    introduction/index
    writing-documentation/index
    building-documentation/index
    configuring-esp-docs-projects/index
    troubleshooting/index
    contributing-guide
    related-resources
    glossary
```

Rendering result:

See *ESP-Docs User Guide*

You may give the following options to the directive:

- `:maxdepth:` The maximum depth of the TOC.
- `:hidden:` The toctree is hidden in which case they will be used to build the left navigation column but not appear in the main page text.
- `:numbered:` (**not recommended**): Numbering starts from the heading of the top level. Sub-toctrees are also automatically numbered. In the example above, numbering will begin from the heading level of `introduction`.

For more information, see Sphinx TOC tree documentation.

2.1.9 Substitutions

Use a substitution to reuse short, inline content. Substitution definitions are indicated by an explicit markup start (".. ") followed by a vertical bar, the substitution text, another vertical bar, whitespace, and the definition block. A substitution definition block contains an embedded inline-compatible directive (without the leading ".. "), such as "image" or "replace" .

For example, use a substitution for a short list of CPU exceptions. To print the CPU exceptions, enter `|CPU_EXCEPTIONS_LIST|`.

Syntax and example:

```
CPU exceptions: |CPU_EXCEPTIONS_LIST|
```

The value of `|CPU_EXCEPTIONS_LIST|` is defined in a substitution definition.

Syntax and example:

```
.. |CPU_EXCEPTIONS_LIST| replace:: Illegal instruction, load/store alignment error,
↪ load/store prohibited error, double exception.
```

Rendering result:

CPU Exceptions: Illegal instruction, load/store alignment error, load/store prohibited error, double exception.

If you then change the replace value of the substitution, the new value will be used in all instances when you rebuild the project.

For more information, see Sphinx substitutions documentation.

2.1.10 To-Do Notes

Working on a document, you might need to:

- Give some suggestions on what should be added or modified in future.
- Leave a reminder for yourself or somebody else to follow up.

In this case, add a to-do note to your reST file using the directive `.. todo::`.

Syntax and example:

```
.. todo::

    Add a package diagram.
```

If you add `.. todolist::` to a reST file, the directive will be replaced by a list of all to-do notes from the whole documentation.

By default, the directives `.. todo::` and `.. todolist::` are ignored by documentation builders. If you want the notes and the list of notes to be visible in your locally built documentation, take the following steps:

1. Open your local `conf_common.py` file.
2. Find the parameter `todo_include_todos`.
3. Change its value from `False` to `True`.

Note: Before pushing your changes to origin, please set the value of `todo_include_todos` back to `False`. Otherwise, you will make all the to-do notes visible to customers, too.

For more information, see sphinx.ext.todo documentation.

To learn more about the basic syntax, visit Docutils Quick reStructuredText.

2.2 Figures

This document will briefly introduce the common image formats used in Espressif software documentation built with ESP-Docs, describe their usage, and provide corresponding examples for writers' reference.

2.2.1 Why Add Figures?

Figures serve an essential role in conveying complex technical information. If you are writing some technical text and feel like expressing your ideas is getting increasingly harder (for example, while describing logical connections), consider using a diagram. Even the most complex ideas that are hard to understand when written as text can be quickly understood with the simplest of diagrams. The key to success is to choose the right diagram type for your case.

Luckily, diagrams in Espressif software documentation built with ESP-Docs already have more or less established styles.

2.2.2 Adding Figures in ESP-Docs

There are different ways of rendering images in documentation: - Directives to include ready-to-use pictures created by graphic editors. - Diagram as Code to create diagrams based on textual descriptions for documents based on markup languages.

Using Directives

Pictures could be built in documentation using directives and options. Writers can include a ready-to-use figure with the following source code:

```
.. figure:: ../../_static/figure-raster-image-usage.png
    :align: center
    :width: 70%
    :alt: Development of Applications

    This is the caption of the figure (optional)
```

Below is the image in PNG format added through the above directives and options:

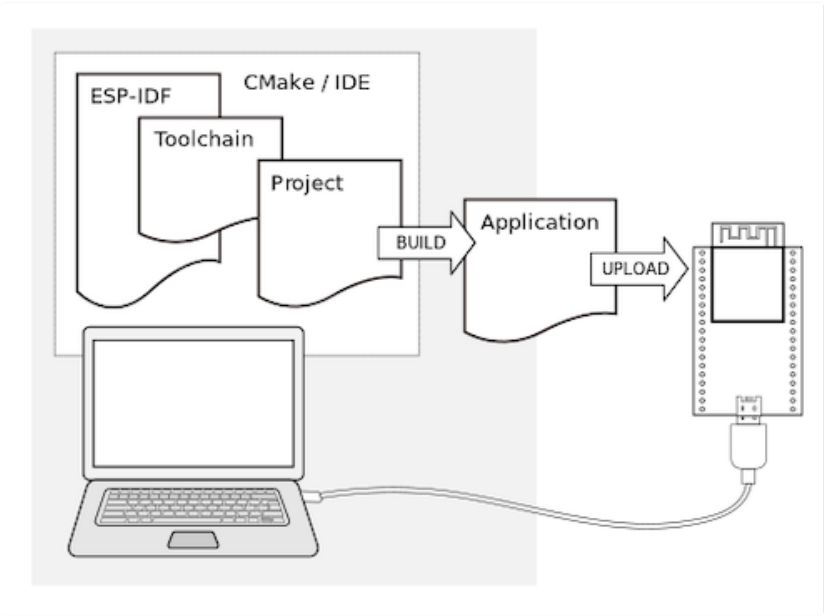


Fig. 3: This is the caption of the figure (optional)

For detailed information about how to use these directives, please refer to Section Figure in the reStructuredText documentation. Below are some notes for writers when using the directives in our documentation.

- For the `.. figure::` directive, the path followed can either be a URL, or a relative path to your figures in the current project. For example, to link the specific figure under the `_static` folder, it can be written as:

```
    .. figure:: ../../_static/doc-format1-recommend.png

or to access the separate server through the URL::

    .. figure:: https://dl.espressif.com/dl/sche,atocs/pictures/esp32-s2-
↪kaluga-1-kit-v1.0-3d.png

Note that, for the relative path, if you are not sure about it, please check␣
```

(continues on next page)

(continued from previous page)

```
↪in the terminal using ``cd ..``. For the URL, if the figures are too large,␣
↪upload it to a separate server, then provide the URL.

Generally, for each repo, figures are stored in the ``_static`` folder. Below␣
↪are some of the paths for your information:

    - ESP-IDF: `esp-idf/docs/_static <https://github.com/espressif/esp-idf/
↪tree/master/docs/_static>`_
    - ESP-ADF: `esp-adf-internal/docs/_static <https://github.com/espressif/
↪esp-adf/tree/master/docs/_static>`_
    - ESP-AT: `esp-at/docs/_static <https://github.com/espressif/esp-at/tree/
↪master/docs/_static>`_
    - ESP-Docs: `esp-docs/docs/_static <https://github.com/espressif/esp-docs/
↪tree/master/docs/_static>`_
    - esp-dev-kits: `esp-dev-kits/docs/_static <https://github.com/espressif/
↪esp-dev-kits/tree/master/docs/_static>`_

Note that if you use the ``... figure::`` directive to upload the non-editable␣
↪diagrams (PNG, JPG, etc.), please remember to also upload the editable copy␣
↪(SVG, ODG, etc.) with the same name as the non-editable diagrams uploaded to␣
↪the internal image-storing GitLab repository corresponding to the current␣
↪repository. It is also recommended to add a commented-out link to the␣
↪editable copy in the figure directive for easier search. The reason why we␣
↪are doing this is that while the editable copy could be too large to make␣
↪the repository hard to pull, storing them in another repository could always␣
↪be a fortune when the content of the document has changed and writers are␣
↪able to find the original images and edit them at any time.
```

- For the `align:` option, while another option, `figclass: align-` is sometimes used together in ESP-IDF, the priorities are listed below:
  - If the alignments are the same, such as `:align: left` and `:figclass: align-left` are used, then the figure will be aligned left.
  - If different alignments are defined, such as `:align: center` and `:figclass: align-left` are used, then the figure will be aligned center (top priority) > left > right (the lowest priority), as `align:` has a higher priority than `figclass: align-`.

  Thus, it is recommended to use `align:` instead of `figclass: align-` in the documentation.
- For the `:width:` and `:scale:` options, note that the text column width on RTD HTML pages is approximately **700 px**, so figures should be sized accordingly to display properly. Without either option, figures are displayed at their original pixel dimensions.
  - `:width:` (**recommended**): specify the image width as a percentage of the **available text column width** (e.g., `:width: 70%`). The height is adjusted automatically to maintain the aspect ratio. Because this percentage is always relative to the text column width —approximately 700 px in HTML and the LaTeX `\textwidth` in PDF —all figures are displayed at a consistent size in both HTML and PDF output, regardless of their original dimensions.
  - `:scale:` is an alternative, but it has several limitations:
    * **Requires knowing the original dimensions**: Because `:scale:` is relative to the original pixel dimensions of the image, you need to know the original size and calculate the right percentage manually. For example, if the original figure is 3452×1590 px, you need to calculate that `:scale: 20%` results in 690×318 px, which fits within the 700 px page width.
    * **Not applicable to URL images without a workaround**: If a URL is provided as the figure path, an error `Could not obtain image size. :scale: option is ignored.` might occur. As a workaround, you need to provide the original width and height of the image explicitly:

```
.. figure:: https://dl.espressif.com/dl/schematics/pictures/esp-lyrap-
↪lcd32-v1.1-3d.png
    :align: center
    :width: 2243px
    :height: 1534px
    :scale: 30%
    :alt: ESP-LyraP-LCD32
```

    - **Inconsistent sizes across figures**: Because `:scale:` is relative to each image's original pixel dimensions, the same percentage value produces different display sizes for images with different original sizes, which can lead to layout inconsistencies in both HTML and PDF output.
- For the `:alt:` option, it shows the alternate description of figures. This description will be displayed when the figure is shown not properly on display. Normally, the caption of the figure would be placed here. If the figure is scaled, then the writer should also add **(Click to enlarge)** after the caption.

Using Diagram as Code

For adding graphics using Diagram as Code, several Sphinx extensions are provided to generate diagram images from simple text files:

- sphinx.ext.graphviz: Sphinx extension to generate diagrams from Graphviz DOT code, including directed graphs, undirected graphs, and so on.
- Sphinx extensions in the blockdiag suite that generate diagrams from a common DOT-like *blockdiag* syntax:
  - sphinxcontrib-blockdiag: generates block diagrams.
  - sphinxcontrib-seqdiag: generates sequence diagrams.
  - sphinxcontrib-actdiag: generates activity diagrams.
  - sphinxcontrib-nwdiag: generates logical network diagrams, rack-structure diagrams, and packet header diagrams.
- sphinxcontrib-wavedrom: independent extension to generate digital timing diagrams from WaveJSON code.

With these extensions, diagram images are generated from simple text descriptions. The diagram elements are laid out automatically. The diagram code is then typically rendered as raster images (often PNG) and integrated "behind the scenes" into **Sphinx** documents. Choose the one that best fits your diagram type.

Using sphinx.ext.graphviz sphinx.ext.graphviz supports various diagram types, including:

- Directed graphs, such as flowcharts
- Undirected graphs, such as data packet structure diagrams

To use this extension, follow the steps below:

1. **Install the Graphviz package.**
   Before building documentation with this extension, install the Graphviz binaries on your system. Installers for all major platforms are available at https://graphviz.org/download/.
2. **Add** `sphinx.ext.graphviz` **to the** `extensions` **list in** `conf.py`:

```
extensions += [
    'sphinx.ext.graphviz',
]
```

3. **Create a diagram.**
   Below is an example and its source code:
   Source code:

```
.. graphviz::
    :caption: ESP-IDF Development Workflow
    :align: center

    digraph esp_idf_develop {
        graph [rankdir=TB fontname="Helvetica" fontsize=13
               splines=ortho nodesep=0.5 ranksep=0.6 pad=0.4]
        node  [fontname="Helvetica" fontsize=11 style="filled"
               fillcolor="white" color="#333333" shape=box margin="0.2,0.12"]
        edge  [fontname="Helvetica" fontsize=10 arrowsize=0.8 color="#333333"]

        start_project   [label="Start a project\nCreate or copy an example"␣
↪style="filled,dashed"]
        connect_device  [label="Connect your device"]
        configure       [label="Configure project"]
```

(continues on next page)

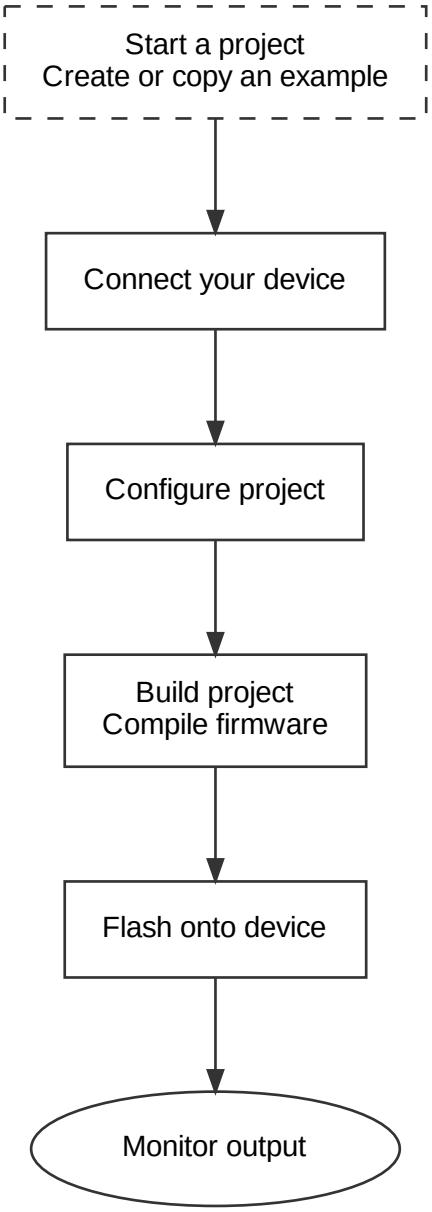


Fig. 4: ESP-IDF Development Workflow

(continued from previous page)

```
        build           [label="Build project\nCompile firmware"]
        flash           [label="Flash onto device"]
        monitor         [label="Monitor output" shape=oval]

        start_project -> connect_device -> configure -> build -> flash ->␣
↪monitor
    }
```

For diagrams with lengthy DOT source, save the code in a `.dot` file and reference it by path:

```
.. graphviz:: ../../../_static/diagrams/development_workflow.dot
    :caption: ESP-IDF Development Workflow
    :align: center
```

4. **Preview the diagram.**
   To quickly preview the diagram, you can:
   - Use the Graphviz Online tool.
   - Use a Graphviz extension in your IDE, such as Graphviz Preview for VS Code.
   - *Build documentation locally* and preview the diagram in your browser.

For the full DOT language reference and additional layout options, see the Graphviz documentation and the sphinx.ext.graphviz extension documentation.

Using the blockdiag suite The blockdiag suite supports the following diagram types, each handled by a separate extension:

- sphinxcontrib-blockdiag: block diagrams
- sphinxcontrib-seqdiag: sequence diagrams
- sphinxcontrib-actdiag: activity diagrams
- sphinxcontrib-nwdiag: logical network diagrams, rack-structure diagrams, and packet header diagrams

All four extensions are enabled in esp-docs by default. They share the same DOT-like syntax. Using `sphinxcontrib-blockdiag` as an example:

1. **Create a diagram.**
   Below is an example and its source code:

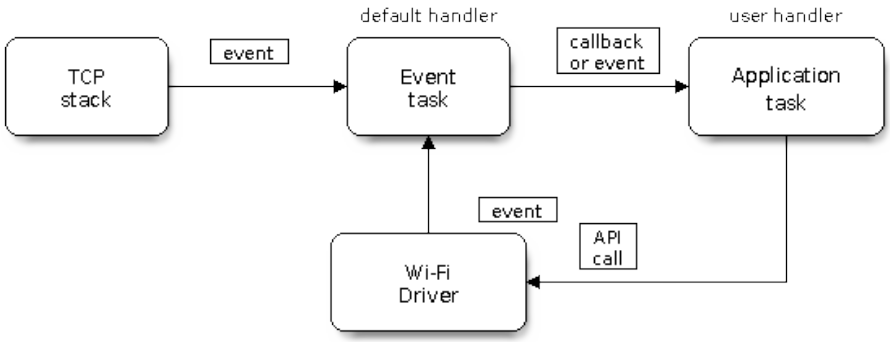


Fig. 5: Wi-Fi Programming Model

Source code:

```
.. blockdiag::
    :caption: Wi-Fi Programming Model
    :align: center
```

(continues on next page)

(continued from previous page)

```
    blockdiag wifi-programming-model {

        # global attributes
        node_height = 60;
        node_width = 100;
        span_width = 100;
        span_height = 60;
        default_shape = roundedbox;
        default_group_color = none;

        # node labels
        TCP_STACK [label="TCP\n stack", fontsize=12];
        EVNT_TASK [label="Event\n task", fontsize=12];
        APPL_TASK [label="Application\n task", width = 120,␣
↪fontsize=12];
        WIFI_DRV  [label="Wi-Fi\n Driver", width = 120, fontsize=12];
        KNOT [shape=none];

        # node connections + labels
        TCP_STACK -> EVNT_TASK [label=event];
        EVNT_TASK -> APPL_TASK [label="callback\n or event"];

        # arrange nodes vertically
        group {
        label = "default handler";
        orientation = portrait;
        EVNT_TASK <- WIFI_DRV [label=event];
        }

        # intermediate node
        group {
            label = "user handler";
            orientation = portrait;
            APPL_TASK -- KNOT;
        }
        WIFI_DRV <- KNOT [label="API\n call"];
    }
```

For diagrams with lengthy source, save the code in a `.diag` file and reference it by path:

```
.. blockdiag:: ../../../_static/diagrams/twai/state_transition.diag
    :caption: State transition diagram of the TWAI driver (see table␣
↪below)
    :align: center
```

2. **Preview the diagram.**
   To quickly preview the diagram, you can:
   - Use the Kroki online editor.
   - *Build documentation locally* and preview the diagram in your browser.

The following examples from ESP-IDF are available for reference:

- Simple **block diagram** / `blockdiag` - Wi-Fi TX buffer allocation
- Slightly more complicated **block diagram** - Wi-Fi programming model
- **Sequence diagram** / `seqdiag` - Foreground scan of all Wi-Fi channels
- **Rack-structure diagram** / `rackdiag` - Ethernet Data Frame Format
- **Packet diagram** / `packetdiag` - Structure of RMT symbols

For more details, see the blockdiag online documentation.

Using sphinxcontrib-wavedrom sphinxcontrib-wavedrom generates signal waveforms from WaveJSON code.

To use this extension, follow the steps below:

1. **Add the extension to the `extensions` list in `conf.py`:**

```
extensions += [
    'sphinxcontrib.wavedrom',
]
```

2. **Create a diagram.**
   Below is an example and its source code:

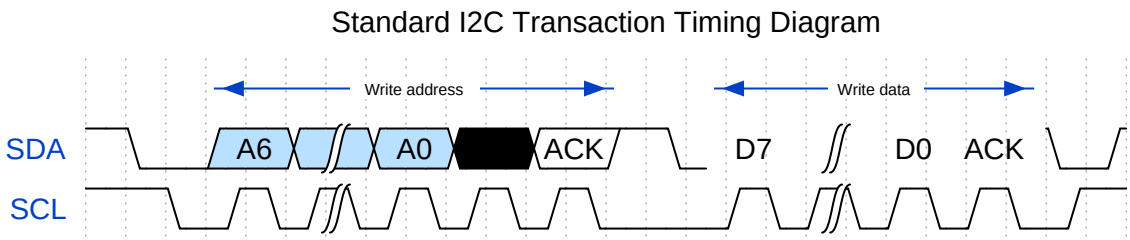


Fig. 6: I2C Transmission Waveform

Source code:

```
.. wavedrom::
    :caption: I2C Transmission Waveform
    :align: center

    {
        "head": {
            "text": "Standard I2C Transaction Timing Diagram"
        },
        "signal": [
            {
                "node":"A.....B..................C..D.E.............
↪F...G",
                "period": 0.5},
            {
                "name": "SDA",
                "wave": "1.0...3...3.|.3...4...5...1..0.6...6.|.6...7..
↪.10..1", "data": "A6 . A0 R/W ACK D7 . D0 ACK",
                "period": 0.5},
            {
                "name": "SCL",
                "wave": "1...0..1.0.1|0.1.0.1.0.1.0......1.0.1|0.1.0.1.
↪0..1..",
                "period": 0.5}
            ],
        "config":
        {
            "skin": "narrow"
        },
        "edge": [
            "B<->C Write address",
            "E<->F Write data"
        ]
    }
```

3. **Preview the diagram.**
   To quickly preview the diagram, you can:
   - Use the WaveDrom online editor.
   - Use a WaveDrom extension in your IDE, such as Waveform Render for VS Code.
   - *Build documentation locally* and preview the diagram in your browser.

For the WaveJSON language reference and additional options such as bitfield diagrams, see the WaveDrom tutorial and the sphinxcontrib-wavedrom documentation.

Summary To conclude, while ready-to-use images drawn in graphic editors might be easier to handle for writers with little experience in creating diagrams, they have rather larger size based on their resolution. As for text-based Diagram as Code graphics, it would undoubtedly cost writers some time to get started and master, but they are smaller in size and easier to version with Git. Thus, it is recommended to use Diagram as Code to present pictures in your files.

2.3 Tables

Tables can present complex information in an understandable way. With reStructuredText syntax, you can create tables in the following formats:

- *Simple Table*
- *Grid Table*
- *List Table* ???
- *CSV Table*

This document covers the syntax for all table formats and their pros and cons, so that you can choose the best fit for your use scenario. For more detailed instructions, please refer to reStructuredText Directives > Tables.

2.3.1 Simple Table

Simple tables are preceded and ended with a sequence of = to indicate columns.

Texts in the same column should be aligned with = on the left, and not extend beyond = on the right.

Simple tables supports:

- **Column span**: Cells in multiple columns (except last row) can be merged by adding a sequence of –
- **Table notes**: Manually numbered footnote `[1]_` and autonumbered footnote `[#]`
- **Insert pictures**

```
.. table::
  :align: center

  ================================   ============================
  ESP-Docs 用户指南
  ------------------------------------------------------------------
  ================================   ============================
  |write-doc|                        |build-doc|
  Writing Documentation [1]_         Building Documentation [#]_
  Covers ESP-Docs supported syntax   介绍如何预览、构建文档
  ================================   ============================

.. |write-doc| image:: ../../_static/writing-documentation.png
            :height: 100px
            :width: 100px
.. |build-doc| image:: ../../_static/building-documentation.png
            :height: 100px
            :width: 100px

.. [1] This is a manually numbered table note. Note that it generates links from␣
↪notes back to the table.
.. [#] This is an autonumbered table note. It generates no backlinks and continues␣
↪numbering from the previous note.
```

The above table would be rendered as:

ESP-Docs 用户指南	
	
Writing Documentation[1]	Building Documentation[2]
Covers ESP-Docs supported syntax	介绍如何预览、构建文档

2.3.2 Grid Table

Grid tables are named after its grid structure formed by delimiters +, −, and |.

Grid tables support:

- **Column span**
- **Row span**
- **Table notes**
- **Bullet Lists**
- **Insert pictures** (For example, see *ESP-Docs User Guide*)

If there are Chinese characters, the vertical bars | can hardly be aligned to form a grid.

```
.. table::
  :align: center

+----------+-----------------------------------------+---------------------------+
| 芯片     | 描述                                    | Ambient Temperature [#]_ |
|          |                                          +-------------+-------------+
|          |                                          | Min (°C)    | Max (°C)    |
+==========+=========================================+=============+=============+
| ESP32-C3 | ESP32-C3 is a single-core, 32-bit,      | -40         | 105         |
|          | RISC-V-based MCU with 400 KB of SRAM,   |             |             |
|          | which is capable of running at 160 MHz.|             |             |
+----------+-----------------------------------------+-------------+-------------+
| ESP32-S3 | ESP32-S3 is a dual-core XTensa LX7 MCU,| -40         | 105         |
|          | capable of running at 240 MHz.          |             |             |
+----------+-----------------------------------------+-------------+-------------+

.. [#] This is an autonumbered table note. Note that the automatic numbering␣
↪continues from the previous table note.
```

The above table would be rendered as:

[1] This is a manually numbered table note. Note that it generates links from notes back to the table.
[2] This is an autonumbered table note. It generates no backlinks and continues numbering from the previous note.

芯片	描述	Ambient Temperature[Page 30, 3]	
		Min (°C)	Max (°C)
ESP32-C3	ESP32-C3 is a single-core, 32-bit, RISC-V-based MCU with 400 KB of SRAM, which is capable of running at 160 MHz.	-40	105
ESP32-S3	ESP32-S3 is a dual-core Xtensa LX7 MCU, capable of running at 240 MHz.	-40	105

To facilitate the generation of grid tables, you may use tools such as Tables Generator.

2.3.3 List Table

List tables are formed of two-level lists, where the first level * represents rows, and the second level - represents columns.

The number of columns must be consistent. Empty table cells should still by marked by -, even if there is no content.

List tables support:

- **Adjustable column width**
- **Table notes**
- **Bullet Lists**
- **Insert pictures**

```
.. list-table::
    :header-rows: 1
    :widths: 40 60
    :align: center

    * - Field
      - Value (Byte)
    * - Type (Least Significant Bit)
      - 1
    * - Frame Control (Frag)
      -
    * - 序列号
      - 1
    * - 数据长度
      - 1
    * - Data
      - * Total Content Length: 2
        * Content: ${Data Length} - 2
    * - CheckSum (Most Significant Bit) [#]_
      - 2

.. [#] This is an autonumbered table note. Note that the automatic numbering␣
↪continues from the previous table note.
```

The above table would be rendered as:

[3] This is an autonumbered table note. Note that the automatic numbering continues from the previous table note.

Field	Value (Byte)
Type (Least Significant Bit)	1
Frame Control (Frag)	
序列号	1
数据长度	1
Data	• Total Content Length: 2 • Content: ${Data Length} - 2
CheckSum (Most Significant Bit)[4]	2

2.3.4 CSV Table

CSV (comma-separated values) tables might be the choice if you want to include CSV data into your documentation. The CSV data may be:

- placed in a separate CSV file
- an integral part of the document

As for formatting, CSV tables only support adjustable column width.

- Example of integrating a separate CSV file:

```
.. csv-table:: Table Title
  :file: CSV file path and name
  :widths: 30, 70
  :align: center
  :header-rows: 1
```

- Example of integrating CSV data as an integral part of the document:

```
.. csv-table:: Ordering Information
  :header: "订购代码", "Flash Size"
  :widths: 50, 50
  :align: center

  ESP32-C3,N/A
  ESP32-C3FN4,"4 MB"
  ESP32-C3FH4,"4 MB"
```

The above table would be rendered as:

Table 1: Ordering Information

订购代码	Flash Size
ESP32-C3	N/A
ESP32-C3FN4	4 MB
ESP32-C3FH4	4 MB

Note: Text with spaces in between should be enclosed by quotation marks, such as `"4 MB"`.

2.3.5 Comparison

To summarize:

[4] This is an autonumbered table note. Note that the automatic numbering continues from the previous table note.

- 推荐 List tables are ideal because they achieve a balance between easy maintenance and advanced formatting features.
- Simple tables are good choices when table cells do not contain long sentences.
- Grid tables provide more formatting options, but they are the most difficult to maintain.
- CSV tables are convenient to present simple data, but not friendly to text with spaces.

	Simple Table	Grid Table	List Table	CSV Table
What you see is what you get	✓	✓		
Easy to maintain	✓		✓	
Friendly to Chinese characters	✓		✓	✓
Friendly to long text		✓	✓	
Adjustable table width			✓	✓
Row span		✓		
Column span	✓	✓		
Bullet points		✓	✓	

2.3.6 Still No Good Fit?

If the above table formats cannot meet your needs, consider adding new table extensions. For example, to use a list table for its easy maintenance, but with column span and row span features, you may refer to to the flat-table directive.

2.4 Links

This document introduces how to link to different elements of documentation when you write documents with ESP-Docs.

2.4.1 Table of Contents

- *Linking to Language Versions*
- *Linking to Other Sections Within the Document*
- *Linking to Other Documents*
- *Linking to a Specific Place of Other Documents in a Same Project*
- *Linking to Kconfig References*
- *Linking to Classes, Functions, Enumerations, etc*
- *Linking to GitHub Files*
- *Linking to External Pages*
- *Linking to ESP TRMs and Datasheets*
  - *Linking to a Whole TRM or Datasheet File*
  - *Linking to Chapters of a TRM or Datasheet File*
- *Resources*

When writing documentation, you often need to link to other language versions of the document, other sections within the document, other documents, GitHub files, etc. An easy way is just to use the raw URL that Sphinx generates for each page or section. This works, but it has some disadvantages:

- Links can change, so they are hard to maintain.
- Links can be verbose and hard to read, so it is unclear what page or section they are linking to.
- There is no easy way to link to specific sections like paragraphs, figures, or code blocks.
- URL links only work for the HTML version of your documentation.

Instead, Sphinx offers a powerful way to link to different elements of the document, called cross-references. Some advantages of using them:

- Use a human-readable name of your choice, instead of a URL.
- Portable between formats: HTML, PDF, ePub.

- Sphinx will warn you of invalid references.
- You can cross-reference more than just pages and section headers.

2.4.2 Linking to Language Versions

Switching between documentation in different languages may be done using the `:link_to_translation:` custom role. The role placed on a page of documentation provides a link to the same page in a language specified as a parameter. Examples below show how to enter links to Chinese and English versions of documentation.

Syntax and examples:

```
:link_to_translation:`zh_CN: 中文版`
:link_to_translation:`en:English`
```

The language is specified using standard abbreviations like `en` or `zh_CN`. The text after last semicolon is not standardized and may be entered depending on the context where the link is placed, e.g.:

```
:link_to_translation:`en:see description in English`
```

2.4.3 Linking to Other Sections Within the Document

Syntax and example:

```
`Linking to ESP TRMs and Datasheets`_
```

Rendering result:

Linking to ESP TRMs and Datasheets

2.4.4 Linking to Other Documents

If you want to link to other documents in the same folder, which is the `docs` folder here, you can either use the path relative to the root folder or relative to the document you want to link to. In addition, you can also display the document title as the link text or customize the link text. Please note that we recommend using the path relative to the root folder as links will not break when you move the document containing the links.

- You can use the following syntax to display the document title as the link text.
  Syntax:

  ```
  :doc:`relative path to the root folder`
  :doc:`relative path to the document you want to link to`
  ```

  Example:

  ```
  :doc:`/introduction/index`
  :doc:`../introduction/index`
  ```

  Rendering result:
  Introduction
  Introduction
- If you want to customize the link text, you can use the following syntax.
  Syntax:

  ```
  :doc:`CustomizedLinkText <path from the documentation root, starting with />`
  :doc:`CustomizedLinkText <relative path to the document you want to link to>`
  ```

  Example:

```
:doc:`Another Introduction </introduction/index>` (the documentation root here␣
↪is esp-docs/docs/en)
:doc:`Another Introduction <../introduction/index>`
```

Rendering result:
Another Introduction
Another Introduction

2.4.5 Linking to a Specific Place of Other Documents in a Same Project

To link to a specific place of documents in a same project, you need to first add an anchor in the specific place and then reference it in the document.

- Add an anchor to the specific place where you want to link to with the following syntax.
  Syntax:

  ```
  .. _AnchorName:
  ```

  Example:

  ```
  .. _building-documentation-1
  ```

- Insert the anchor in your document with the following syntaxes. You can either display the section name after the anchor as the link text or customize the link text.
  - Display the section name after the anchor as the link text
    Syntax:

    ```
    :ref:`AnchorName`
    ```

    Example:

    ```
    :ref:`building-documentation-1`
    ```

    Rendering result:
    Building HTML Locally on Your PC
  - Customize the link text
    Syntax:

    ```
    :ref:`CustomizedLinkText <AnchorName>`
    ```

    Example:

    ```
    :ref:`Building Document <building-documentation-1>`
    ```

    Rendering result:
    Building Document

If the anchor is placed directly before a section title, figure, or captioned table, Sphinx automatically uses the heading or caption as the link text.

Anchors that are not placed before a section title can still be referenced, but you must give the link an explicit title, using this syntax:

```
:ref:`Link title <anchor-name>`
```

Otherwise, warnings will be raised:

```
WARNING: Failed to create a cross reference. A title or caption not found: anchor-
↪name
```

Note: Using anchors is advised over standard reStructuredText links to sections mentioned in *Linking to Other Sections Within the Document* because anchors work across files, and when section headings are changed, warnings will be raised if incorrect.

2.4.6 Linking to Kconfig References

If you need to link to Kconfig references when writing documentation, please refer to the following syntax. The references are generated by kconfig_reference.py . We use the Kconfig files of ESP-IDF as examples to introduce this syntax.

Syntax and examples:

```
- :ref:`CONFIG_APP_COMPATIBLE_PRE_V3_1_BOOTLOADERS`
- :ref:`CONFIG_APP_COMPATIBLE_PRE_V2_1_BOOTLOADERS`
- :ref:`CONFIG_APP_BUILD_TYPE`
- :ref:`CONFIG_APP_REPRODUCIBLE_BUILD`
- :ref:`CONFIG_APP_NO_BLOBS`
```

If you use `:ref:`CONFIG_APP_COMPATIBLE_PRE_V3_1_BOOTLOADERS`` in ESP-IDF documents, this can lead you to the description of this Kconfig reference.

2.4.7 Linking to Classes, Functions, Enumerations, etc

For linking to classes, functions, enumerations and other structure types in the doxygen API documentation, please refer to the following syntax. We also use structure types defined in ESP-IDF as examples to introduce this syntax.

Syntax:

```
- Class - :cpp:class:`name`
- Function - :cpp:func:`name`
- Structure - :cpp:type:`name`
- Structure Member - :cpp:member:`struct_name::member_name`
- Enumeration - :cpp:type:`name`
- Enumeration Value - :cpp:enumerator:`name`
- Defines - :c:macro:`name`
```

Examples:

```
- Class - :cpp:class:`esp_mqtt_client_config_t`
- Function - :cpp:func:`esp-gcov_dump`
- Structure - :cpp:type:`mesh_cfg_t`
- Structure Member - :cpp:member:`eth_esp32_emac_config_t::clock_config`
- Enumeration - :cpp:type:`esp_partition_type_t`
- Enumeration Value - :cpp:enumerator:`WIFI_MODE_APSTA`
- Defines - :c:macro:`ESP_OK`
```

2.4.8 Linking to GitHub Files

In addition to linking to documentation in the `docs` folder, you may also need to link to other files in the project, for example, the header and program files. You can link to them on GitHub.

When linking to files on GitHub, do not use absolute/hardcoded URLs. We have developed link_roles.py , so that you can use Docutils custom roles to generate links. These auto-generated links point to the tree or blob for the git commit ID (or tag) of the repository. This is needed to ensure that links do not get broken when files in the master branch are moved around or deleted. The roles will transparently handle files that are located in submodules and will link to the submodule' s repository with the correct commit ID.

Syntax and explanation:

```
- :project:`path`  - points to directories in the project repository
- :project_file:`path`  - points to files in the project repository
- :project_raw:`path`  - points to raw view of files in the project repository
- :component:`path`  - points to directories in the components directory of the␣
↪project repository
```

(continues on next page)

(continued from previous page)

```
- :component_file:`path`  - points to files in the components directory of the␣
↪project repository
- :component_raw:`path`  - points to raw view of files in the components directory␣
↪of the project repository
- :example:`path`  - points to directories in the examples directory of the␣
↪project repository
- :example_file:`path`  - points to files in the examples directory of the project␣
↪repository
- :example_raw:`path`  - points to raw view of files in the examples directory of␣
↪the project repository
```

Examples:

```
- :example:`doxygen/en`
- :example:`English Version <doxygen/en>`
- :example_file:`doxygen/en/conf.py`
- :example_raw:`doxygen/en/conf.py`
```

Rendering results:

- doxygen/en
- English Version
- doxygen/en/conf.py
- doxygen/en/conf.py

By running `build-docs gh-linkcheck`, you can search .rst files for presence of hard-coded links (identified by tree/master, blob/master, or raw/master part of the URL). This check is recommended to be added to the CI pipeline.

2.4.9 Linking to External Pages

Generally, you can always use URL to link to external pages. For example, if you want link to Espressif ' s homepage, you can refer to the following syntax.

Syntax and example:

```
Welcome to `Espressif <https://www.espressif.com/>`_!
```

Rendering result:

Welcome to Espressif!

Please note that if you have several links with the same display text, it will lead to the Sphinx warning `duplicate explicit target names`. To avoid this issue, you can use two underscores __ at the end of links. For example,

```
Welcome to `Espressif <https://www.espressif.com/>`__!
```

Rendering result:

Welcome to Espressif!

2.4.10 Linking to ESP TRMs and Datasheets

If you need to link to Espressif ' s TRMs and datasheets of different targets, you can also use the external links introduced above. However, ESP-Docs offers a simple way by defining the macros {IDF_TARGET_TRM_EN_URL}, {IDF_TARGET_TRM_CN_URL}, {IDF_TARGET_DATASHEET_EN_URL} and {IDF_TARGET_DATASHEET_CN_URL}. You can directly use them to link to related TRMs and datasheets. For details, please refer to format_esp_target.py .

Linking to a Whole TRM or Datasheet File

You can choose a macro to link to the TRM or datasheet of a specific target in your document.

Syntax and example:

```
Please refer to `ESP32-S3 TRM <{IDF_TARGET_TRM_EN_URL}>`__.
Please refer to `ESP32-S3 Datasheet <{IDF_TARGET_DATASHEET_EN_URL}>`__.
```

Linking to Chapters of a TRM or Datasheet File

You can link to a specific chapter of a TRM or datasheet file by appending #hypertarget-name at the end of the macros. This hypertarget acts like a bookmark.

For example, if you need to refer to Chapter I2C Controller in the ESP32-S3 TRM, use the following link.

Syntax and example:

```
For details, please refer to *ESP32-S3 Technical Reference Manual* > *I2C␣
↪Controller (I2C)* [`PDF <{IDF_TARGET_TRM_EN_URL}#i2c>`__].
```

For the specific hypertargets of chapters in different ESP TRMs, please go to Documentation Team Site > Section ESP-Docs User Guide > Hypertargets of chapters.

2.4.11 Resources

For more information about links, please refer to Cross-referencing with Sphinx.

2.5 Creating a Glossary

A glossary or "glossary of terms" is a collection of words pertaining to a specific topic. Usually, it is a list of all terms you used that may not immediately be obvious to your reader. Your glossary only needs to include terms that your reader may not be familiar with, and is intended to enhance their understanding of your work.

Glossaries are not mandatory, but if you use a lot of technical or field-specific terms, it may improve readability to add one. A good example is *Glossary* in the ESP-Docs User Guide.

If you are going to create a glossary for your project, then you are the target audience of this document. This document describes how to:

- create a consolidated glossary of terms.
- link terms in other documents to their definitions in the glossary.

2.5.1 Create Glossary of Terms

To create a glossary of terms, you can use the directive `.. glossary::`. Write each glossary entry as a definition list in the form of a term followed by a single-line indented definition as below:

```
.. glossary::

    Term A
        Definition

    Term B
        Definition
```

The above content will be rendered in the document in the form of:

Term A
: Definition

Term B
: Definition

You can also give the glossary directive a `:sorted:` flag that will automatically sort the entries alphabetically.

```
.. glossary::
    :sorted:

    B-term
        Definition B

    A-term
        Definition A
```

As you can notice, although we wrote B-term before A-term, after applying `:sorted:`, the rendered effect would be:

A-term
: Definition A

B-term
: Definition B

2.5.2 Link a Term to its Glossary Entry

After a glossary is created with the `.. glossary::` directive containing a definition list with terms and definitions, you can link a term to its definition in the glossary by using the `:term:` role.

For example the ESP-Docs User Guide has one global *Glossary*. You can use the the following syntax to link the term add-ons to its definition:

```
Please refer to :term:`add-ons`.
```

This will be rendered as:

Please refer to *add-ons*.

Important:

- The term specified must exactly match a term in the glossary directive. If you use a term that is not explained in a glossary, you' ll get a warning during the documentation build.
- The term used in your document can only be linked to its definition in the glossary when your document and the glossary are in the same project. For example, this document, which is in the project of ESP-Docs User Guide, can not be linked to the terms defined in the ESP-ADF Glossary.

You can link to a term in the glossary while showing different text in the topic by including the term in angle brackets. For example:

```
This file is written in :term:`rst <reStructuredText>` format.
```

This will be rendered as:

This file is written in *rst* format.

Important: The term in angle brackets must exactly match a term in the glossary. The text before the angle brackets is what users see on the page.

2.6 Writing for Multiple Targets

Espressif provides a rich list of chip products, e.g., ESP32, ESP32-S2, ESP32-C3, which are referred to as "targets" in ESP-Docs. Technical documentation differs for each specific chip, yet a large part of the content is reusable among different targets.

To facilitate the writing of documents that can be reused for multiple different chips, several functionalities are provided in ESP-Docs for writers to deal with target-specific inline text, paragraph, bullet point, and even document while building the documentation for all Espressif's chips from the same files.

2.6.1 Target-Specific Inline Text

When the content is reusable for all ESP chips, but you need to refer to the specific chip name, toolchain name, path, hardware/software specification, or other inline text that varies among different targets in the paragraph, consider using the substitution macros supplied by the extension Format ESP Target . Substitution macros allow you to generate target-specific inline text from the same source file with the target passed to the Sphinx command line.

For example, in the following reStructuredText content, the substitution macros (referred to as tag hereinafter) IDF_TARGET_NAME, IDF_TARGET_PATH_NAME, IDF_TARGET_TOOLCHAIN_PREFIX, and IDF_TARGET_TOOLCHAIN_PREFIX defined in esp_extensions/format_esp_target.py are used:

```
This is {IDF_TARGET_NAME} with /{IDF_TARGET_PATH_NAME}/soc.c, compiled with `{IDF_
↪TARGET_TOOLCHAIN_PREFIX}-gcc` with `CONFIG_{IDF_TARGET_TOOLCHAIN_PREFIX}_MULTI_
↪DOC`.
```

The text will be rendered for ESP32-S2 chip as the following:

```
This is ESP32-S2 with /esp32s2/soc.c, compiled with `xtensa-esp32s2-elf-gcc` with␣
↪`CONFIG_ESP32S2_MULTI_DOC`.
```

This extension also supports markup for defining local substitutions within a single source file. Place a definition like the following in a single line to define a target-dependent substitution of the tag IDF_TARGET_SUFFIX in the current reStructuredText file:

```
{IDF_TARGET_SUFFIX:default="DEFAULT_VALUE", esp32="ESP32_VALUE", esp32s2="ESP32S2_
↪VALUE", esp32c3="ESP32C3_VALUE"}
```

For example:

```
{IDF_TARGET_TX_PIN:default="IO3", esp32="IO4", esp32s2="IO5", esp32c3="IO6"}
```

The above line will define a substitution for the tag IDF_TARGET_TX_PIN, which would be replaced by the text "IO5" if Sphinx is called with the target esp32s2 and "IO3" if called with esp32s3. You may also use the text "Not updated" for the default value.

In the case when multiple targets have the same value (may not be the default value) to be substituted, you can even group such targets together to avoid re-writing the same values multiple times.

For example::

{IDF_TARGET_SBV2_KEY:default=" RSA-3072" , esp32c6, esp32h2=" RSA-3072 or ECDSA-256 or ECDSA-192" }

The above line will define a substitution for the tag IDF_TARGET_SBV2_KEY, which would be replaced by the text "RSA-3072 or ECDSA-256 or ECDSA-192" if Sphinx is called with the target esp32c6 or esp32h2 and "RSA-3072" if called with any other target.

Note:

- These single-file definitions can be placed anywhere in the reStructuredText file on their own line, but the name of the directive must start with IDF_TARGET_.
- Also note that these replacements cannot be used inside markup that rely on alignment of characters, e.g., tables.

ESP-Docs also allows other extensions to add additional substitutions through Sphinx events. For example, in ESP-IDF it is possible to use defines from `soc_caps.h`:

```
The target has {IDF_TARGET_SOC_SPI_PERIPH_NUM} SPI peripherals.
```

The text will be rendered for ESP32-S2 as the following:

```
The target has 3 SPI peripherals.
```

For a full overview of available substitutions in your project, you can take a look at `IDF_TARGET-substitutions.txt`, which is generated in the build folder when a project is built.

2.6.2 Target-Specific Paragraph

In a document shared by multiple targets, occasionally there will be paragraphs only applicable to one or some of the targets, or the paragraphs should be customized for different targets. `ESP-Docs` introduces the `.. only:: TAG` directive provided by the Sphinx selective exclude extension to help you define specific chip targets for target-specific content in the document.

To use the `.. only:: TAG` directive, simply follow the steps described below:

1. Define the target of the content and replace "TAG" with one of the following options:

- Chip names. For example:
  - esp32 > `.. only:: esp32`
  - esp32s2 > `.. only:: esp32s2`
  - esp32c3 > `.. only:: esp32c3`
- Or other tags you define and configure based on your own needs. For example, there are two kinds of customized tags in esp-idf:
  - Tags defined in the `sdkconfig.h` header files, e.g., `CONFIG_FREERTOS_UNICORE`, which are generated by the default menuconfig settings for the target.
  - Tags defined in the `*_caps.h` header files, e.g., `SOC_BT_SUPPORTED` and `SOC_CAN_SUPPORTED`.

2. Place the directive before the content that you want to exclude from the rest of the document:

```
.. only:: esp32

    ESP32-specific content.
```

Note: Note that it is required to leave a blank line after the directive and to indent before the content.

In this way, Sphinx will only generate the content for the target that you have defined using the directive, e.g., ESP32 in the example above.

This directive also supports the boolean operators `and`, `or`, and `not`. For example:

- `.. only:: not esp32c2`
- `.. only:: esp32 or esp32s2`
- `.. only:: SOC_BT_SUPPORTED and CONFIG_FREERTOS_UNICORE`

Note that the extension sometimes does not correctly handle the case where you exclude a section that is directly followed by a labeled new section. For example:

```
.. only:: esp32

    .. _section_1_label:
```

(continues on next page)

(continued from previous page)

```
    Section 1
    ^^^^^^^^^
    Section 1 content

.. _section_2_label:

Section 2
^^^^^^^^^
Section 2 content
```

In the above case, if the label `section_2_label` does not correctly link to the section that follows, refer to the temporary workaround below when this cannot be avoided:

```
.. only:: esp32

    .. _section_1_label:

    Section 1
    ^^^^^^^^^
    Section 1 content

    .. _section_2_label:

.. only:: not esp32

    .. _section_2_label:

Section 2
^^^^^^^^^
Section 2 content
```

2.6.3 Target-Specific Bullet Point

The `:TAG:` role provided by `ESP-Docs` comes in handy when you need to define targets for content inside a list of bullet points. To achieve this, simply add the `:TAG:` inside the `.. list::` directive before the items. For example:

```
.. list::

    :esp32: - ESP32-specific content
    :esp32c2 or esp32c3: - Content specific to ESP32-C2 and ESP32-C3
    :SOC_BT_SUPPORTED: - Bluetooth-specific content
    - Common bullet point 1
    - Common bullet point 2
```

Then Sphinx will only generate the first bullet point for ESP32 documentation, the second bullet point for ESP32-C2 and ESP32-C3 documentations, and the third bullet point for targets that support Bluetooth after you define the `SOC_BT_SUPPORTED` tag.

2.6.4 Target-Specific Document

It is also possible to define targets for a whole document using the `:TAG:` role in a table of content tree. After you place the `:TAG:` role before the toctree item, Sphinx will use the role to include or exclude content based on the target it was called with.

For example, in the following toctree extracted from the index of api-guides for esp-idf, the tags `SOC_BT_SUPPORTED`, `SOC_RTC_MEM_SUPPORTED`, and `SOC_USB_OTG_SUPPORTED` (defined in the `\*_caps` header files) are used:

```
.. toctree::
    :maxdepth: 1

    app_trace
    startup
    :SOC_BT_SUPPORTED: blufi
    bootloader
    build-system
    core_dump
    :SOC_RTC_MEM_SUPPORTED: deep-sleep-stub
    error-handling
    :esp32s3: flash_psram_config
    :not esp32c6: RF_calibration
```

In this way, Sphinx will only link to the documents `blufi.rst` and `deep-sleep-stub.rst` for targets that support these functions, the document `flash_psram_config.rst` for ESP32-S3, and the document `RF_calibration.rst` for all the targets except for ESP32-C6.

Note that if you need to exclude an entire document from the toctree based on targets, it is necessary to also update the configuration in `docs/conf_common.py` to exclude the file for other targets, or a Sphinx warning "WARNING: document isn't included in any toctree" will be generated.

The recommended way of doing it is adding a `conditional_include_dict` list in `docs/conf_common.py` and include the document to one of the list that gets included. Examples can be found in docs/conf_common.py in esp-idf, where, for instance, a document which should only be shown for Bluetooth-capable targets should be added to `BT_DOCS`. The `exclude_docs.py` will then take care of adding it to `conditional_include_dict` if the corresponding tag is not set.

2.7 Redirecting Documents

During documentation lifetime, some source files are moved between folders or renamed, and the original links to these files will be broken. Manually fixing these links one by one is time-consuming.

To solve this issue, the html_redirects.py extension is provided to redirect pages that have changed URLs. The extension is integrated in *ESP-Docs*.

This extension is used together with a redirection list *html_redirect_pages*, which is generated by conf_common.py from page_redirects.txt .

conf_common.py is a configuration file for your project. To enable the html_redirects.py extension, you need to add html_redirects.py to its *extension* list to enable this extension.

page_redirects.txt is a file that includes both old URLs and updated URLs. By reading this file, html_redirects.py generates a redirection list *html_redirect_pages*, thus redirecting old URLs to updated ones.

If you want to rename a document, for example, rename *docs/en/introduction* to *docs/en/get-started*, or redirect a document, for example, redirect *docs/en/writing-documentation/basic-syntax* to a web page, and do not want to update the links manually, you can follow the below steps.

- Open your *conf_common.py* file and append html_redirects.py to the *extensions* list, thus enabling it in your project:

```
extensions += [
    ...
    'generic_extensions.html_redirects'
    ...
              ]
```

This step is done only once for a project.

- Create the file *docs/page_redirects.txt* to include the old and new URLs. *conf_common.py* will build the list *html_redirect_pages* from *docs/page_redirects.txt*. You can check page_redirects.txt as an example.

- Add content following the below format to the *page_redirects.txt* file.

```
old URL   new URL
```

In the above two scenarios, the URLs added in the file should be:

```
docs/en/introduction                                    docs/en/get-started
docs/en/writing-documentation/basic-syntax              "https://www.sphinx-doc.org/
↪en/master/usage/restructuredtext/basics.html"
```

The old URL must be relative to the document root only and MUST NOT contain the file extension, which is *.rst* in this case.

The new URL can either be an absolute URL or a relative URL.

- For absolute URLs, the URLs must be wrapped with double quotation marks. Whatever is inside the quotation marks is used verbatim as the URL. Don't forget to add the "https://" prefix to your absolute URL.
- For relative URLs, the URLs must be relative to the document root only and MUST NOT be wrapped with any quotation marks.

In this way, page_redirects.txt is used as a "recipe" to redirect to the new URLs.

2.8 Writing API Description

The API descriptions contain all the information required to work with the API, with details about every function, structure, enumeration, and other elements used in the API. To help you write informative API descriptions in a consistent style, this document provides guidelines on what to write with practical examples.

- *Document Conventions*
- *Macro*
- *Type Definition*
- *Enumeration*
- *Structure*
- *Union*
- *Function*

For details about formatting API documentation in header files, please refer to *Formatting and Generating API Descriptions*.

2.8.1 Document Conventions

This document uses the conventions shown below to indicate types of patterns:

Pattern	Example	Identifies
Descriptions enclosed in quotation marks ""	"Measurement unit: …"	A fixed sentence pattern for describing measurement unit
Directives indicated by the @ character	@brief	Tags used to define the formatting of the descriptions

2.8.2 Macro

Macros are used to define reusable values or code snippets, such as a clock frequency, Wi-Fi's SSID, default configurations, etc.

Macro descriptions should include:

- **@brief A brief description of the macro**
  - Use concise sentence fragments if possible
  - Example: *@brief The number of CPU cores.*
- **@note Target-specific information, or anything that needs emphasis**
  - Example: *@note This macro is only for ESP32.*

Example 1: ESP_BLUFI_BD_ADDR_LEN

```
/**
* @brief Bluetooth address length in bytes.
*
* @note Must be 6 bytes.
*/
#define ESP_BLUFI_BD_ADDR_LEN     6
```

The above code will be rendered as:



Example 2: ESP_NETIF_INHERENT_DEFAULT_OPENTHREAD

```
/**
* @brief  Default configuration of OT ESP-NETIF.
*/
#define ESP_NETIF_INHERENT_DEFAULT_OPENTHREAD() \
    {   \
        .flags = 0, \
        ESP_COMPILER_DESIGNATED_INIT_AGGREGATE_TYPE_EMPTY(mac) \
        ESP_COMPILER_DESIGNATED_INIT_AGGREGATE_TYPE_EMPTY(ip_info) \
        .get_ip_event = 0,     \
        .lost_ip_event = 0,    \
        .if_key = "OT_DEF",  \
        .if_desc = "openthread",     \
        .route_prio = 15       \
};
```

The above code will be rendered as:



2.8.3 Type Definition

Type definitions are used to create a type alias or define a new type.

Type definition descriptions should include:

- **@brief A brief description of the typedef**
  - Use concise sentence fragments if possible
  - Example: *@brief Event handler type.*

Note: When a type definition is used for function pointers or other similar cases, please refer to the corresponding guidelines for `function`, etc.

Example: pcnt_unit_handle_t

```
/**
* @brief PCNT unit handle type.
*/
typedef struct pcnt_unit_t *pcnt_unit_handle_t;
```

The above code will be rendered as:



2.8.4 Enumeration

Enumerations allow you to define a set of named values (or enumerators) as something textual and meaningful.

Enumeration descriptions should include:

- **@brief An overall description of the enumeration**
  - Use concise sentence fragments if possible
  - Example: *@brief Clock sources.*
- **Meanings of each enumerator**
  - Use concise sentence fragments or sentences
  - Example: *The duty resolution is 13 bits.*
- **@note Target-specific information, prerequisites to configure a structure member, or anything that needs emphasis**
  - Example: *@note The number of channels is different across chips.*

Example: ledc_mode_t

```
/**
* @brief LEDC speed mode.
*/
typedef enum {
    LEDC_HIGH_SPEED_MODE = 0, /*!< High speed mode. */
                              /*!< @note Only ESP32's LEDC supports high speed␣
↪mode. */
    LEDC_LOW_SPEED_MODE,      /*!< Low speed mode. */
    LEDC_SPEED_MODE_MAX,      /*!< Speed limit. */
} ledc_mode_t;
```

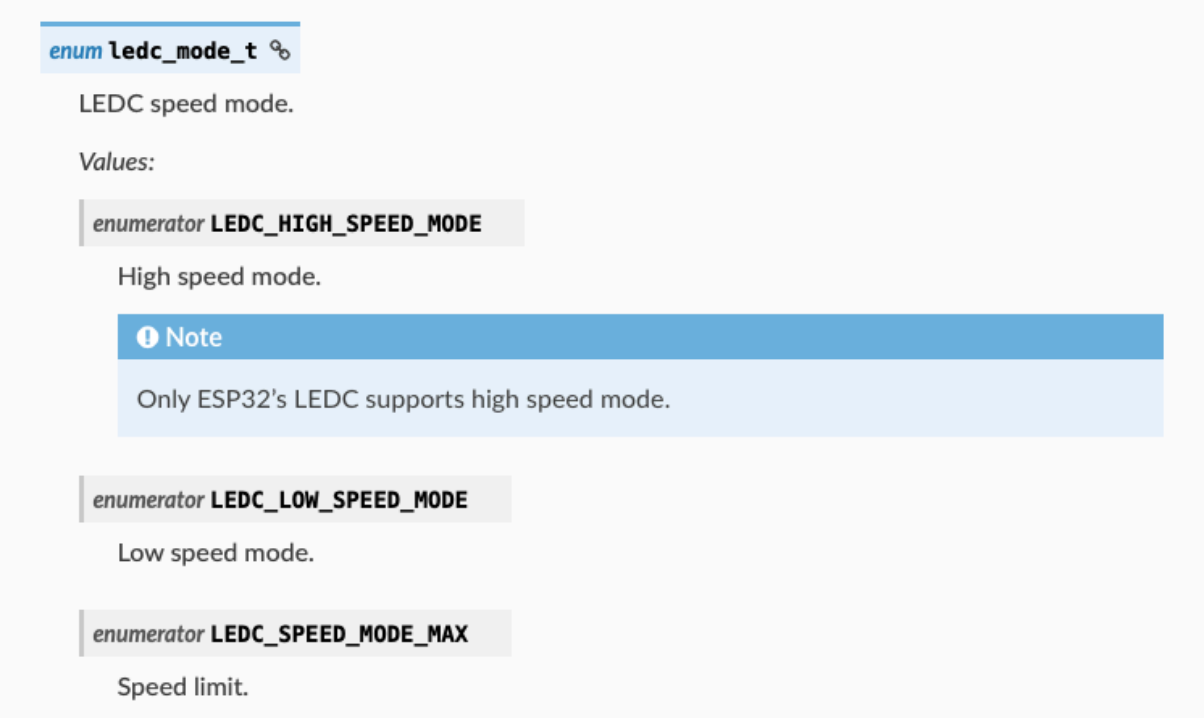
The above code will be rendered as:

2.8.5 Structure

Structures provide a way to group several related data elements (or members) into one place, so that functions can easily use them as parameters. Members in a structure may be of different data types such as `int`, `char`, and `bool`.

Structure descriptions should include:

- **@brief An overall description of the structure**



enum **ledc_mode_t**

LEDC speed mode.

Values:

enumerator **LEDC_HIGH_SPEED_MODE**

High speed mode.

Note

Only ESP32's LEDC supports high speed mode.

enumerator **LEDC_LOW_SPEED_MODE**

Low speed mode.

enumerator **LEDC_SPEED_MODE_MAX**

Speed limit.

  - Use concise sentence fragments if possible
  - Example: *@brief ESP-NOW rate configuration.*

- **A list of structure members**
  - **Description of each structure member**
    - Use concise sentence fragments if possible
    - If the structure member is a `bool`, use the format "True if …; false otherwise. "
    - Example: *True if the timer interrupts are shared; false otherwise.*
  - **"Measurement unit: …"** , if any
  - @note **Target-specific information, prerequisites to configure a structure member, or anything that needs emphasis**

Example 1: struct esp_ble_mesh_gen_level_set_t

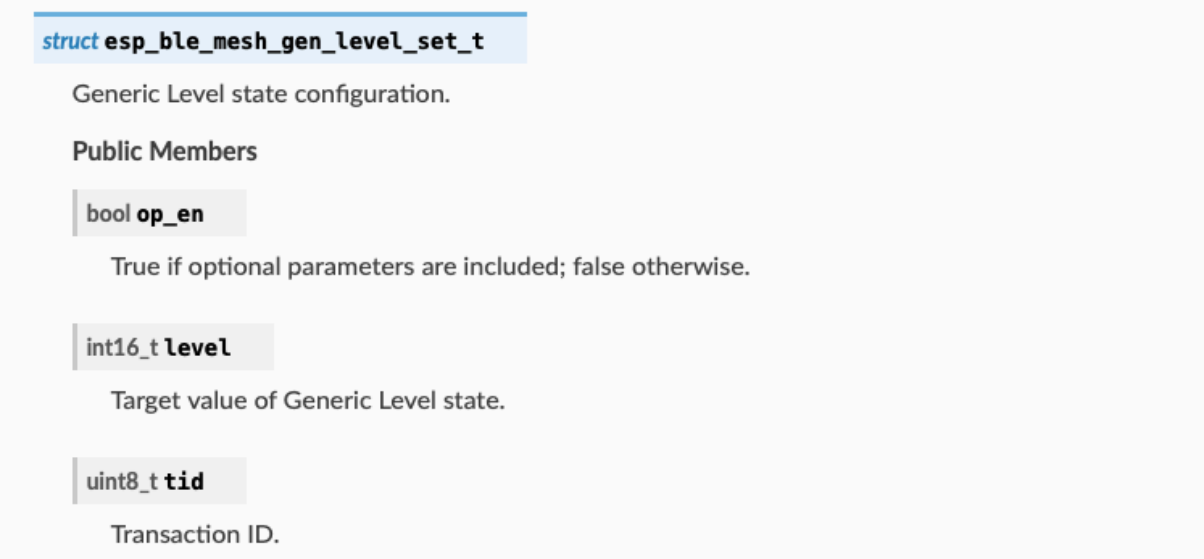
```
/**
* @brief Generic Level state configuration.
*/
typedef struct {
    bool    op_en;      /*!< True if optional parameters are included; false␣
↪otherwise. */
    int16_t level;      /*!< Target value of Generic Level state. */
    uint8_t tid;        /*!< Transaction ID. */
} esp_ble_mesh_gen_level_set_t;
```

The above code will be rendered as:

Example 2: struct ledc_channel_config_t

```
/**
* @brief LEDC timer configuration.
*/
typedef struct {
    ledc_mode_t speed_mode;                 /*!< LEDC speed mode. */
    ledc_timer_bit_t duty_resolution;       /*!< LEDC channel duty resolution. */
    uint32_t freq_hz;                       /*!< LEDC timer frequency. Measurement␣
↪unit: Hz. */
    ledc_clk_cfg_t clk_cfg;                 /*!< LEDC clock. */
```

(continues on next page)



struct **esp_ble_mesh_gen_level_set_t**

Generic Level state configuration.

Public Members

bool **op_en**

True if optional parameters are included; false otherwise.

int16_t **level**

Target value of Generic Level state.

uint8_t **tid**

Transaction ID.

(continued from previous page)

```
                                        /*!< @note  For ESP32 and ESP32-S2,␣
↪each timer can have a independent clock source. For other chips, all timers use␣
↪one collective clock source. */
} ledc_timer_config_t;
```

The above code will be rendered as:

2.8.6 Union

Similar to structures, unions are also data structures to hold multiple variables, but the members of unions are stored in the same memory locations.

Union descriptions should include:

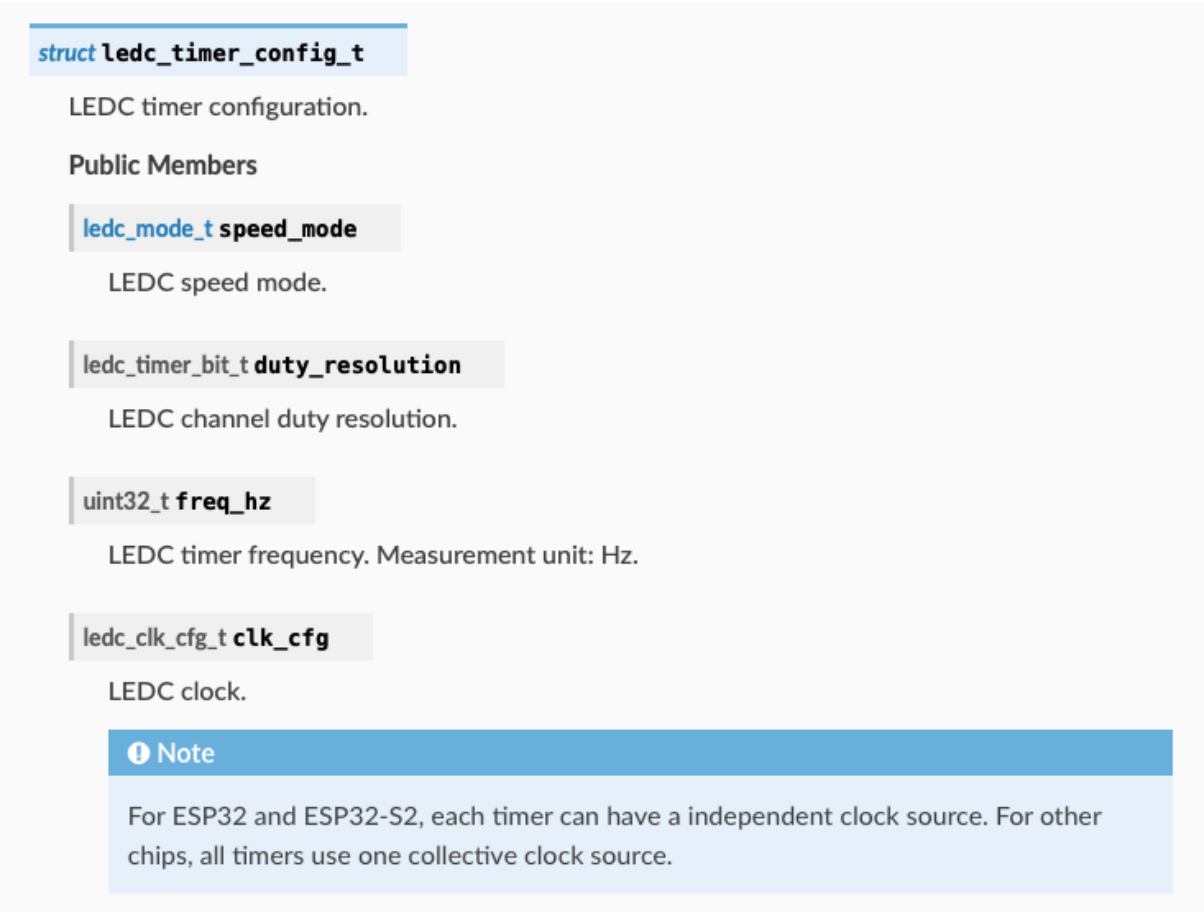
- **@brief An overall description of the union**
  - Use concise sentence fragments if possible
  - Example: *@brief GATT client callback parameters.*
- **A list of union members with descriptions**
  - **Description of each union member**
    - Use concise sentence fragments if possible
    - Example: *Signal duration.*

    If the union member is a structure, follow the writing guidelines for structures, that is, provide an overall description for the structure and individual descriptions for structure members. For reference, see line 5 to line 13 in the following example.
  - **"Measurement unit: ···"** , if any
  - @note **Target-specific information, prerequisites to configure a union member, or anything that needs emphasis**

Example: rmt_symbol_word_t

```
1  /**
2  * @brief Union to store the RMT symbol layout.
3  */
4  typedef union {
5      /**
6      * @brief RMT symbol duration and level configuration.
7      */
8      struct {
9          unsigned int duration0 : 15; /*!< Duration of level0. Measurement unit:␣
```

(continues on next page)



struct **ledc_timer_config_t**

LEDC timer configuration.

Public Members

ledc_mode_t **speed_mode**

LEDC speed mode.

ledc_timer_bit_t **duty_resolution**

LEDC channel duty resolution.

uint32_t **freq_hz**

LEDC timer frequency. Measurement unit: Hz.

ledc_clk_cfg_t **clk_cfg**

LEDC clock.

Note

For ESP32 and ESP32-S2, each timer can have a independent clock source. For other chips, all timers use one collective clock source.

(continued from previous page)

```
↪RMT tick. */
        unsigned int level0 : 1;      /*!< Level of the first part. */
        unsigned int duration1 : 15; /*!< Duration of level1. Measurement unit:␣
↪RMT tick. */
        unsigned int level1 : 1;      /*!< Level of the second part. */
    } structure_name;
    unsigned int val; /*!< The entire 32-bit RMT symbol. */
} rmt_symbol_word_t;
```

The above code will be rendered as:

2.8.7 Function

Functions encapsulate a set of instructions, and can accept parameters and return values.

Function descriptions should include:

- **@brief A brief description of the function**
  - Use concise sentence fragments if possible
  - Example: *Reset the timer.*
- **Description and direction of parameters**
  - Use concise sentence fragments if possible
  - If parameters have a measurement unit, remember to mention it with **"Measurement unit: …"**
  - Example: *PWM frequency. Measurement unit: MHz.*
- **Returned values and their meanings for non-void functions**
  - If the return value is a `bool`, use the format "True if …; false otherwise."
  - If the return value (especially for functions of the `esp_err_t` type) is a error code such as

union **rmt_symbol_word_t**

#include <my_api.h>

Union to store the RMT symbol layout.

Public Members

unsigned int **duration0**

Duration of level0. Measurement unit: RMT tick.

unsigned int **level0**

Level of the first part.

unsigned int **duration1**

Duration of level1. Measurement unit: RMT tick.

unsigned int **level1**

Level of the second part.

struct rmt_symbol_word_t **structure_name**

RMT symbol duration and level configuration.

unsigned int **val**

The entire 32-bit RMT symbol.

`ESP_ERR_INVALID_STATE`, provide specific error cause. For example, the description for `ESP_ERR_INVALID_STATE` can be `Duty cycle fading function not installed or started`, instead of `Invalid state` (see the highlighted line 12 ~ 14 in the example below).

- @note **Target-specific information, prerequisites to configure a structure member, or anything that needs emphasis**

Example: ledc_fade_stop

```
1  /**
2  * @brief Stop LEDC duty cycle fading.
3  *
4  * @note
5  *     1. This function can be called when you want to configure a fixed duty cycle␣
   ↪or a new fading but the last fade is still in progress.
6  *     2. This function only stops duty cycle fading if the fading is started via␣
   ↪`ledc_fade_start()` in `LEDC_FADE_NO_WAIT` mode. It cannot stop duty cycle␣
   ↪fading in `LEDC_FADE_WAIT_DONE` mode.
7  *     3. After this function returns values, the duty cycle of the channel will be␣
   ↪fixed one PWM cycle at most.
8  *
9  * @param[in] speed_mode LEDC speed mode.
10 * @param[in] channel LEDC channel number.
11 *
12 * @return
13 *     - ESP_OK: Done.
14 *     - ESP_ERR_INVALID_STATE: Duty cycle fading function not installed or started.
15 *
16 */
17 esp_err_t ledc_fade_stop(ledc_mode_t speed_mode, ledc_channel_t channel);
```

The above code will be rendered as:

esp_err_t **ledc_fade_stop**(ledc_mode_t speed_mode, ledc_channel_t channel)

Stop LEDC duty cycle fading.

Note

a. This function can be called when you want to configure a fixed duty cycle or a new fading but the last fade is still in progress.
b. This function only stops duty cycle fading if the fading is started via `ledc_fade_start()` in `LEDC_FADE_NO_WAIT` mode. It cannot stop duty cycle fading in `LEDC_FADE_WAIT_DONE` mode.
c. After this function returns values, the duty cycle of the channel will be fixed one PWM cycle at most.

Parameters:
- **speed_mode** – [in] LEDC speed mode.
- **channel** – [in] LEDC channel number.

Returns:
- ESP_OK: Done.
- ESP_ERR_INVALID_STATE: Duty cycle fading function not installed or started.

Note:

- If a parameter should be assigned with enum values (e.g. values of `ledc_channel_t`), there is no need to mention the `enum` in parameter descriptions given that the link to `enum` descriptions will be automatically generated and added (the pink circle in the above Figure).
- When referring to a function in API descriptions, always add brackets `()` after the function. That is, `ledc_fade_start()`, instead of `ledc_fade_start`.

2.9 Formatting and Generating API Descriptions

When you are documenting an API, there are some guidelines to follow, as demonstrated in *Writing API Descriptions*. Preparing such documentation could be tedious.

To simplify this process, ESP-Docs provides the run_doxygen.py extension, which generates API descriptions from header files during documentation build. This extension allows for automatic updates whenever code changes occur.

This document will cover the following topics:

- **Syntax and formatting rules to document API in header files**
  - *Comment Blocks*
  - *In-Body Comments*
  - *Target-Specific Information*
  - *Style*
- How to generate the API descriptions and include them in rst files

2.9.1 Document API in Header Files

This section covers the formatting rules for API descriptions, so that the run_doxygen.py extension knows which descriptions should be extracted from header files.

Comment Blocks

Comment blocks are used when documenting `functions`. Such comment blocks start with `/**`, and end with `*/`. Other lines within comment blocks should be marked with `*` at the beginning:

```
/**
* @brief A brief explanation for this function. It is mandatory.
*        If the explanation cannot fit into one line, start the second line with␣
↪indentation and a * at the beginning.
*
*        To break a line, break it twice (add an empty line in between), just like␣
↪how you do in rst files.
*
* @param [parameter_1's_name] [meaning.]
* @param [parameter_2's_name] [meaning.]
*
* @return
*     - [response_1]: meaning.
*     - [response_2]: meaning.
*/
[function_type] [function_name](parameter_1_type parameter_1, parameter_2_type␣
↪parameter_2);
```

`@brief`, `@param`, and `@return` form the basic structure for API descriptions. `@param` and `@return` can be skipped if a function does not have parameters or return any response.

If the function might return different responses, use a **bullet list** to document the responses under `@return`.

Comment blocks have some additional features, which can make the formatting of API descriptions fancier:

```
/**
 * @brief Stop LEDC duty cycle fading.
 *
 * @note
 *     1. This function can be called when you want to configure a fixed duty cycle or a new fading but
the last fade is still in progress.
 *     2. This function only stops duty cycle fading if the fading is started via `ledc_fade_start()`
in `LEDC_FADE_NO_WAIT` mode. It cannot stop duty cycle fading in `LEDC_FADE_WAIT_DONE` mode.
 *     3. After this function returns values, the duty cycle of the channel will be fixed one PWM cycle
at most.
 *
 * @param[in] speed_mode LEDC speed mode.
 * @param[in] channel LEDC channel number.
 *
 * @return
 *     - ESP_OK: Done.
 *     - ESP_ERR_INVALID_STATE: Duty cycle fading function not installed or started.
 *
 */
esp_err_t ledc_fade_stop(ledc_mode_t speed_mode, ledc_channel_t channel);
```



esp_err_t **ledc_fade_stop**(ledc_mode_t speed_mode, ledc_channel_t channel)

Stop LEDC duty cycle fading.

Note

a. This function can be called when you want to configure a fixed duty cycle or a new fading but the last fade is still in progress.

b. This function only stops duty cycle fading if the fading is started via `ledc_fade_start()` in `LEDC_FADE_NO_WAIT` mode. It cannot stop duty cycle fading in `LEDC_FADE_WAIT_DONE` mode.

c. After this function returns values, the duty cycle of the channel will be fixed one PWM cycle at most.

Parameters:
- **speed_mode** – **[in]** LEDC speed mode.
- **channel** – **[in]** LEDC channel number.

Returns:
- ESP_OK: Done.
- ESP_ERR_INVALID_STATE: Duty cycle fading function not installed or started.

Fig. 7: Rendered Result - Comment Blocks (Click to enlarge)

- Use [in], [out], [in, out] to document the direction of parameters:

```
*
* @param[in] [parameter_1's_name] [meaning.]
* @param[out] [parameter_2's_name] [meaning.]
*
```

- Add notes, warnings, or attentions after `@note`, `@warning`, or `@attention` respectively:

```
*
* @note
*    1. This is a numbered note. It is optional. Pay attention to the␣
↪indentation.
*    2. You can replace @note with @warning and @attention. The␣
↪formatting rules are the same.
*
* @warning If there is only one warning, the warning can be placed in␣
↪the same line with @warning.
*
```

- Add code snippets enclosed by `@code{c}` and `@endcode`:

```
*
* @code{c}
* // Example of using nvs_get_i32:
* int32_t max_buffer_size = 4096; // default value
* esp_err_t err = nvs_get_i32(my_handle, "max_buffer_size", &max_
↪buffer_size);
* assert(err == ESP_OK || err == ESP_ERR_NVS_NOT_FOUND);
* // if ESP_ERR_NVS_NOT_FOUND was returned, max_buffer_size will still
* // have its default value.
* @endcode
*
```

- Group similar functions by enclosing them with `/**@{*/` and `/**@}*/`:

```
/**@{*/
/**
* @brief      Set int8_t value for given key.
*
*
* @param[in]  value   The value to set.
*
* @return
*             - ESP_OK.
*             - ESP_FAIL.
*/
esp_err_t nvs_set_i8 (int8_t value);

/**
* @brief      Set uint16_t value for given key.
*
* This function is the same as \c nvs_set_i8 except for the data type.
*/
esp_err_t nvs_set_u16 (uint16_t value);
/**@}*/
```

- Use Markdown Syntax:

```
*
* @brief   Returns a random number inside a range.
*
*          See [ESP32 Technical Reference Manual](https://www.
↪espressif.com/sites/default/files/documentation/esp32_technical_
↪reference_manual_en.pdf).
```

(continues on next page)

(continued from previous page)

```
*
```

In-Body Comments

In-body comments are used when documenting a `macro`, a `typedef`, and members of a `struct`, `enum`, etc. Such in-body comments start with `/*!<`, and end with `*/`.

```
typedef struct {
    type member_1; /*!< Explanation for structure member_1. */
    type member_2; /*!< Explanation for structure member_2. */
    type member_3; /*!< Explanation for structure member_3. */
} structure_name
```

Optionally, comment blocks can be used together with in-body comments when you provide overall descriptions for a `struct`, `enum`, etc.

```
/**
* @brief A brief explanation for this structure.
*/
typedef struct {
    type member_1; /*!< Explanation for structure member_1. */
    type member_2; /*!< Explanation for structure member_2. */
    type member_3; /*!< Explanation for structure member_3. */
} structure_name
```

You may skip repetitive macros, enumerations, or other code by enclosing them within `/** @cond */` and `/** @endcond */`, so that they will not show in the generated API descriptions:

```
/** @cond */
typedef struct esp_flash_t esp_flash_t;
/** @endcond */
```

Target-Specific Information

ESP-Docs introduces several functionalities to deal with target-specific contents (see *Writing for Multiple Targets*), but such functionalities are not supported for API descriptions generated from header files.

For target-specific information, it is preferable to use `@note` to clarify the applicable targets.

Use `@note` for a target-specific function:

```
/**
* @brief Enable RX PDM mode.
* @note  ESP32-C3: Not applicable, because it doesn't support RX PDM mode.
*
* @param hw Peripheral I2S hardware instance address.
* @param pdm_enable Set true to RX enable PDM mode (ignored).
*/
static inline void i2s_ll_rx_enable_pdm(i2s_dev_t *hw, bool pdm_enable)
```

Use `@note` for a target-specific `struct`:

```
/**
* @brief ADC digital controller (DMA mode) output data format.
*        Used to analyze the acquired ADC (DMA) data.
* @note  ESP32: Only `type1` is valid. ADC2 does not support DMA mode.
*/
typedef struct {
```

(continues on next page)

```c
/**
 * @brief LEDC timer configuration.
 */
typedef struct {
ledc_mode_t speed_mode;                /*!< LEDC speed mode. */
ledc_timer_bit_t duty_resolution;      /*!< LEDC channel duty resolution. */
uint32_t freq_hz;                      /*!< LEDC timer frequency. Measurement unit: Hz. */
ledc_clk_cfg_t clk_cfg;                /*!< LEDC clock. */
                                        /*!< @note  For ESP32 and ESP32-S2, each timer can have a
                                        independent clock source. For other chips, all timers use one
                                        collective clock source. */
} ledc_timer_config_t;
```

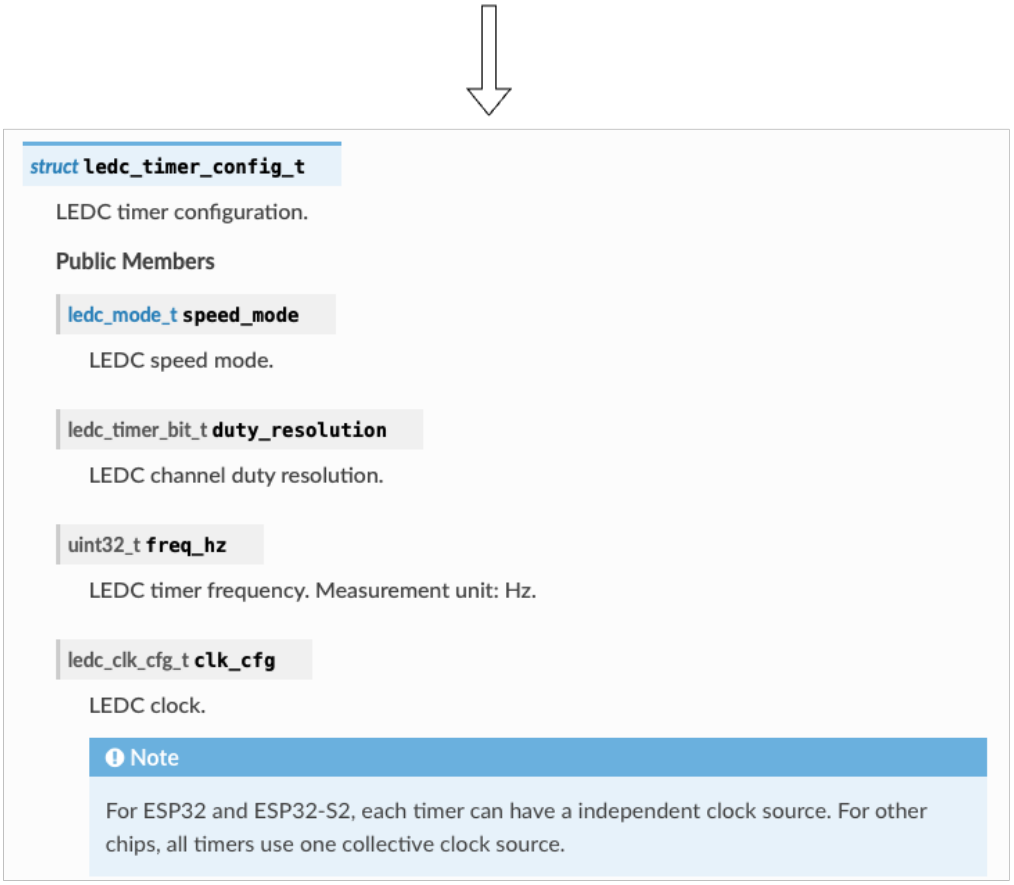


Fig. 8: Rendered Result - In-Body Comments with Comment Blocks (Click to enlarge)

(continued from previous page)

```
    union {
        struct {
            uint16_t data:     12;  /*!<ADC real output data info. Resolution: 12␣
↪bit. */
            uint16_t channel:   4;  /*!<ADC channel index info. */
        } type1;                    /*!<ADC type1. */
        struct {
            uint16_t data:     11;  /*!<ADC real output data info Resolution: 11␣
↪bit. */
            uint16_t channel:   4;  /*!<ADC channel index info. For ESP32-S2:
                                        If (channel < `ADC_CHANNEL_MAX`), The data␣
↪is valid.
                                        If (channel > `ADC_CHANNEL_MAX`), The data␣
↪is invalid. */
            uint16_t unit:      1;  /*!<ADC unit index info. 0: ADC1; 1: ADC2.  */
        } type2;                    /*!<When the configured output format is 11␣
↪bit.*/
        uint16_t val;               /*!<Raw data value. */
    };
} adc_digi_output_data_t;
```

Alternatively, you can use if statements (`#if` and `#endif` directives in C language) together with macros defined in `*_caps.h` header files as shown in the following examples.

Note: Please note that some developers tend to read header files directly instead of API documentation. If statements would make header files hard to read, so they are less recommended.

Use an if statement to mark a target-specific function:

```
#if SOC_I2C_SUPPORT_SLAVE
/**
* @brief Write bytes to internal ringbuffer of the I2C slave data. When the TX␣
↪fifo empty, the ISR will
*        fill the hardware FIFO with the internal ringbuffer's data.
*        @note This function shall only be called in I2C slave mode.
*
* @param i2c_num I2C port number.
* @param data Bytes to write into internal buffer.
* @param size Size, in bytes, of `data` buffer.
* @param ticks_to_wait Maximum ticks to wait.
*
* @return
*     - ESP_FAIL (-1): Parameter error.
*     - Other (>=0): The number of data bytes pushed to the I2C slave buffer.
*/
int i2c_slave_write_buffer(i2c_port_t i2c_num, const uint8_t *data, int size,␣
↪TickType_t ticks_to_wait);
#endif // SOC_I2C_SUPPORT_SLAVE
```

Use an if statement to mark a target-specific `enum`:

```
/**
* @brief I2C port number, can be `I2C_NUM_0` ~ (`I2C_NUM_MAX` - 1).
*/
typedef enum {
    I2C_NUM_0 = 0, /*!< I2C port 0. */
#if SOC_I2C_NUM >= 2
    I2C_NUM_1, /*!< I2C port 1. */
#endif
```

(continues on next page)

(continued from previous page)

```
    I2C_NUM_MAX, /*!< Maximum I2C ports. */
} i2c_port_t;
```

Style

When preparing the API descriptions, follow the style below for consistency:

- The maximum line length is 120 characters for better code readability, as described in Espressif IoT Development Framework Style Guide
- If descriptions in combination with code are more than 120 characters, manually break lines, or consider if the descriptions better fit in the main text (namely the `.rst` files)
- Capitalize the first word of every sentence segment or sentence
- **Use a period . at the end of:**
  - Complete sentences
  - Sentence fragments
  - Bulleted or numbered list items
- Use **bullet points** if there are 2 or more returned values
- Use **:** between a returned value and its meaning
- Between parameters and parameter meanings, do not add any punctuation marks such as **-** and **:**

The example below shows how to follow above style after >>>:

```
1  /**
2  *
3  * @brief           This function is called to send wifi connection report
   ↪ >>> Should add a ending period "." for complete sentences
4  * @param opmode :  Wi-Fi opmode.
   ↪ >>> Should delete the colon ":" between parameter's name and perameters' meaning
5  * @param sta_conn_state     station is already in connection or not.
   ↪ >>> Should be capitalized
6  * @param softap_conn_num    Softap connection number.
7  * @param extra_info         Extra information, such as sta_ssid, softap_ssid and
   ↪etc.
8  *
9  * @return          ESP_OK - Success, other - Failed
   ↪ >>> Values should be listed using bullet points, and "-" should be changed to ":
   ↪"
10 *
11 */
12 esp_err_t esp_blufi_send_wifi_conn_report(wifi_mode_t opmode, esp_blufi_sta_conn_
   ↪state_t sta_conn_state, uint8_t softap_conn_num, esp_blufi_extra_info_t *extra_
   ↪info);
```

Above examples can be updated as follows in line with the rules (note that the returned error codes and their descriptions in line 10 can be more specific):

```
1  /**
2  *
3  * @brief Send Wi-Fi connection report.
4  * @param opmode Wi-Fi operation mode.
5  * @param sta_conn_state Whether station is connected or not.
6  * @param softap_conn_num SoftAP connection number.
7  * @param extra_info Extra information, such as sta_ssid, softap_ssid and etc.
8  *
9  * @return
10 *       - ESP_OK: Done.
11 *       - Other error code: Failed.
12 *
13 */
```

(continues on next page)

(continued from previous page)

```
14  esp_err_t esp_blufi_send_wifi_conn_report(wifi_mode_t opmode, esp_blufi_sta_conn_
    ↪state_t sta_conn_state, uint8_t softap_conn_num, esp_blufi_extra_info_t *extra_
    ↪info);
```

esp_err_t **esp_blufi_send_wifi_conn_report**(wifi_mode_t opmode, esp_blufi_sta_conn_state_t sta_conn_state, uint8_t softap_conn_num, esp_blufi_extra_info_t *extra_info)

Send Wi-Fi connection report.

Parameters:
- **opmode** – Wi-Fi operation mode.
- **sta_conn_state** – Whether station is connected or not.
- **softap_conn_num** – SoftAP connection number.
- **extra_info** – Extra information, such as sta_ssid, softap_ssid and etc.

Returns:
- ESP_OK: Done.
- Other error code: Failed.

2.9.2 Generate and Include API Descriptions

Enabling the Doxygen Extension

API generation is handled by the run_doxygen.py extension. To enable it, add it to your project's `conf_common.py`:

```
extensions = ['esp_docs.esp_extensions.run_doxygen']
```

Setting up a Doxyfile

`Doxyfile` is the must-have Doxygen configuration file for automatic API generation. It must be placed in your `docs/` directory. At minimum it must list the header files to document and enable XML output:

```
INPUT = ../src/api/my_api.h \
        ../src/api/my_other_api.h

RECURSIVE = YES
GENERATE_XML = YES
XML_OUTPUT = xml
```

Note: `GENERATE_XML = YES` is required —ESP-Docs reads Doxygen's XML output, not HTML.

Note: Target-specific header files may be placed in a separate `Doxyfile`. For example, Doxyfile_esp32 is provided to generate ESP32-specific API descriptions in ESP-IDF.

For a more complete reference, see the Doxyfile in the doxygen example or the ESP-IDF Doxyfile.

Including Generated API in RST Files

When you run the build (see *Building Documentation Locally*), run_doxygen.py generates `.inc` files from the header files listed in `Doxyfile` and places them in `_build/<language>/<target>/inc/`. The file name matches

the header —`my_api.h` produces `my_api.inc`.

To embed them in an `.rst` page, use the `include-build-file::` directive:

```
API Reference
-------------

.. include-build-file:: inc/my_api.inc
```

Linking to API Symbols in Prose

Cross-references to documented functions, types, structs, enumerations, and macros can be inserted anywhere in `.rst` files using Breathe's roles. For a full list of roles and examples, see *Linking to Classes, Functions, Enumerations, etc*.

Common examples:

```
Call :cpp:func:`my_function` to initialise the driver.
The :cpp:type:`my_config_t` struct controls its behaviour.
Check the return value against :c:macro:`MY_ERR_OK`.
```

Note: Use `:cpp:` roles for C++ symbols and C symbols defined in a C++ project. Use `:c:macro:` for preprocessor macros (`#define`), as the C++ domain has no macro role.

Hover Tooltips for API Links

ESP-Docs includes an optional extension that shows the full API description as an inline tooltip when a reader hovers over any cross-reference link, without leaving the page.

To enable hover tooltips, add the following to your project's `conf_common.py`:

```
esp_hover_api_enable = True
```

No other configuration is needed. The tooltip content is fetched from the linked API page and cached in the browser, so it works across pages as well as within the same page.

Tooltips are also **recursive**: if the tooltip content itself contains cross-reference links, hovering over them opens a second tooltip.

2.9.3 Example

For reference, you may navigate to the doxygen folder, and check the header files stored in the `src/api` subfolder.

To see the API descriptions in HTML, please run `build_example.sh`.

2.10 Formatting Documents for Translation

Espressif aims to provide well-formatted and up-to-date English and Chinese documents for customers. To keep English and Chinese versions always in sync, writers are encouraged to update both versions at the same time. However, the documents of one language version may lag behind the other sometimes since some writers, who are non-bilingual, can only update one language version. Therefore, the Documentation Team will provide translation for these documents as soon as possible as the lag-behind documents will be misleading for customers.

To make it easier to update both versions for writers and facilitate the translation process for the Documentation Team, writers and translators should follow the guidelines below when writing and updating documentation.

2.10.1 One Line per Paragraph

One paragraph should be written in one line. Breaking lines to enhance readability is only suitable for writing codes. In the documentation, please do not break lines like the below:

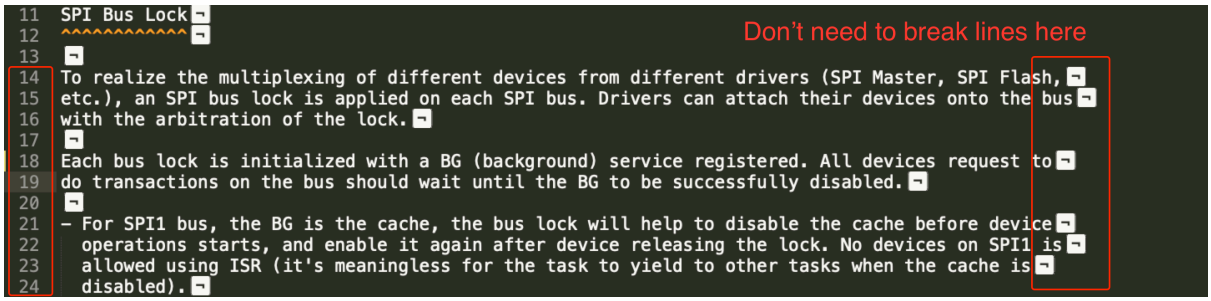


Fig. 9: Line breaks within the same paragraph - not recommended (click to enlarge)

To make the document easier to read, it is recommended to place an empty line to separate the paragraph.

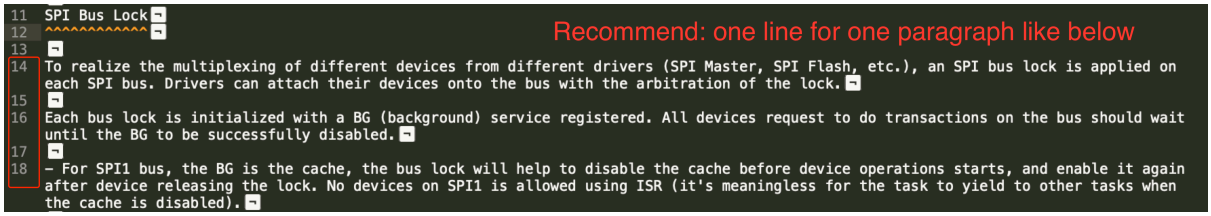


Fig. 10: One line per paragraph - recommended (click to enlarge)

2.10.2 Line Number Consistency

Make the line numbers of English and Chinese documents consistent. For example, as shown below, the title of the 9th line in the English version should also be placed on the 9th line in the Chinese version. Other lines follow the same rule.



Fig. 11: Keep the line number for English and Chinese documents consistent (click to enlarge)

This approach could be beneficial in the following ways:

- For non-bilingual writers, they only need to update the same line in the corresponding Chinese or English document when updating documents.
- For translators, if documents are updated in English, then translators can quickly locate where to update in the corresponding Chinese document later.
- By comparing the total number of lines in English and Chinese documents, Documentation Team can quickly find out which document lags behind the other version and provide translation soon.

Note: This document only describes formatting rules that facilitate translation. For other formatting rules, see Espressif Manual of Style.

2.11 Using Spellcheckers in VS Code

Using spellcheckers in your code editors is recommended to ensure high-quality and error-free documentation. This helps reduce spelling mistakes and grammatical errors, improving readability and professionalism.

2.11.1 Why Use Spellcheckers?

Our documentation is collaboratively maintained by multiple contributors, many of whom prioritize technical accuracy over grammatical precision. This is particularly common in documentation embedded in header files, where grammar reviews are often skipped during technical revisions.

As a result, spelling and grammatical errors may go unnoticed until they are reported long after publication. Addressing these issues requires considerable manual effort from the documentation team and may still leave some errors unresolved. This not only undermines the overall clarity and professionalism of the documentation, but also leads to unnecessary resource consumption.

2.11.2 Using Code Spell Checker in VS Code

Visual Studio Code (VS Code) provides a wide range of extensions to help detect spelling and grammar issues during documentation writing. These tools make it easier to identify and resolve language problems early in the drafting process, ensuring higher-quality output. One popular lightweight choice is the **Code Spell Checker** extension, which works well with code and documents.

To install and configure Code Spell Checker in VS Code:

1. Open VS Code.
2. Go to the `Extensions` view by clicking the square icon on the sidebar.
3. Search for **Code Spell Checker** and click **Install**.

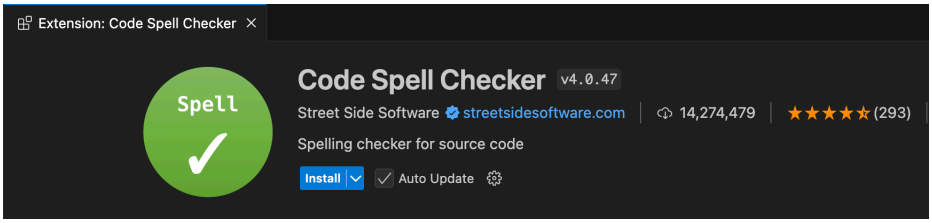


Fig. 12: Install Code Spell Checker (Click to enlarge)

4. Once installed, the extension will start underlining potential spelling mistakes.

Note: Code Spell Checker includes a variety of built-in dictionaries covering general English (US and GB), software terminology, multiple programming languages, etc. These dictionaries help reduce false positives

when checking technical content. You can also define custom dictionaries to include domain-specific terms or abbreviations. For more features and configuration options, refer to the official documentation.

Chapter 3

Building Documentation

3.1 Previewing Documentation inside Your Text Editor

This section describes how to preview your rst documentation inside your text editor on your PC.

reStructuredText documents are text files, and can be edited with any text editor. Inside these text editors, there are plenty of extensions or plugins you can use to achieve a live preview.

This approach is good for achieving a real-time live preview while you write because it's simple and fast, but it will only render "base" rst content without any esp-docs specific features. The styles of rendering really depend on the extensions or plugins you use, and you may face issues such as broken links. If you want to preview your rst documentation rendered in exactly the same style as if it is on-line with all the correct reference, go to Section *Building Documentation locally on Your OS*.

In this section, we will use Visual Studio Code and Sublime Text as examples.

3.1.1 Visual Studio Code

1. Open your **VS Code** instance, and navigate to `Extensions`.
2. In the top search bar, type in keywords such as "preview" or "rst preview".
3. Install the previewer extension of your choice (for example, Preview), and follow the instruction inside the extension to enable a live preview.

3.1.2 Sublime Text

1. Open your **Sublime Text** instance, go to `Tools`, and click `Install Package Control` from the drop-down menu.
2. After step 1, go to `Tools` again, and click on `Command Palette...`.
3. In the top search bar, type "Install" and select `Package Control: Install Package`.
4. In the top search bar, type in keywords such as "preview" or "rst preview".
5. Install the previewer plugin (for example, OmniMarkupPreviewer) of your choice, and follow the instruction inside the plugin to enable a live preview.

3.2 Building Documentation Locally

The purpose of this description is to provide a summary on how to build documentation locally using ESP-Docs.

3.2.1 Building HTML Locally on Your PC

ESP-Docs allows you to build your rst documentation into HTML pages on you local computer with the same style exactly as how it will be rendered on the server. In this way, you can:

- Catch and fix any potential build errors (due to markup syntax, incorrect links, labels, missing images, etc.) early, instead of waiting on CI errors.
- Of course, have a peek of your final documentation early.

If you just want to roughly preview your rst files while your write and don't care too much about styles and broken links at this moment, then go to Section *Previewing Documentation inside Your Text Editor*.

Installing Dependencies

In order to build documentation locally on your PC, you need to install the following prerequisites:

1. ESP-Docs - https://github.com/espressif/esp-docs
2. CairoSVG - https://cairosvg.org/documentation/
3. Doxygen (only needed when generating API documentation from header files)- http://doxygen.nl

For building the ESP-IDF documentation, see its own Building Documentation section instead.

Note: Docs building now supports Python 3 only. Python 2 installations will not work.

Note: If you are a Windows user or simply want to use a Docker container, then go directly to *Using a Docker Container* at the end of this section.

ESP-Docs All applications needed are Python packages, and you can install them in one step as follows:

```
pip install --user esp-docs
```

This will pull in all the necessary dependencies such as Sphinx, Breathe, etc.

Note: The installed esp-docs may not be added to your PATH environment variable yet at this moment. To make this tool usable from the command line, add the bin folder where it is installed to your PATH variable by running `export PATH=path_to_bin_folder:$PATH` in your terminal.

To get this `path_to_bin_folder`, try entering `pip uninstall esp-docs`, you will see something like:

```
Found existing installation: esp-docs 1.3.0
Uninstalling esp-docs-1.3.0:
Would remove:
/Users/dummy/Library/Python/3.10/bin/build-docs
/Users/dummy/Library/Python/3.10/bin/deploy-docs
/Users/dummy/Library/Python/3.10/lib/python/site-packages/esp_docs-1.3.0.dist-info/
↪*
/Users/dummy/Library/Python/3.10/lib/python/site-packages/esp_docs/*
```

The path before `build-docs` is your bin path. However, this configuration is only effective in the current terminal session. You need to add PATH again once you reopen your terminal.

Therefore, if you plan to use esp-docs frequently, consider adding `export PATH="path_to_bin_folder:$PATH"` to your shell profile files, such as `.zprofile`, then refresh the configuration by restarting your terminal or by running `source [path_to_profile_file]`, for example `source ~/.zprofile`. Afterwards, you can use esp-docs in any terminal session anytime.

CairoSVG CairoSVG is an SVG 1.1 to PNG, PDF, PS and SVG converter. You can install it as follows:

```
pip3 install cairosvg
```

If you have issues, please check out CairoSVG documentation.

Doxygen Installation of Doxygen is OS dependent:

Linux

```
sudo apt-get install doxygen
```

MacOS

```
brew install doxygen
```

After these steps, you should be able to build HTML pages on your PC already. To see the details, go to *Building HTML Pages*.

Building HTML Pages

After completing the above-mentioned preparation, you can navigate to your docs folders (`cd ~/$PROJECT_PATH/docs`), then build HTML pages locally with the *build-docs* command.

Note: If `$PROJECT_PATH` is not the parent to the `docs` folder, then please specify the project path with `--project-path` option. This is only required when you want to build API documentation.

- **Build HTML pages in projects that do not support targets**

  ```
  build-docs build
  ```

- **Build HTML pages for a single language**

  ```
  build-docs -l en
  ```

  Choices for language (`-l`) are `en` and `zh_CN`.

- **Build HTML pages for a single target**

  ```
  build-docs -t esp32
  ```

  Choices for target (`-t`) are any supported chip targets (for example `esp32` and `esp32s2`).

- **Build HTML pages for a single language and target combination only**

  ```
  build-docs -t esp32 -l en
  ```

  Choices for language (`-l`) are `en` and `zh_CN`, and for target (`-t`) are any supported chip targets (for example `esp32` and `esp32s2`).

- **Build HTML pages excluding Doxygen-generated API documentation, which drastically reduces build time**

  ```
  build-docs -f
  ```

  To keep API references for specific modified headers during a fast build, pass them via `--modified-files`:

  ```
  build-docs -f --modified-files components/driver/include/driver/uart.h
  ```

  For now, `--modified-files` only affects fast builds and expects paths relative to the project path. It also accepts a single semicolon-delimited string, which is useful in CI variables.
  or by setting the environment variable `DOCS_FAST_BUILD`. To set an environment variable, go to your project's **Settings > CI/CD** and expand the **Variables** section. Select **Add variable** and fill in the

details for your variables. For more information on how to add a variable to a project, see the GitLab documentation.

Note: To set an environment variable, you need to be a project admin or contact the project admin for help.

Note: The time it takes to build is mainly determined by the amount of Doxygen API included. This is the reason why build with option *-f* for fast build is much faster.

- **Build HTML pages for a single document or a subset of documentation**

  For a single document

```
build-docs -t esp32 -l en -i api-reference/peripherals/can.rst
```

  For a subset of documentation by listing all of them

```
build-docs -t esp32 -l en -i api-reference/peripherals/can.rst api-
↪reference/peripherals/adc.rst
```

  For a subset of documentation by using wildcards:

```
build-docs -l en -t esp32 -i api-reference/peripherals/* build
```

  Note: Note that when you only build a single document or a subset of documentation. The HTML output won't be perfect, i.e. it will not build a proper index that lists all the documents, and any references to documents that are not built will result in warnings.

- **To see the complete list of options:**

```
build-docs --help
```

Checking Output

The built HTML pages will be placed in `_build/<language>/<target>/html` folder.

Note: There are a couple of spurious warnings that cannot be resolved without doing updates to the Sphinx or Doxygen source code. For such specific cases, respective warnings can be documented in `docs/sphinx-known-warnings.txt` and `docs/doxygen-known-warnings.txt` files, which are checked during the build process to ignore these spurious warnings.

3.2.2 Building PDF Documentation Locally on Your PC

ESP-Docs also allows you to build your rst files into PDF files on your local PC. To do this, on top of all the packages and steps described in *Building HTML Locally on Your PC*, you also need to complete some additional steps.

Installing Dependencies

1. Install the following LaTeX packages:
   - latexmk
   - texlive-latex-recommended
   - texlive-fonts-recommended
   - texlive-xetex
2. Install the following fonts:
   - Freefont Serif, Sans and Mono OpenType fonts, available as the package `fonts-freefont-otf` on Ubuntu

- Lmodern, available as the package `fonts-lmodern` on Ubuntu
- Fandol, can be downloaded from ctan.org archive

Note: Another alternative is to simply install TeX Live, which contains all LaTeX packages and fonts required to build PDF files. However, it may take you hours to install.

Note: If you are a Windows user or simply want to use a Docker container, then go directly to *Using a Docker Container* at the end of this section.

After these steps, you should be able to build PDF files on your PC already. To see the details, go to *Building PDF Documents*.

Building PDF Documents

Now you can navigate to your docs folders (`cd ~/$PROJECT_PATH/docs`), then build PDF documents with the same *build-docs* command, but with the `-bs latex` option.

- **Build PDF for "generic" documentation that doesn't contain a target**

```
build-docs -bs latex
```

- **Build PDF for a single language and target combination only**

```
build-docs -bs latex -t esp32 -l en
```

Choices for language (`-l`) are `en` and `zh_CN`, and for target (`-t`) are any supported chip targets (for example `esp32` and `esp32s2`).

- **Or alternatively build both HTML and PDF:**

```
build-docs -bs html latex -l en -t esp32
```

Checking Output

The built LaTeX and PDF files will be placed in `_build/<language>/<target>/latex/build` folder.

Note: There are a couple of spurious warnings that cannot be resolved without doing updates to the Sphinx or Doxygen source code. For such specific cases, respective warnings can be documented in `docs/sphinx-known-warnings.txt` and `docs/doxygen-known-warnings.txt` files, which are checked during the build process to ignore these spurious warnings.

3.2.3 Using a Docker Container

A Docker container image is a lightweight, standalone, executable package of software that can be prepared to include everything needed to run an application: code, runtime, system tools, system libraries, and in our case, to build the documentation locally. This approach saves you the trouble to configure your PC.

To build documentation locally in a Docker container, complete the steps below:

1. Navigate to your project folder. For example `cd esp/esp-docs`.
2. Create a container for your project using the image provided by Espressif.

```
docker run -v $PWD:/esp-docs -w /esp-docs -it gitlab.espressif.cn:5050/ci/
↪images/idf-v5.5-docs:1
```

3. Configure your container by running `pip install -U esp-docs`.

After these steps, you can build docs following the instructions described in Sections *Building HTML Pages* and *Building PDF Documents*.

3.2.4 Troubleshooting

If you experience any warning or error when building documentation locally:

- Check *Troubleshooting Build Errors and Warnings*;
- Or contact us by submitting a documentation feedback.

Chapter 4

Configuring ESP-Docs Projects

4.1 Integrating ESP-Docs into Your Project

This document describes how to integrate *ESP-Docs* into your project to continuously build and deploy your documentation to a server, such as Espressif's server `docs.espressif.com` (recommended for Espressif software documentation).

While performing the steps in this document, you can always refer to the documentation that has already been deployed to Espressif's server as examples, such as ESP-IDF Programming Guide, ESP-AT User Guide, esptool.py Documentation, and ESP-Docs User Guide.

The process to integrate ESP-Docs can be broken down into the following steps:

- *Get Familiar with the Documentation Folder*
- *Prepare a Documentation Folder*
- *Update Build Configuration Files*
- *Update CI Configuration File*
  - *Build Documentation*
  - *Deploy Documentation*
- *What's Next?*

4.1.1 Get Familiar with the Documentation Folder

The contents of the basic documentation folder are described below to provide more details about the folder structure and the function of each file. Your folder might look slightly different, but being familiar with these building blocks will help you better understand the following steps in this document.

- `_static`: contains graphics files, sources of diagrams, attachments not shown directly in the documentation (e.g., schematics) as well as other resources, such as font files.
  - `docs_version.js`: configures target and version information displayed in HTML layout, such as the target and language selector in the top-left corner of ESP-IDF Programming Guide.
  - `periph_timing.json`: sample figure in JSON format.
- `en`: English language folder that contains English documents and a build configuration file.
  - `conf.py`: build configuration file that contains configuration information specific to the English documents, such as the English copyright information.
  - `index.rst`: English homepage that defines documentation structure with a table of contents tree (toctree). See Defining document structure for more information.
  - `subpage.rst`: sample subpage of `index.rst`.
- `zh_CN`: the same as `en` but for the Simplified Chinese language.
  - `conf.py`: the same as `en/conf.py` but for the Chinese documents.
  - `index.rst`: the same as `en/index.rst` but for the Chinese documents.

  - `subpage.rst`: the same as `en/subpage.rst` but in Chinese.
- `README.md`: introduction to the `docs` folder.
- `build_example.sh`: contains the command to simplify building this sample documentation.
- `conf_common.py`: contains the build configuration information common to both English and Chinese documents. The contents of this file are imported during the building process for each language to the standard Sphinx configuration file `conf.py` located in respective language folders (e.g. `docs/en`, `docs/zh_CN`). See Sphinx Configuration for more information.
- `requirements.txt`: package dependencies and their versions for building documentation, such as ESP-Docs.

4.1.2 Prepare a Documentation Folder

1. Copy one of the following sample documentation folders to the root directory of your project depending on whether the project needs support for target, version, or building API documentation from header files:

Doc Folder	Target	Version	API Doc
basic	Y	Y	–
doxygen	Y	Y	Y
test/build_tests/no_target	–	Y	Y
test/build_tests/no_version_i	–	–	Y
test/build_tests/target_only	Y	–	Y

2. Rename the folder to `docs`.
3. Delete the `build_example.sh` file (if there is one).
4. (Optional) Go to `docs/requirements.txt` and change the ESP-Docs version as needed. ESP-Docs follows the semantic versioning scheme. For features supported by each release, please see release history.

4.1.3 Update Build Configuration Files

Build configuration files are where you set the variables that are affecting the final documentation output built with ESP-Docs. As mentioned in *Get Familiar with the Documentation Folder*, there should be two types of configuration files in each project:

- `conf_common.py`
- `en/conf.py` and `zh_CN/conf.py`

The configuration files in the sample documentation folder configure how to build the **sample** documentation instead of your documentation, so you need to reconfigure a few variables for your documentation.

1. In `conf_common.py`, reconfigure some of the following variables based on your needs:

- `languages`: supported languages, such as `en` and `zh_CN`. It must be set to at least one language element, namely the current project's language.
- `idf_targets`: project target used as a URL slug, such as `esp32` in `https://docs.espressif.com/projects/esp-idf/en/latest/esp32/`. The variable is optional, but you should set either both this variable and `idf_target` or neither. For more information about `idf_target`, see *Build System Integration* .
- `extensions`: extensions that add more functionalities to ESP-Docs, such as `sphinx_copybutton` and `sphinxcontrib.wavedrom`.
- `github_repo`: GitHub repository to which the links generated by link_roles.py point.
- `html_context['github_user']`: GitHub user name used by sphinx_idf_theme.
- `html_context['github_repo']`: GitHub repo name used by sphinx_idf_theme.
- `html_static_path`: path to the `_static` folder.
- `project_slug`: short name of the project as a URL slug, such as `esp-docs`.
- `versions_url`: URL from which to download the `versions.js` file. If it is specified as a relative URL, such as `_static/docs_version.js`, the file will be downloaded relative to the `HTML` root folder.

- `pdf_file_prefix`: PDF filename prefix used for generating the link to download the PDF together with the target and version name.

2. In `en/conf.py` and `zh_CN/conf.py`, reconfigure some of the following variables based on your needs:

- `project`: name of your documentation in HTML, such as ESP-IDF Programming Guide, ESP-AT User Guide.
- `copyright`: copyright statement.
- `pdf_title`: name of your documentation in PDF.
- `language`: language for content autogenerated by ESP-Docs.

4.1.4 Update CI Configuration File

Note: The following descriptions assume you are using Gitlab CI for building documentation and deploying it to `www.espressif.com`, and will have to be tweaked if you are running something else for CI/CD.

The GitLab CI configuration file, `.gitlab-ci.yml`, is where you add jobs to enable the automatic and continuous building and deploying of your documentation to the `www.espressif.com` server.

In the `.gitlab-ci.yml` of your project, do the steps given below. For examples, please refer to esp-docs/.gitlab-ci.yml and esp-idf/.gitlab/ci/docs.yml.

Build Documentation

1. In your `.gitlab-ci.yml`, add a template that installs the documentation packages in `docs/requirements.txt` and selects the Docker image. The HTML and PDF build jobs share this template.

```
.build_template:
  stage: build
  image: $ESP_DOCS_ENV_IMAGE
  variables:
    DOCS_DIR: $CI_PROJECT_DIR/docs
  before_script:
    - pip install -r docs/requirements.txt
```

Note: If you define `ESP_DOCS_ENV_IMAGE` once in the top-level `variables` section of `.gitlab-ci.yml`, the build jobs and the deploy jobs then share `image: $ESP_DOCS_ENV_IMAGE`. You can reuse the image used by ESP-IDF, such as `$CI_REGISTRY/ci/images/idf-v5.5-docs:1`. For the latest version of this image, go to Documentation Team Site > Section ESP-Docs User Guide > esp-idf-doc-env image.
The following snippet from .gitlab-ci.yml shows that global variable:

```
8  variables:
9    # Versioned esp-idf docs image to use for all document building jobs
10   ESP_DOCS_ENV_IMAGE: "$CI_REGISTRY/ci/images/idf-v5.5-docs:1"
```

2. Add the HTML and PDF build jobs to your `.gitlab-ci.yml`. The snippets below are from .gitlab-ci.yml.

```
96  build_esp_docs_html:
97    extends:
98      - .build_template
99    variables:
100     DOCS_DIR: $CI_PROJECT_DIR/docs
101   artifacts:
102     when: always
103     paths:
104       - $DOCS_DIR/_build/*/*/html/*
```

(continues on next page)

(continued from previous page)

```
      - $DOCS_DIR/_build/*/*/*.txt
    expire_in: 4 days
  script:
    - cd $DOCS_DIR
    - build-docs --skip-reqs-check -l $DOCLANG
    - echo "ESP-Docs documentation preview available at $CI_JOB_URL/artifacts/
↪file/docs/_build/$DOCLANG/generic/html/index.html"
  parallel:
    matrix:
      - DOCLANG: ["en"]

build_esp_docs_pdf:
  extends:
    - .build_template
  variables:
    DOCS_DIR: $CI_PROJECT_DIR/docs
  artifacts:
    when: always
    paths:
      - $DOCS_DIR/_build/*/*/latex/*
      - $DOCS_DIR/_build/*/*/*.txt
    expire_in: 4 days
  script:
    - cd $DOCS_DIR
    - build-docs --skip-reqs-check -l $DOCLANG  -bs latex
  parallel:
    matrix:
      - DOCLANG: ["en"]
```

3. Change these values in your `.gitlab-ci.yml` when your project differs from the snippets above:
   - `DOCS_DIR`: the directory the job enters before it runs `build-docs`. The snippets above set `$CI_PROJECT_DIR/docs`. `build-docs` uses the directory it is run from. It does not read `DOCS_DIR`.
   - `ESP_DOCS_LATEST_BRANCH_NAME`: the branch to be built and published as `latest`. If not configured, the default value is `master`. Set this variable when the branch published as `latest` has another name, such as `main`.

Deploy Documentation

1. In your `.gitlab-ci.yml`, add a deploy template. The template uses the same `ESP_DOCS_ENV_IMAGE` from the top-level `variables` section, so the deploy jobs use the same image as the build jobs.

```
.deploy_docs_template:
  stage: deploy
  image: $ESP_DOCS_ENV_IMAGE
  needs:
    - build_esp_docs_html
    - build_esp_docs_pdf
  script:
    - add_doc_server_ssh_keys $DOCS_DEPLOY_PRIVATEKEY $DOCS_DEPLOY_SERVER
↪$DOCS_DEPLOY_SERVER_USER
    - export GIT_VER=$(git describe --abbrev=10 --always)
    - pip install -r ${CI_PROJECT_DIR}/docs/requirements.txt
    - deploy-docs
```

2. Add jobs to deploy docs to the preview server when the MR is open, and to the production server when the MR is merged.

```
deploy_docs_preview:
  extends:
```

(continues on next page)

(continued from previous page)

```
    - .deploy_docs_template
  rules:
    - if: $CI_PIPELINE_SOURCE == 'merge_request_event'
      changes:
        - "docs/**/*"
  variables:
    TYPE: "preview"
    DOCS_BUILD_DIR: "${CI_PROJECT_DIR}/docs/_build/"
    DOCS_DEPLOY_PRIVATEKEY: "$DOCS_PREVIEW_DEPLOY_KEY"
    DOCS_DEPLOY_SERVER: "$DOCS_PREVIEW_SERVER"
    DOCS_DEPLOY_SERVER_USER: "$DOCS_PREVIEW_SERVER_USER"
    DOCS_DEPLOY_PATH: "$DOCS_PREVIEW_PATH"
    DOCS_DEPLOY_URL_BASE: "$DOCS_PREVIEW_SERVER_URL/your-project-name"

deploy_docs_production:
  extends:
    - .deploy_docs_template
  rules:
    - if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH
      changes:
        - "docs/**/*"
  variables:
    TYPE: "production"
    DOCS_BUILD_DIR: "${CI_PROJECT_DIR}/docs/_build/"
    DOCS_DEPLOY_PRIVATEKEY: "$DOCS_PROD_DEPLOY_KEY"
    DOCS_DEPLOY_SERVER: "$DOCS_PROD_SERVER"
    DOCS_DEPLOY_SERVER_USER: "$DOCS_PROD_SERVER_USER"
    DOCS_DEPLOY_PATH: "$DOCS_PROD_PATH"
    DOCS_DEPLOY_URL_BASE: "https://docs.espressif.com/projects/your-project-
↪name"
```

Note: If your project is hosted on GitLab and the updates made in GitLab later are synchronized to GitHub, run the deploy job after the job that synchronizes your repository to GitHub. If synchronization to GitHub fails, links in the documentation that refer to the GitHub project do not work.

3. Change these values in your `.gitlab-ci.yml` when your project differs from the snippets above:
   - `DOCS_BUILD_DIR`: directory that contains the built HTML and PDF files. The snippets above set `${CI_PROJECT_DIR}/docs/_build/`.
   - `DOCS_DEPLOY_URL_BASE`: base URL printed after deployment. The preview job uses `$DOCS_PREVIEW_SERVER_URL/<project-path>`. The production job uses `https://docs.espressif.com/projects/<project-name>`.
   - `DEPLOY_STABLE`: optionally set this to `1` on the production job when a stable release should also be published under `stable`. For example of a stable release, see https://docs.espressif.com/projects/esp-idf/en/stable/esp32/index.html.
4. Add the SSH private key, server host, SSH user, remote directory, and preview URL as CI/CD variables:
   a. Find out who the server's admin is. To know who this person is, go to Documentation Team Site > Section ESP-Docs User Guide > Deploying documentation to docs.espressif.com.
   b. Ask the admin to create an SSH key for you and a directory for your documentation on the server.
   c. Go to your project's **Settings > CI/CD** and expand the **Variables** section. Select **Add variable** and fill in the details for your variables. For more information on how to add a variable to a project, see the GitLab documentation.

   `deploy_docs_preview` reads these variables:
   - `DOCS_PREVIEW_DEPLOY_KEY`: SSH private key. The job assigns it to `DOCS_DEPLOY_PRIVATEKEY`.
   - `DOCS_PREVIEW_SERVER`: server host. The job assigns it to `DOCS_DEPLOY_SERVER`.
   - `DOCS_PREVIEW_SERVER_USER`: SSH user. The job assigns it to `DOCS_DEPLOY_SERVER_USER`.
   - `DOCS_PREVIEW_PATH`: directory on the server. The job assigns it to `DOCS_DEPLOY_PATH`.
   - `DOCS_PREVIEW_SERVER_URL`: base URL of the preview server. The job uses it in

DOCS_DEPLOY_URL_BASE.

deploy_docs_production reads these variables:

- DOCS_PROD_DEPLOY_KEY: SSH private key. The job assigns it to DOCS_DEPLOY_PRIVATEKEY.
- DOCS_PROD_SERVER: server host, such as docs.espressif.com. The job assigns it to DOCS_DEPLOY_SERVER.
- DOCS_PROD_SERVER_USER: SSH user. The job assigns it to DOCS_DEPLOY_SERVER_USER.
- DOCS_PROD_PATH: directory on the server. The job assigns it to DOCS_DEPLOY_PATH.

4.1.5 What' s Next?

1. Push your changes to GitLab and check if the pipeline passes.
2. If yes, you can check the **Artifacts** to see what the built sample documentation looks like.
3. Now it is time to put your reST source files into the respective language folder and have them built and deployed!

4.2 Adding Extensions

Sometimes your project might need features that the ESP-Docs package does not support yet. In this case, you may add extensions either to ESP-Docs or to your project.

This document describes how to add third-party extensions and self-developed extensions to ESP-Docs or your project.

4.2.1 Where to Add?

An extension can be added either to ESP-Docs or to your project depending on the range of use.

If the extension might be used later in other projects integrating ESP-Docs, then add it to ESP-Docs. Otherwise, add it to your own project.

4.2.2 Third-Party Extensions

Third-party extensions are those extensions contributed by other users, for example extensions in LinuxDoc and sphinx-contrib libraries.

- To add a third-party extension to **ESP-Docs**, you should:
  1. Add the extension to src/esp_docs/conf_docs.py of ESP-Docs.

```
extensions = ['breathe',
              'sphinx.ext.todo',
              'sphinx_idf_theme',
              ]
```

  2. Add the extension and its version to setup.cfg of ESP-Docs.

```
install_requires =
        sphinx==4.5.0
        breathe==4.33.1
```

- To add a third-party extension to **your project**, you should:
  1. Add the extension to docs/conf_common.py of your project, or to the language specific configuration file docs/$lang$/conf.py of your project.

```
extensions = ['sphinx_copybutton',
              'sphinxcontrib.wavedrom',
              'linuxdoc.rstFlatTable',
              ]
```

  2. Add the extension and its version to docs/requirements.txt of your project.

```
linuxdoc==20210324
sphinx-design==0.2.0
```

4.2.3 Self-Developed Extensions

Self-developed extensions are those local extensions created by you and not provided as a package.

- To add a self-developed extension to **ESP-Docs**, you should:
  1. Place the extension in one of the following three folders of ESP-Docs:
     - src/esp_docs/generic_extensions, for extensions that do not rely on any Espressif-docs-specific behavior or configuration.
     - src/esp_docs/esp_extensions, for extensions that rely on any Espressif-docs-specific behavior or configuration.
     - src/esp_docs/idf_extensions, for extensions that rely on ESP-IDF-docs-specific behavior or configuration.

     For more information about self-developed extension types, you may refer to *Extensions Developed by Espressif*.
  2. Add the extension to src/esp_docs/conf_docs.py of ESP-Docs:

     ```
     extensions = ['esp_docs.generic_extensions.html_redirects',
                   'esp_docs.esp_extensions.include_build_file',
                   ]
     ```
- To add a self-developed extension to **your project**, you should:
  1. Place the extension in a proper folder of your project.

     For example, in the esp-iot-solution repository, the self-developed extension link-roles.py is placed in the docs folder.
  2. Add the extension to `docs/conf_common.py` of your project, or to the language specific configuration file `docs/$lang$/conf.py` of your project.

     ```
     extensions = ['link-roles',
                   ]
     ```

4.3 Adding the Link-check Function

Links play an important role in documentation with the function of directing users to supplementary information. However, broken links can not lead the user to where the author intended, which will harm reading experience and leave a bad impression on users.

Considering the great amount of time and labor to conduct manual check on link function throughout hundreds or even thousands of pages, a helpful tool is provided to automatically check links in the process of building the document. It helps identify and locate broken links.

This document describes how to integrate the link-check tool to your project pipeline and how to suppress falsely reported links.

4.3.1 How to Integrate the Link-check Function

- To integrate the link-check function to your project, add a CI/CD job to the GitLab YAML file of your project. It can be `.gitlab-ci.yml` or `.docs.yml`. Here is an example for your reference.

  ```
  check_doc_links:
      extends:
          - .build_docs_template
      only:
          - master
  ```

(continues on next page)

(continued from previous page)

```
    stage: post_deploy
    tags:
        - build
    artifacts:
        when: always
        paths:
            - docs/_build/*/*/*.txt
            - docs/_build/*/*/linkcheck/*.txt
        expire_in: 1 week
    allow_failure: true
    script:
        - cd docs
        - build-docs -l $DOCLANG linkcheck
    parallel:
        matrix:
            - DOCLANG: ["en", "zh_CN"]
```

  - `extends`: This command is to include the configuration from a predefined template in the `.yml` file. It helps reusing common configurations across multiple jobs. If you mention this template here, make sure you have defined it first. The following code provides an example on how to define the template:

    ```
    .build_docs_template:
        image: $ESP_IDF_DOC_ENV_IMAGE
        stage: build_doc
        tags:
            - build_docs
        dependencies: []
    ```

  - `only`: In this case, the link-check function will only run on the master branch. Note that this is not a must for projects without any links to GitHub Files.
  - `stage`: Specifies the stage of the pipeline where this job belongs. It is up to you which stage to do the link check. `post_deploy` is just one possible stage. Note that if the docs contain links to GitHub files then link-check should be done in the stage after the code is deployed to GitHub. Otherwise all GitHub links will be broken.
  - `tags`: Specifies the runner tags that this job should be picked up by. In this case, the job requires a runner with the tag *build*.
  - `artifacts`: Defines the artifacts to be collected from the job after it completes.
    - when: always means that the artifacts are collected regardless of whether the job succeeds or fails.
    - paths: specifies which files and directories to collect as artifacts.
    - expire_in: 1 week sets the expiration time for these artifacts, meaning they will be automatically deleted after one week.
  - `allow_failure: true`: Link-check might fail due circumstances outside of our control, e.g. a website being temporarily down or network outage. We use allow_failure so as not to mark a pipeline as failed just because the link-check failed.
  - `scripts`:
    - cd docs: changes the working directory to docs.
    - build-docs -l $DOCLANG linkcheck: runs a command to check links within the documentation. The -l $DOCLANG flag specifies the language for the documentation, with $DOCLANG being an environment variable set by the parallel matrix.
  - `parallel`: Defines a matrix of job configurations to run in parallel. In this case, it specifies that the build_docs command should be run twice: once for each value of DOCLANG (i.e., en for English and zh_CN for Simplified Chinese).
  - `image`: to define the Docker image or environment used for running jobs or pipelines.
- After automatic check, a report will be generated in `.txt` file.

Similar to the image above, the report file will generate detailed information, including the path of file that goes through link-check, the location of the link, link-check result and the complete link.

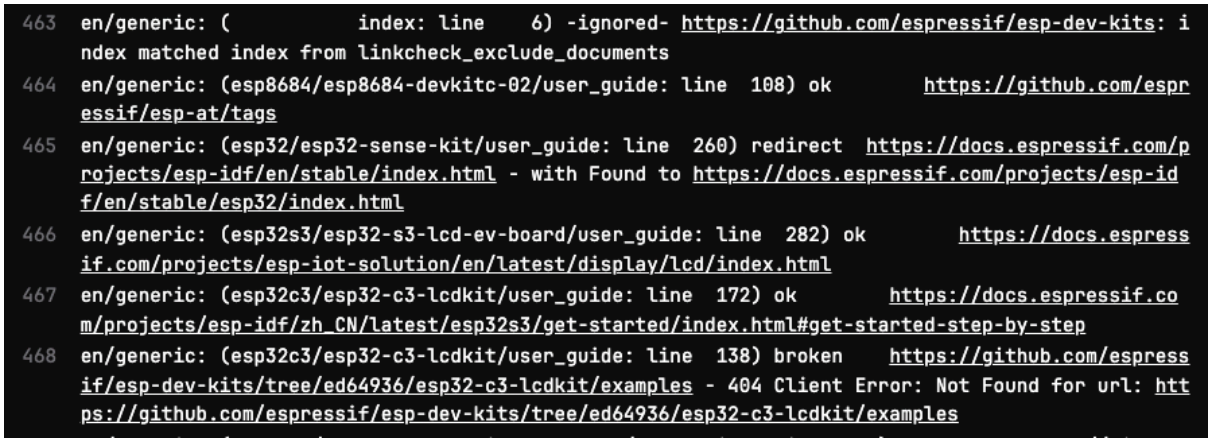


Fig. 1: Link-check Report

The result of the link-check can be classified in to 4 status:

- **ok:** the link passes the check.
- **redirect:** the broken link has been modified.
- **broken:** the link is identified as invalid.
- **ignored:** the link is excluded from check.

4.3.2 Note

It is possible that some links are reported as broken but when you open these links in the browser, they function well. These cases are called false positives. Common false positives are listed below. To exclude certain links from the scan, add the following code in `conf_common.py` file of your project.

1. Links in index documents

```
linkcheck_exclude_documents = ['index',  # several false positives due␣
↪to the way we link to different sections]
```

2. Links in documents located in a specific subdirectory (take the subdirectory named 'wifi_provisioning' as an example)

```
linkcheck_exclude_documents = ['api-reference/provisioning/wifi_
↪provisioning', # Fails due to `https://<mdns-hostname>.local`]
```

3. Github links with anchors

   Disable checking automatically generated anchors on *github.com*, such as anchors in reST/Markdown documents.

```
linkcheck_anchors = False
```

4. Links requesting too many times from github

   If certain links are consistently reported as broken due to rate limiting but are valid, you might need to handle them manually. You can exclude them from the scan by referring to previous instructions.

5. Links to unpublished documents (take ESP32-C2 Datasheet as an example)

```
linkcheck_ignore = ['https://www.espressif.com/sites/default/files/
↪documentation/esp32-c2_datasheet_en.pdf',  # Not published]
```

4.4 Adding the Language Link Checker

When maintaining documentation in multiple languages (e.g., English and Chinese), it is important to ensure that all pages have proper translation links. The language link checker automatically verifies that all `.rst` files included

in `toctree` directives have the appropriate `:link_to_translation:` role, which allows users to switch between language versions of the documentation.

This document describes how to integrate the language link checker into your project pipeline and how to configure it to handle special cases.

4.4.1 What the Language Link Checker Verifies

The language link checker verifies that:

- All `.rst` files included in `toctree` directives have a `:link_to_translation:` role
- The translation link points to the correct target language (e.g., English files should link to Chinese, and Chinese files should link to English)
- Each file has exactly one translation link (not multiple links)

There are cases where files should not be reported as errors even if they don't have translation links:

1. **Translation file doesn't exist**: If a file in one language doesn't have a corresponding translation file, it won't be reported as an error.
2. **Placeholder translation files**: If a translation file only contains an `.. include::` directive pointing to the original file, it is considered a placeholder and won't trigger an error. For example:

   ```
   .. include:: ../en/api-guides/build-system-v2.rst
   ```

3. **Files listed in warnings file**: You can specify files to ignore in `lang-linkcheck-warnings.txt`, similar to how `sphinx-known-warnings.txt` works for Sphinx warnings. For more information, refer to *Excluding Files from Check*.

4.4.2 How to Integrate the Language Link Checker

To integrate the language link checker into your project, add a CI/CD job to the GitLab YAML file of your project, for example `.gitlab-ci.yml`. Here is an example for your reference.

```
.. code-block::

    check_lang_links:
        rules:
            - if: $CI_PIPELINE_SOURCE == 'merge_request_event'
        stage: check
        image: $ESP_DOCS_ENV_IMAGE
        tags:
            - check_doc_links
        allow_failure: false
        script:
            - pip install esp-docs
            - cd docs
            - build-docs -l $DOCLANG lang-linkcheck
        parallel:
            matrix:
                - DOCLANG: ["en", "zh_CN"]
```

- `script`:
  - `pip install esp-docs`: installs the esp-docs package. If you have already installed the esp-docs package in the template job, you can extend the template job and skip this step. See gitlab-ci.yml for reference.
  - `cd docs`: changes the working directory to docs (where the `build-docs` command should be run).
  - `build-docs -l $DOCLANG lang-linkcheck`: runs the language link checker command. The `-l $DOCLANG` flag specifies the language for the documentation, with `$DOCLANG` being an environment variable set by the parallel matrix. You can also specify target using the `-t` flag.

For explanations of the other parameters, please refer to *Adding the Link-check Function*.

4.4.3 Understanding the Check Results

After running the language link checker, a report will be displayed in the console output. The report is organized into sections:

- **[EN] Files missing link to Chinese translation**: Lists English files that are missing or have incorrect translation links
- **[ZH_CN] Files missing link to English translation**: Lists Chinese files that are missing or have incorrect translation links
- **[PASSED] Files with correct translation links**: Lists files that passed the check (shown for debugging purposes)

Each error entry shows the file path and a description of the issue:

- `Missing :link_to_translation: directive`: The file has no translation link
- `Incorrect :link_to_translation: directive (found 'X', expected 'Y')`: The file links to the wrong language
- `Multiple :link_to_translation: directives found (expected exactly one)`: The file has multiple translation links

Example Output

```
================================================================================
TRANSLATION LINK CHECK RESULTS
================================================================================

[EN] Files missing link to Chinese translation (2):
  ✗ en/getting-started/index.rst: Missing :link_to_translation: directive
  ✗ en/api-reference/uart.rst: Incorrect :link_to_translation: directive (found 'en
↪', expected 'zh_CN')

[ZH_CN] Files missing link to English translation (1):
  ✗ zh_CN/getting-started/index.rst: Missing :link_to_translation: directive

[PASSED] Files with correct translation links (15):
  ✓ en/index.rst
  ✓ en/api-reference/i2c.rst
  ...

================================================================================
ERROR: Found 3 file(s) missing translation links.
Please add the appropriate :link_to_translation: directive to these files.

Example:
  For English files: :link_to_translation:`zh_CN:[中文]`
  For Chinese files: :link_to_translation:`en:[English]`
================================================================================
```

4.4.4 Excluding Files from Check

To suppress warnings for specific files, create a `lang-linkcheck-warnings.txt` file in your docs directory. This file should contain one file path per line (relative to the docs directory, e.g., `en/page.rst` or `zh_CN/folder/page.rst`). Lines starting with `#` are treated as comments and ignored.

You can use **wildcards**: `*` matches any string and `?` matches any single character. For example, `en/getting-started/*` ignores all RST files under `en/getting-started/`, and `zh_CN/api-reference/*.rst` ignores all `.rst` files in that folder.

```
# Files to ignore in lang-linkcheck
# One file path per line (relative to docs directory); * and ? are allowed
en/writing-documentation/links.rst
en/getting-started/*.rst
```

4.5 Collecting User Analytics

Google Analytics is a free web analytics service offered by Google that provides data for your website, such as the number of visitors to your site, where they came from, and what pages they viewed. Additionally, you can track the active user and conversion rate. The service is widely used by website owners and marketers to track the performance of their website and improve its performance.

4.5.1 Enabling Google Analytics for Your Project

To enable Google Analytics for your project:

1. Obtain **Tracking ID** by sending your request to Documentation Team Manager.
2. Go to `docs/conf_common.py`, and add the following code to it. The `YOUR_TRACKING_ID` should be changed to the obtained **Tracking ID** from the previous step.

   ```
   # add Tracking ID for Google Analytics

   google_analytics_id = 'YOUR_TRACKING_ID'
   ```

3. Once you' ve added the **Tracking ID** to your project, you will need to wait for some time before data will appear on the Google Analytics platform.

4.5.2 Viewing Google Analytics Data or Reports

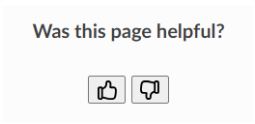
To view Google Analytics data or reports:

1. Create a Google account if you don' t already have one, then sign up for Google Analytics.
2. Gain access to data or reports by sending your Google account to Documentation Team Manager.
3. Log in to Google Analytics and view reports.

See more descriptions in Google Analytics for Beginners.

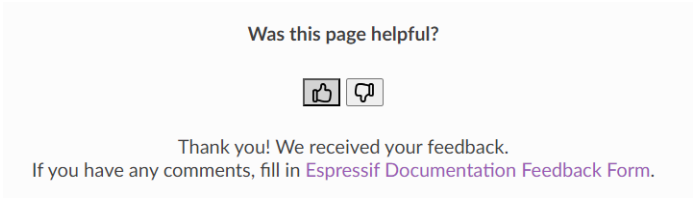
4.6 Collecting Document Feedback

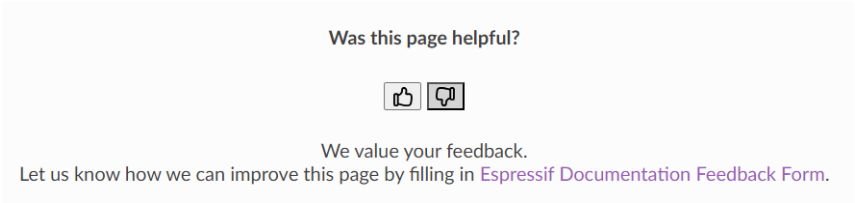
To better gather user feedback, we can use a quicker and easier feedback channel by clicking thumbs up or down, and the number of likes and dislikes will be collected using Google Analytics.



After clicking the thumbs up button:

After clicking the thumbs down button:





Note: *Espressif Documentation Feedback Form* is the link to the classic feedback form.

4.6.1 How to Add the Feedback Section to Your Repo

Currently, there is a *_templates* folder under both the *en* and *zh_CN* directories in the documentation repository. Each folder contains a *layout.html* file that stores the feedback section for this documentation repository. Replace the content of this *layout.html* with the corresponding language' s *layout.html* from the *../../feedback_templates* directory.

You need to replace *YOUR_DOCID* in the official website links (lines 53 and 60) with the corresponding docId for the document. Please note that the docIds for English and Chinese documents are different.

After that, you can compile the documentation, then the feedback section with thumbs up and down will appear.

4.6.2 Configuration of Collection of User Reactions in Google Analytics

We need to add custom events to Google Analytics (GA4) to enable tracking of users feedback on documentation in the form of thumbs up/down.

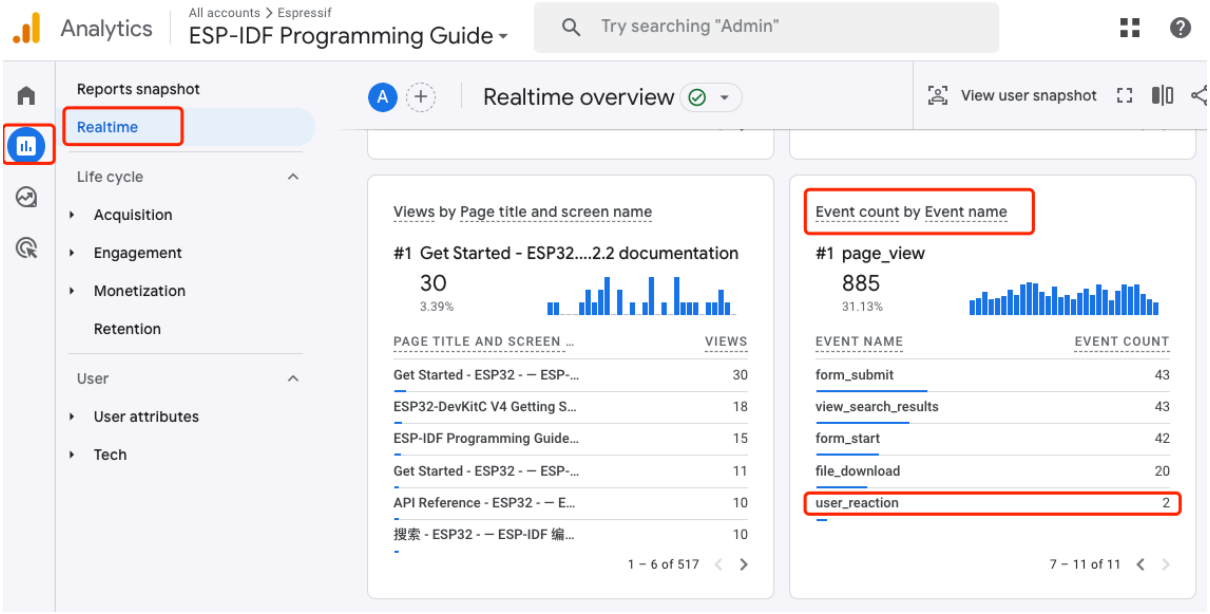
Prerequisites

1. The site with documentation to track reactions should be added to Google Analytics, see ESP-Docs > Configuring ESP-Docs Projects » Collecting User Analytics.
2. The Google Analytics should collect data from the site.
3. The site should include the feedback section that shows thumbs up/down buttons (previous step).
4. To configure the collection of user reactions in Google Analytics, you need to have access to the Google Analytics account that is used for the site with documentation.

Configuration

The configuration of the collection of user reactions in Google Analytics consists of the following steps:

- **Check if Custom Events are Sent to Google Analytics**
    1. Open the site with documentation in a browser and click the thumbs up/down buttons on a couple of pages.
    2. Open the Google Analytics account that is used for the site with documentation. Go to Reports > Realtime.
    3. On the dashboard find the *Event count by Event name* section and check if the events *user_reaction* are sent to Google Analytics. You may need to wait a couple of seconds for the events to appear. See the picture below.



4. If the events are not sent to Google Analytics, check the code that sends the events to Google Analytics (previous step).

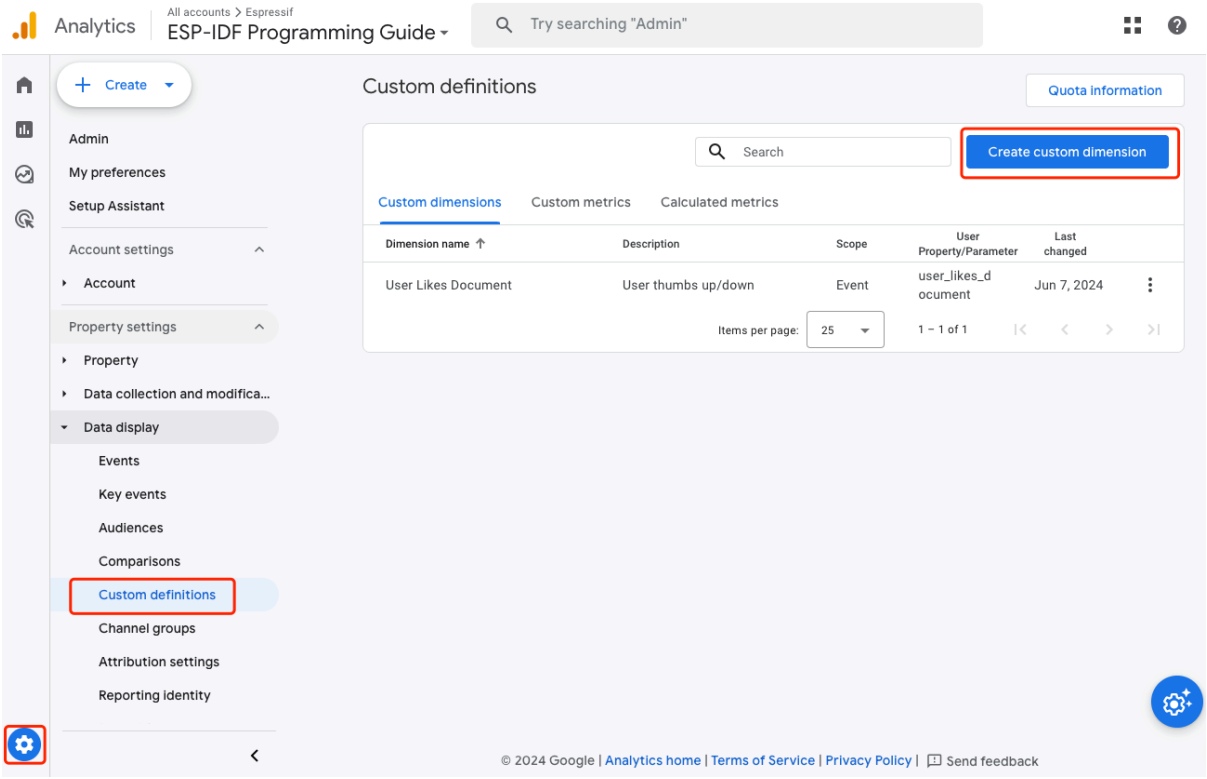
- **Add Custom Definition to Google Analytics**
  1. Open the Google Analytics account that is used for the site with documentation. Go to Admin > Property settings > Data display > Custom definitions.
  2. Click *Create custom dimension* and configure the custom dimension as follows:
     - Dimension name: *User Like Document*
     - Description: *User thumbs up/down reaction on documentation*
     - Scope: *Event*
     - User Property/Parameter: *user_likes_document*
  3. Click *Save* to save the custom dimension. See the picture below how the custom dimension should look like.

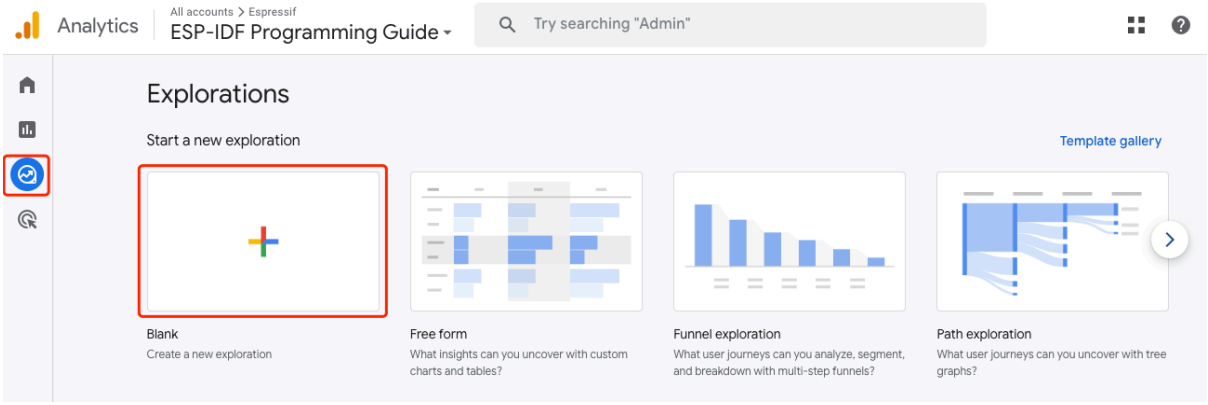
- **Configure Exploration Report**
  1. Go to Explore and click *Create a new exploration*. See the picture below.
  2. Configure the report' s input *Variables*.
     Under *DIMENSIONS*, click +. On the next page that will show up search and then select the following dimensions (see picture below). Click *Import* once both dimensions are selected.
     - Page Title
     - User Likes Document
     - Event Name

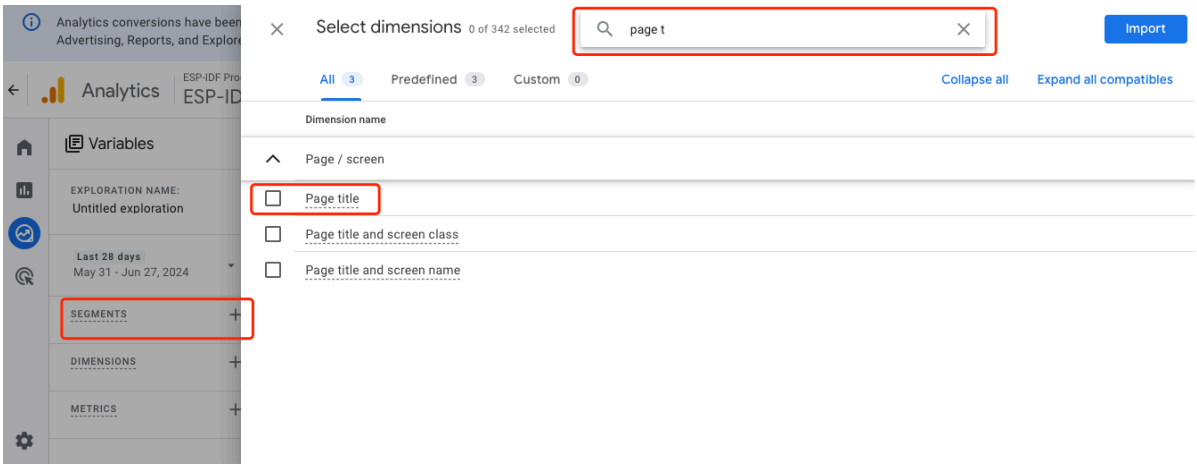
     Under *METRICS*, click +. On the next page that will show up search and then select the following metric. Click Import once the metric is selected.
     - Event count
  3. Configure report by dragging and dropping previously added dimensions and metrics to the *Settings* panel.
     - Drop *Page Title* dimension to the *ROWS* section.
     - Drop *User Likes Document* dimension to the *COLUMNS* section.
     - Drop *Event count* metric to the *VALUES* section.
     - Drop *Event name* dimension to the *FILTERS* section.
     - Set the *Event name* filter to *exactly matches* the *user_reaction*.

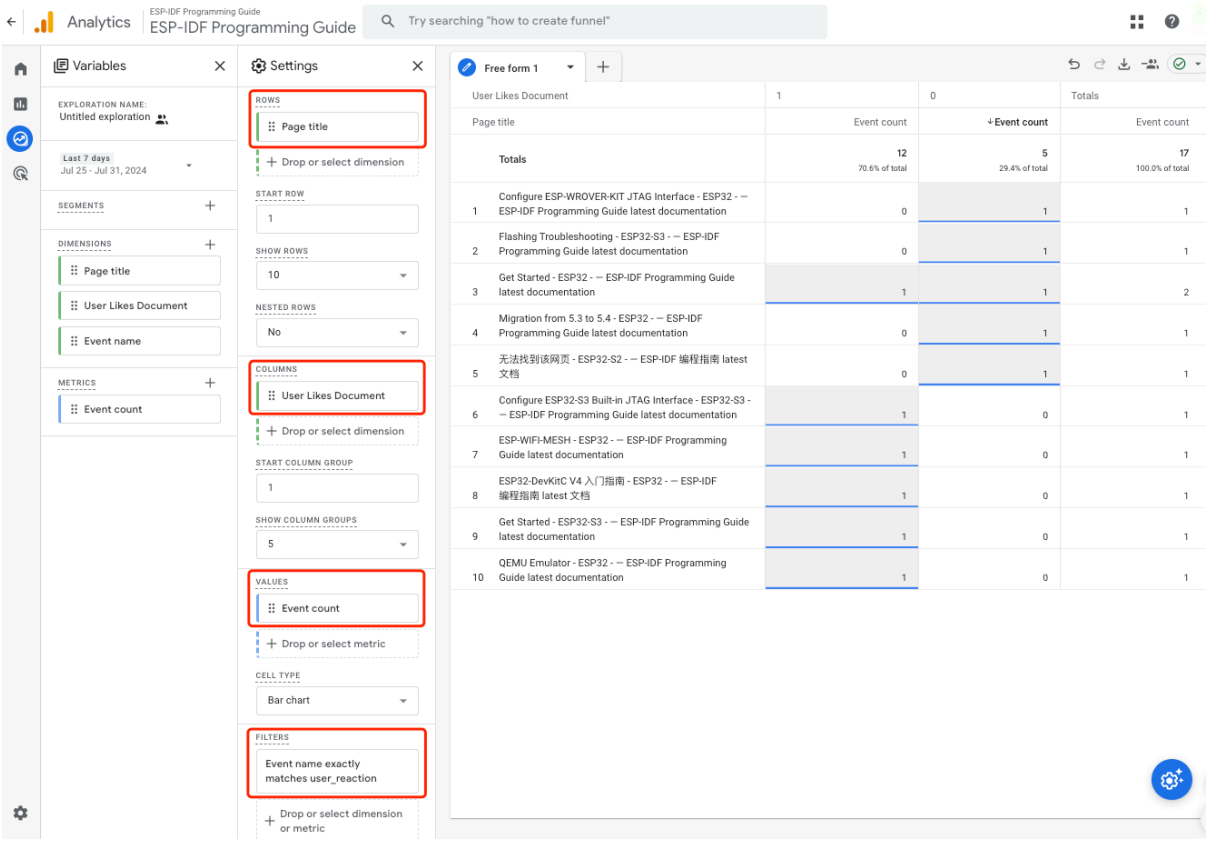
     See the picture below. You may need to scroll down the *Settings* panel to see the *FILTERS* section.
  4. Fine tune the configuration:
     - Change the *EXPLORATION NAME* to *Show User Reactions*.
     - Select desired period to show the results.
     - Change *SHOW ROWS* to e.g. *100* to display more rows.

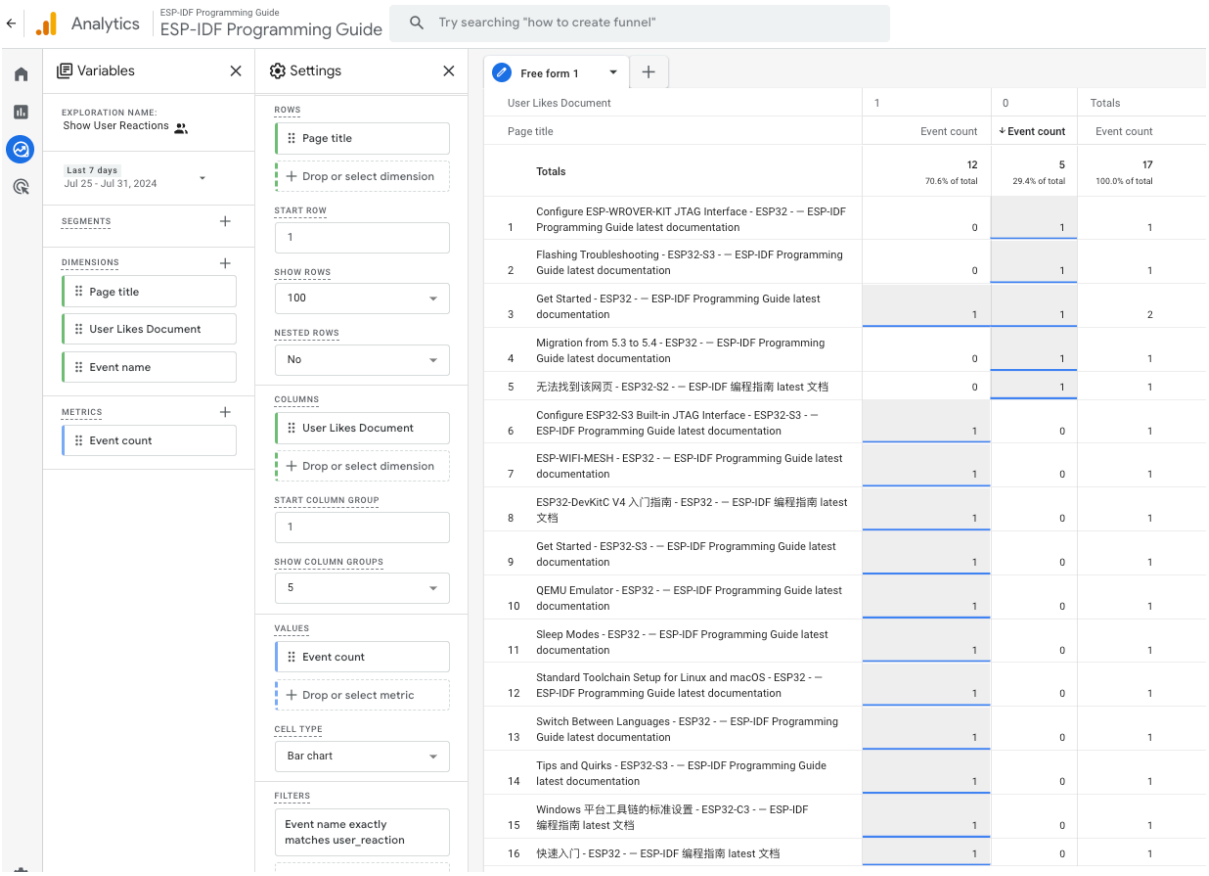
     The report would look like the picture below.


Analytics
All accounts > Espressif
ESP-IDF Programming Guide
Try searching "Admin"
Create
Admin
My preferences
Setup Assistant
Account settings
Account
Property settings
Property
Data collection and modifica...
Data display
Events
Key events
Audiences
Comparisons
Custom definitions
Channel groups
Attribution settings
Reporting identity
Custom definitions
Quota information
Search
Create custom dimension
Custom dimensions
Custom metrics
Calculated metrics
Dimension name
Description
Scope
User Property/Parameter
Last changed
User Likes Document
User thumbs up/down
Event
user_likes_document
Jun 7, 2024
Items per page:
25
1 – 1 of 1
© 2024 Google | Analytics home | Terms of Service | Privacy Policy | Send feedback


Analytics
All accounts > Espressif
ESP-IDF Programming Guide
Try searching "Admin"
Explorations
Start a new exploration
Template gallery
Blank
Create a new exploration
Free form
What insights can you uncover with custom charts and tables?
Funnel exploration
What user journeys can you analyze, segment, and breakdown with multi-step funnels?
Path exploration
What user journeys can you uncover with tree graphs?


Select dimensions
0 of 342 selected
page t
Import
All
Predefined
Custom
Collapse all
Expand all compatibles
Dimension name
Page / screen
Page title
Page title and screen class
Page title and screen name
Variables
EXPLORATION NAME:
Untitled exploration
Last 28 days
May 31 - Jun 27, 2024
SEGMENTS
DIMENSIONS
METRICS


Analytics
ESP-IDF Programming Guide
ESP-IDF Programming Guide
Try searching "how to create funnel"
Variables
Settings
EXPLORATION NAME:
Untitled exploration
Last 7 days
Jul 25 - Jul 31, 2024
SEGMENTS
DIMENSIONS
Page title
User Likes Document
Event name
METRICS
Event count
ROWS
Page title
Drop or select dimension
START ROW
1
SHOW ROWS
10
NESTED ROWS
No
COLUMNS
User Likes Document
Drop or select dimension
START COLUMN GROUP
1
SHOW COLUMN GROUPS
5
VALUES
Event count
Drop or select metric
CELL TYPE
Bar chart
FILTERS
Event name exactly matches user_reaction
Drop or select dimension or metric
Free form 1
User Likes Document
1
0
Totals
Page title
Event count
↓ Event count
Event count
Totals
12
70.6% of total
5
29.4% of total
17
100.0% of total
1 Configure ESP-WROVER-KIT JTAG Interface - ESP32 - — ESP-IDF Programming Guide latest documentation 0 1 1
2 Flashing Troubleshooting - ESP32-S3 - — ESP-IDF Programming Guide latest documentation 0 1 1
3 Get Started - ESP32 - — ESP-IDF Programming Guide latest documentation 1 1 2
4 Migration from 5.3 to 5.4 - ESP32 - — ESP-IDF Programming Guide latest documentation 0 1 1
5 无法找到该网页 - ESP32-S2 - — ESP-IDF 编程指南 latest 文档 0 1 1
6 Configure ESP32-S3 Built-in JTAG Interface - ESP32-S3 - — ESP-IDF Programming Guide latest documentation 1 0 1
7 ESP-WIFI-MESH - ESP32 - — ESP-IDF Programming Guide latest documentation 1 0 1
8 ESP32-DevKitC V4 入门指南 - ESP32 - — ESP-IDF 编程指南 latest 文档 1 0 1
9 Get Started - ESP32-S3 - — ESP-IDF Programming Guide latest documentation 1 0 1
10 QEMU Emulator - ESP32 - — ESP-IDF Programming Guide latest documentation 1 0 1


Analytics
ESP-IDF Programming Guide
ESP-IDF Programming Guide
Try searching "how to create funnel"
Variables
Settings
EXPLORATION NAME:
Show User Reactions
Last 7 days
Jul 25 - Jul 31, 2024
SEGMENTS
DIMENSIONS
Page title
User Likes Document
Event name
METRICS
Event count
ROWS
Page title
Drop or select dimension
START ROW
1
SHOW ROWS
100
NESTED ROWS
No
COLUMNS
User Likes Document
Drop or select dimension
START COLUMN GROUP
1
SHOW COLUMN GROUPS
5
VALUES
Event count
Drop or select metric
CELL TYPE
Bar chart
FILTERS
Event name exactly matches user_reaction
Free form 1
User Likes Document
1
0
Totals
Page title
Event count
↓ Event count
Event count
Totals
12
70.6% of total
5
29.4% of total
17
100.0% of total
1 Configure ESP-WROVER-KIT JTAG Interface - ESP32 - — ESP-IDF Programming Guide latest documentation 0 1 1
2 Flashing Troubleshooting - ESP32-S3 - — ESP-IDF Programming Guide latest documentation 0 1 1
3 Get Started - ESP32 - — ESP-IDF Programming Guide latest documentation 1 1 2
4 Migration from 5.3 to 5.4 - ESP32 - — ESP-IDF Programming Guide latest documentation 0 1 1
5 无法找到该网页 - ESP32-S2 - — ESP-IDF 编程指南 latest 文档 0 1 1
6 Configure ESP32-S3 Built-in JTAG Interface - ESP32-S3 - — ESP-IDF Programming Guide latest documentation 1 0 1
7 ESP-WIFI-MESH - ESP32 - — ESP-IDF Programming Guide latest documentation 1 0 1
8 ESP32-DevKitC V4 入门指南 - ESP32 - — ESP-IDF 编程指南 latest 文档 1 0 1
9 Get Started - ESP32-S3 - — ESP-IDF Programming Guide latest documentation 1 0 1
10 QEMU Emulator - ESP32 - — ESP-IDF Programming Guide latest documentation 1 0 1
11 Sleep Modes - ESP32 - — ESP-IDF Programming Guide latest documentation 1 0 1
12 Standard Toolchain Setup for Linux and macOS - ESP32 - — ESP-IDF Programming Guide latest documentation 1 0 1
13 Switch Between Languages - ESP32 - — ESP-IDF Programming Guide latest documentation 1 0 1
14 Tips and Quirks - ESP32-S3 - — ESP-IDF Programming Guide latest documentation 1 0 1
15 Windows 平台工具链的标准设置 - ESP32-C3 - — ESP-IDF 编程指南 latest 文档 1 0 1
16 快速入门 - ESP32 - — ESP-IDF 编程指南 latest 文档 1 0 1

Note: The exploration reports do not show values for the current day. You need to wait at least 24 h from the first thumbs up/down clicked to see some values.

4.6.3 References

- Collect additional information that Google Analytics does not collect automatically

4.7 Configuring `codespell` in GitLab Repositories

This guide explains how to integrate codespell, a tool that detects common misspellings in source code, into a GitLab repository.

Two main usage scenarios are covered:

1. **Local pre-commit hooks**
   `codespell` runs automatically when you attempt a Git commit, scanning staged changes for spelling errors.
2. **CI/CD pipeline enforcement**
   `codespell` runs in GitLab CI as a job. If spelling errors are found, the job fails and contributors must fix issues locally before resubmitting.

In both cases, a shared configuration file `.codespellrc` ensures consistent behavior.

Example:

```
[codespell]
skip = build,*.drawio,*.svg,*.pdf
ignore-words-list = laf,OT
write-changes = true
```

- `skip`: Specifies directories or file extensions to ignore (e.g., build outputs, diagrams).
- `ignore-words-list`: Defines project-specific terms that should not be flagged (e.g., `laf`, `OT`).
- `write-changes`: Lets `codespell` automatically apply suggested fixes. If omitted, `codespell` runs in dry-run mode and only reports issues.

4.7.1 Using `codespell` as a Git Pre-Commit Hook (Local)

This method prevents spelling errors from being committed.

Steps

1. Install `pre-commit`.
   `pre-commit` is a framework that manages Git hooks. Install it with:

   ```
   pip install pre-commit
   ```

2. Activate the pre-commit hook in your repository.
   Go to your local repository and enable hooks with:

   ```
   pre-commit install
   ```

3. Modify `.pre-commit-config.yaml`.
   Add `codespell` configuration:

```yaml
repos:
  - repo: https://github.com/codespell-project/codespell
    rev: v2.4.1 #latest release version
    hooks:
      - id: codespell
        args: [--config=.codespellrc]
```

- `repo`: Location of the codespell hook repository.
- `rev`: The version of codespell to use. It is recommended to use **v2.4.1** to remain consistent with other repositories.
- `id`: Selects the codespell hook.
- `args`: Passes additional arguments, here explicitly pointing to `.codespellrc`.

4.7.2 Expected Behavior of the Local `pre-commit` Hook for `codespell`

When the local `pre-commit` hook for `codespell` runs, it scans only the files that have been modified. During the check, `codespell` automatically fixes typos when it is confident about the correct replacement.

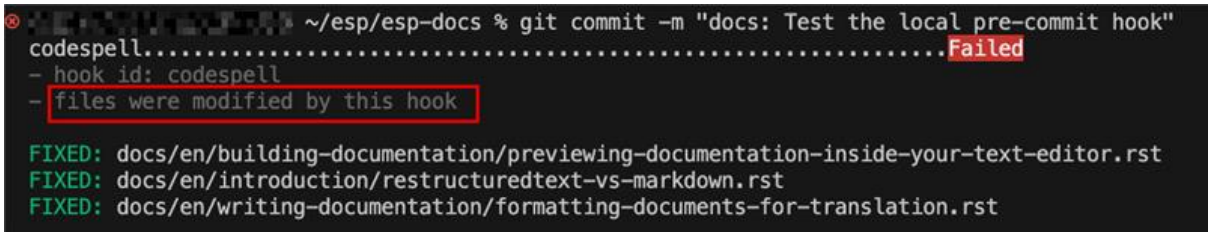


Fig. 2: Automatic Correction by `codespell`

If there are multiple possible corrections, `codespell` does not modify the files directly. Instead, it provides suggestions in the terminal output for you to review. For example, messages in the following screenshot show that `codespell` has found two potential typos, "editers" and "repoted" , in lines 4 and 12 respectively. However, it is uncertain which correction to apply, so it displays the possible alternatives in green highlights for you to review and adopt. # codespell:ignore (intentional typos for demonstration)



Fig. 3: Suggested Corrections from `codespell`

4.7.3 Implement `codespell` in GitLab CI/CD

To ensure spelling checks are enforced for all contributions, you can integrate `codespell` into the GitLab CI/CD pipeline via `pre-commit`. When configured, each Merge Request pipeline triggers a job that scans **only the modified files**. If spelling errors are found, the job fails, and the contributor must fix the typos locally before pushing the corrected commit.

You can add `codespell` to your CI/CD process by either modifying `.gitlab-ci.yml` or, if `pre-commit` is already integrated (usually in `.gitlab/ci/pre_commit.yml`), simply updating `.pre-commit-config.yaml` without changing any CI files.

1. If you use `.gitlab-ci.yml` to manage all your jobs. Take the following code as an example:

```yaml
codespell_check:
  stage: check
```

(continues on next page)

(continued from previous page)

```
  tags: [test]
  image: $ESP_DOCS_ENV_IMAGE
  rules:
    - if: $CI_PIPELINE_SOURCE == "merge_request_event"
  before_script:
    - pip install pre-commit
    - git fetch origin $CI_MERGE_REQUEST_TARGET_BRANCH_NAME --depth=100
    - git fetch origin $CI_COMMIT_REF_NAME --depth=100
    - export MODIFIED_FILES="$(git diff --name-only origin/$CI_MERGE_REQUEST_
↪TARGET_BRANCH_NAME...origin/$CI_COMMIT_REF_NAME)"
  script:
    - |
      if [ -n "$MODIFIED_FILES" ]; then
        echo "Running codespell on changed files:"
        echo "$MODIFIED_FILES"
        CI=true pre-commit run --files $MODIFIED_FILES
      else
        echo "No modified files to check with codespell."
      fi
```

- `stage`: Specifies the stage of the pipeline where this job belongs. The stage can be **check** or **pre-check** as defined earlier in the `yml` file.
- `tags`: (Optional) Selects specific runners for this job. Most repositories require this field, unless the `Run untagged jobs` option is enabled. Use the tag name that matches your repository's runner configuration.
- `image`: Specifies the Docker image used for running jobs. This should match the image already configured in your repository.
- `rules`: Defines conditions for when the job should run. For example, using `if: $CI_PIPELINE_SOURCE == "merge_request_event"` ensures the job only runs in pipelines triggered by a Merge Request, and not for regular branch pushes or scheduled pipelines.
- `before_script`:
  - Installs `pre-commit` into the job environment.
  - Fetches the diff range between target and source branches.
  - Collects the list of modified files for spell checking.
- `script`: Runs `codespell` via `pre-commit` on modified files only. If spelling errors are detected, the job fails.

Note: You can also inherit configuration from a predefined template, e.g., `.before_script_minimal`. This reduces repetition and ensures consistent initialization across jobs.

```
extends:
  - .before_script_minimal
```

---

2. If your project runs unified pre-commit jobs in the CI pipeline, adding a separate `codespell_check` job in `.gitlab-ci.yml` is unnecessary. The CI pipeline simply uses the same `.pre-commit-config.yaml` file, ensuring that the checks performed in CI are identical to those on local machines.

How to Run Unified Pre-Commit Jobs in GitLab CI/CD

Running unified pre-commit jobs in GitLab CI/CD is recommended because it allows you to easily add or update hooks in the future with minimal changes. Follow these steps to set it up:

Steps

1. Include the `.gitlab/ci/pre_commit.yml` file in your root `.gitlab-ci.yml` configuration.
   Example:

```
include:
  - '.gitlab/ci/pre_commit.yml'
```

Note: If your `.gitlab-ci.yml` file does not yet define any global workflow rules, please add the following configuration:

```
workflow:
  rules:
    - if: $CI_PIPELINE_SOURCE == "merge_request_event" # Run pipelines only␣
↪for merge requests
    - when: always # Fallback rule to always run if no other rules match
```

2. Create the `.gitlab/ci/pre_commit.yml` file to include the following job definition:

```
.check_pre_commit_template:
  stage: check  # or "pre-check", depending on your pipeline structure
  tags: [test]  # Optional: specify tags to select appropriate runners
  image: $ESP_DOCS_ENV_IMAGE  # Use the same image as other jobs in your␣
↪repository
  rules:
    - if: $CI_PIPELINE_SOURCE == "merge_request_event"  # Run this job only␣
↪for merge request pipelines.
  script:
    - pip install pre-commit
    - git fetch origin $CI_MERGE_REQUEST_TARGET_BRANCH_NAME --depth=100  #␣
↪Fetch target branch with sufficient history to find merge base
    - git fetch origin $CI_COMMIT_REF_NAME --depth=100  # Fetch source branch␣
↪with sufficient history to find merge base
    - |
      echo "Target branch: $CI_MERGE_REQUEST_TARGET_BRANCH_NAME"
      echo "Source branch: $CI_COMMIT_REF_NAME"

      MODIFIED_FILES=$(git diff --name-only origin/$CI_MERGE_REQUEST_TARGET_
↪BRANCH_NAME...origin/$CI_COMMIT_REF_NAME)
      echo "Modified files to check:"
      echo "$MODIFIED_FILES"

      if [ -n "$MODIFIED_FILES" ]; then
          CI=true pre-commit run --files $MODIFIED_FILES
      else
          echo "No modified files to check."
      fi

check_pre_commit:
  extends:
    - .check_pre_commit_template  # All configured pre-commit hooks will run␣
↪under this job
```

Note: If your repository includes **submodules**, make sure to fetch them in the pre-commit job to ensure all files are available for scanning.
In this case, add the following configuration under `.check_pre_commit_template`:

```
variables:
  GIT_STRATEGY: fetch
  SUBMODULES_TO_FETCH: "all"
script:
  - fetch_submodules  # Usually defined in ``utils.sh`` to fetch submodules
  # If 'fetch_submodules' is not available, you can alternatively use:
  # 'git submodule update --init --recursive'
```

3. Modify the `.pre-commit-config.yaml` file to add the expected hook.
   Example:

```
repos:
  - repo: https://github.com/codespell-project/codespell
    rev: v2.4.1
    hooks:
      - id: codespell
        args: [--config=.codespellrc]
```

4. (Optional) Move any other custom jobs under the "check" or "pre_check" stages from `gitlab-ci.yml` to `.gitlab/ci/pre_commit.yml` if they cannot be defined in the "repos" format in `.pre-commit-config.yaml`.
   Example:

```
check_setup:
  stage: check
  image: $ESP_DOCS_ENV_IMAGE
  extends:
    - .before_script_minimal
  script:
    - pip install .
```

After completing these steps, your CI pipeline will automatically run all pre-commit hooks on the files modified in each Merge Request.

For a more detailed example, refer to .gitlab-ci.yml in the ESP-IDF repository.

4.7.4 Summary

- **Local pre-commit hook**: Prevents contributors from committing misspelled files.
- **CI/CD pipeline check**: Enforces centralized spelling checks in CI pipelines.
- **Unified config**: `.codespellrc` guarantees consistent behavior across environments.

Combining both approaches ensures consistent spelling and improves documentation quality.

Potential Risks of `codespell`

The way `codespell` works is by using a dictionary of common misspellings and their correct forms. During checks, it scans files for words in this dictionary and suggests corrections. This means that `codespell` is **not a comprehensive spell checker** –it only detects typos that exist in its dictionary, and may miss other spelling mistakes.

To catch more typos in a smarter way, it is strongly recommended to enable a spell checker in your IDE or editor. For instance, you can *Use Spellcheckers in VS Code*, which flags spelling mistakes instantly as you type.

Chapter 5

Troubleshooting

5.1 Troubleshooting Build Errors and Warnings

When build fails, a message would pop out to alert you. Such a message has two levels of severity:

- Error, which indicates that the build cannot be completed and no HTML files will be generated.
- Warning, which indicates that the HTML files are generated with errors.

This document provides guidelines on addressing build errors and warnings with the help of messages. Errors and warnings in this document are related to either **the esp-docs package** or the **reStructuredText syntax**.

5.1.1 Message Format

Messages can help you locate errors and warnings and get a hint of why they occur.

For projects using the esp-docs package, a message usually includes the following parameters in sequence:

- Language
- Target
- [Optional] File path
- [Optional] Line number
- Error or warning type

Example of a package-related error:

```
en/esp32s3: Extension error:
en/esp32s3: Could not import extension linuxdoc.rstFlatTable (exception: No module␣
↪named 'linuxdoc')
```

Example of a syntax-related warning:

```
en/esp32s3: Users/johnlee/esp/esp-idf/docs/en/api-reference/peripherals/ledc.
↪rst:318: WARNING: undefined label: pwm-sheet
```

Among these parameters, **file path** and **line number** are optional. They will not be provided if an error or warning is general and does not apply to a specific file or line.

5.1.2 Package-Related Errors and Warnings

Command not found: build-docs

This error occurs when you have not:

- installed the esp-docs package properly;

- or correctly set the environment variable PATH.

To address this error, please go to Section *Building Documentation locally on Your OS*, and make sure you have completed all the steps required.

Application error: Cannot find source directory

This error occurs when you are not in the right directory.

To address this error, navigate to `docs` directory:

```
cd docs
```

Extension error: Could not import extension

This error occurs when you add a new extension to `conf_common.py` (or in some projects `conf.py`), but forget to install this extension.

To address this error, there are two options:

- Option 1: for an extension specific to your project, add it and its version to `docs/requirements.txt` of your project. For example, if the extension is sphinx-design and version is 0.2.0, then add:

  ```
  sphinx-design==0.2.0
  ```

  And run the following command in `docs` directory:

  ```
  pip install -r requirements.txt
  ```

- Option 2: for an extension that might be reused in other projects using the esp-docs package, add it and its version to setup.cfg of the ESP-Docs project, for example:

  ```
  install_requires =
      sphinx-design==0.2.0
  ```

  And run the following command:

  ```
  pip install esp-docs
  ```

SyntaxError: future feature annotations is not defined

Future feature annotations is available from Python 3.7. This error might occur when Python version is too low.

To address this error, try to upgrade your Python to the required version. The required Python version can be found in setup.cfg.

exception: No documents to build

This error occurs when you build a single document, but this document cannot be found at the specified path. For example:

```
build-docs -t esp32 -l en -i api-reference/peripherals/can.rst
```

To address this error, correct the document path:

```
build-docs -t esp32 -l en -i api-reference/peripherals/twai.rst
```

5.1.3 Syntax-Related Errors and Warnings

ERROR: Unknown interpreted text role

This error occurs when you use an incorrect role, for example `docs` instead of `doc`.

To address this error, correct the name of the *role*.

ERROR: Unknown target name

This error occurs when the reference to a `` `target`_ `` cannot be found by Sphinx.

For example, the section is named as `Syntax-Related Errors and Warnings`, but referred to as `Syntax-Related Errors and Warning` without **s** at the end:

```
Related resources:

- `Package-Related Errors and Warnings`_
- `Syntax-Related Errors and Warning`_

Package-Related Errors and Warnings
-----------------------------------

Syntax-Related Errors and Warnings
-----------------------------------
```

To address this error, correct the target name.

ERROR: Unknown directive type

This error occurs when you use directives of an extension not covered by your project or by the esp-docs package.

To address this error, add the extension following *Adding Extensions*.

WARNING: the underline is too short

This warning occurs when the section title underline is too short, for example:

```
Getting Started
===========
```

To fix this warning, make the title underline the same length as or longer than the title:

```
Getting Started
===============
```

Note: For Chinese titles, each Chinese character requires two underline markers (e.g. =).

WARNING: image file not readable

This warning occurs when Sphinx cannot find the image at the specified path.

To fix this warning, check if the image path is correct.

WARNING: unknown document

This warning occurs when Sphinx cannot find the document at specified path.

To fix this warning:

1. Check if the document path is correct.
2. Check if you have used correct syntax for *role*. For instance, `.rst` in the following example should be removed (see *Links*):

```
:doc:`reStructuredText Syntax <../writing-documentation/basic-syntax.rst>`
```

WARNING: document isn't included in any toctree

`toctree` directive glues all `.rst` files together into a table of contents (TOC). Therefore, by default every `.rst` file is required to be placed under a toctree, otherwise this warning will occur.

To fix this warning, there are two options:

- Option 1: add the `.rst` file to its corresponding `toctree`, for example:

  ```
  .. toctree::
      :maxdepth: 2

      user_guide
  ```

  Usually the corresponding `toctree` is in the `index.rst` file of the parent folder, and adding file name without `.rst` extension would be sufficient.
  If you have already included the `.rst` file in a `toctree` and this warning still occur, check whether you have used the `.. only:: TAG` directive or the `:TAG:` role provided by the *multiple target* feature of esp-docs. For example:

  ```
  .. only:: esp32

      .. toctree::
      :maxdepth: 2

      user_guide
  ```

  ```
  .. toctree::
  :maxdepth: 2

  :SOC_BT_SUPPORTED: bluetooth
  ```

  If yes, suppress this warning by adding the `.rst` file to the list of documents it belongs to in `docs/conf_common.py` or. For example:

  ```
  BT_DOCS = ['api-guides/bluetooth.rst]
  ```

- Option 2: add `:orphan:` at the beginning of the `.rst` file. Note that in this way, this file will not be reachable from any table of contents, but will have a matchable HTML file.

WARNING: undefined label

This warning occurs when reference `:ref:` points to a non-existing label, for example:

```
The pin header names are shown in Figure :ref:`user-guide-c6-devkitc-1-v1-board-
↪front`.
```

To fix this warning, add the missing label `.. _user-guide-c6-devkitc-1-v1-board-front:` before the place you want to link to:

```
.. _user-guide-c6-devkitc-1-v1-board-front:

.. figure:: ../../../_static/esp32-c6-devkitc-1/esp32-c6-devkitc-1-v1-annotated-
↪photo.png
:align: center
:alt: ESP32-C6-DevKitC-1 - front
:figclass: align-center

ESP32-C6-DevKitC-1 - front
```

WARNING: Duplicate label

This warning occurs when the label is not unique, for example:

```
.. _order:
Retail orders
^^^^^^^^^^^^

.. _order:
Wholesale Orders
^^^^^^^^^^^^^^^^
```

To fix this warning, rename the labels to make them unique.

WARNING: duplicate substitution

This warning occurs when the substitution is defined multiple times, either in the same file, or in different files within the same project. For example, the substitution to `|placeholder|` is defined both in `bluetooth.rst` and `wifi.rst`:

```
.. |placeholder| image:: https://dl.espressif.com/public/table-header-placeholder.
↪png
```

To fix this warning, delete repetitive substitutions.

You might encounter cases that after deleting repetitive substitution in `bluetooth.rst`, the `|placeholder|` in `bluetooth.rst` cannot be substituted by its definition in `wifi.rst` with the following error message popping out:

```
ERROR: undefined substitution referenced: "placeholder"
```

If this is the case, you may add this substitution definition to the end of every `.rst` file by using `rst_epilog` in `docs/conf_common.py` (or `docs/conf.py`):

```
rst_epilog = """
.. |placeholder| image:: https://dl.espressif.com/public/table-header-placeholder.
↪png
"""
```

5.1.4 Still Have Troubles?

This document is far from comprehensive. If you still have no clue why your build fails, here are a few more support options:

- Contact us by submitting documentation feedback.
- For syntax-related errors and warnings, refer to Chapter *Writing Documentation* for the correct format.

Chapter 6

Contributing Guide

ESP-Docs is an open and common project and we welcome contributions.

Please contribute via GitHub Pull Requests or internal GitLab Merge Requests.

6.1 Report a Bug

- Before reporting a bug, check *Troubleshooting*. You may find the cause of and the fix to the problem.
- If you just want to report the bug, contact the Documentation Team directly, or fill in the documentation feedback form.
- If you want to fix the bug, open a pull/merge request with your fix. In the request, describe the problem and solution clearly.

6.2 Add a New Feature

- Check *What Is ESP-Docs?* to ensure the feature to be introduced is not implemented in the esp-docs project.
- Open a pull/merge request with your code. In the request, mention what the feature is about and how it will improve the esp-docs project.
- Self-check if your code conforms to esp-idf coding style to speed up the following review process.

6.3 Make Minor Changes

- If you identify typos, grammar errors, or broken links, or want to make other minor changes, contact the Documentation Team directly, or fill in the documentation feedback form.
- The Documentation Team will make bulk changes periodically based on such requests.

6.4 Use the ESP-IDF Documentation Agent Guide

ESP-Docs includes a versioned guide for AI coding agents that update ESP-IDF documentation for a chip target. After installing ESP-Docs, print the guide with:

```
esp-docs agent-guide update-docs
```

The command is read-only and writes the complete guide to standard output. It does not require an IDE-specific skill.

To make local agents discover the guide, a downstream repository can add the following instruction to its `AGENTS.md`:

```
When updating ESP-IDF documentation for a chip target, run
`esp-docs agent-guide update-docs` and follow the returned instructions
before editing documentation.
```

The guide matches the installed ESP-Docs package version, so projects that pin ESP-Docs also pin the corresponding agent workflow.

6.5 Commit Messages

This project uses Conventional Commits. A pre-commit hook enforces the format on every commit message, and CI validates commit messages for all commits in a merge request. To install the hooks locally, run:

```
pip install pre-commit
pre-commit install --install-hooks
```

Each commit message must follow the pattern `<type>: <description>`, for example:

- `fix: correct version parsing for tags with prefix`
- `feat: add support for ESP32-H4 target`
- `docs: update contributing guide`
- `ci: add commitizen to pre-commit hooks`

Only `fix` and `feat` commits trigger a version bump. Other types (`docs`, `ci`, `refactor`, `test`, etc.) are recorded in the changelog but do not change the version.

A commit with a `BREAKING CHANGE` footer or a `!` after the type (e.g. `feat!: remove legacy API`) triggers a major version bump.

6.6 Ask a Question

- If you have questions regarding the documentation or code here, contact the Documentation Team directly, or fill in the documentation feedback form.

6.7 Release Process

Releases are managed with Commitizen and published to PyPI via a GitHub Actions workflow.

To create a new release:

```
cz bump
```

This command will:

1. Determine the next version based on commits since the last tag.
2. Update the version in `setup.cfg` and `pyproject.toml`.
3. Generate/update `CHANGELOG.md`.
4. Create a commit and a git tag (e.g. `v2.1.5`).

To preview what would happen without making changes:

```
cz bump --dry-run
```

Once the version bump is ready:

1. Merge the version bump to the `master` branch.
2. Wait for the repository to sync from GitLab to GitHub.
3. Create a new GitHub Release with a tag matching the version (e.g., `v2.2.0`).

4. The `release_pypi.yml` workflow will automatically build and publish the package to PyPI.

The workflow uses PyPI trusted publishing (OIDC) so no API tokens need to be managed. If the version already exists on PyPI the upload will be skipped.

Chapter 7

Related Resources

- reStructuredText Directives describes the directives that extend the reStructuredText (reST) syntax.
- Sphinx covers the basics of getting started with Sphinx. On this site, reStructuredText Primer section introduces reST concepts and syntax.
- API Documentation Template provides a template in rst format to document API.
- To see the rendered output of directives and functionality that ESP-Docs supports, refer to the documentation built with ESP-Docs, such as
  - ESP-IDF Programming Guide
  - ESP-AT User Guide
  - esptool.py Documentation
  - mDNS Service
  - ESP WebSocket Client
  - ASIO port
  - ESP MQTT C++ client
- To learn more about how documentation builds, go to
  - Docutils that explains how this text processing system processes plaintext documentation into formats like HTML
  - Doxygen that explains how the standard tool generates documentation from annotated C++ sources
- To try Markdown in your documentation, Recommonmark parser' s documentation page presents an approach to write CommonMark inside of Sphinx projects.
- An overview of ESP-Docs is given in the ESP DevCon22 talk *How to create awesome documentation for your ESP32-X using ESP-Docs*. The video starts at 2:32:21.
- Espressif Manual of Style answers some writing style questions.
- To seek help from the Documentation Team, please check Documentation Team Site. Type the keywords in the search box, and all relevant ready-made documents will show up.

Chapter 8

Glossary

This document lists terms that are used in ESP-Docs documentation. Each term is followed by its definition and some have notes.

add-ons
: Definition: Add-ons are small programs that expand or extend the features of a browser.

blockdiag
: Definition: blockdiag is an application that generates raster images from plaintext .diag source files.
Note: Do not capitalize the first letter `b` when using this term.

dependency
: Definition: When a project consumes executable code generated by another project, the project that generates the code is referred to as a project dependency of the project that consumes the code.

Docutils
: Definition: Docutils is an open-source text processing system for processing plaintext documentation into useful formats, such as HTML, LaTeX, man-pages, OpenDocument, or XML.

Doxygen
: Definition: Doxygen is a documentation generator and static analysis tool for software source trees.

ESP-Docs
: Definition: ESP-Docs is a documentation-building system developed by Espressif based on Sphinx and Read the Docs. It expands Sphinx functionality and extensions with the features needed for Espressif's documentation and bundles this into a single package. See *What Is ESP-Docs?* for more information.
Note: ESP-Docs is used as a proper noun just like other documentation generators including Sphinx, or software products like ESP-IDF and ESP-IoT Solution. Use `esp-docs` only in the repo name, code, command line, folder/file name, etc.

Graphviz
: Definition: Graphviz is a package of open-source tools for drawing graphs specified in DOT language scripts having the file name extension "gv".

interactive shell
: An interactive shell is defined as the shell that simply takes commands as input on tty from the user and acknowledges the output to the user.

LaTeX
: Definition: LaTeX is a software system for document preparation. When writing, the writer uses plain text as opposed to the formatted text found in WYSIWYG word processors like Microsoft Word, LibreOffice Writer and Apple Pages.
Note: `T` and `X` should be capitalized.

Markdown
: Definition: Markdown is a lightweight markup language for creating formatted text using a plaintext editor.
Note: When placed in documentation, the first letter should be capitalized.

Read the Docs
: Definition: Read the Docs is an open-sourced free software documentation hosting platform. It generates documentation written with the Sphinx documentation generator.
Note: Spaces should be added around `the`, and `D` should be capitalized.

reStructuredText

Definition: reStructuredText is a file format for textual data used primarily in the Python programming language community for technical documentation.
Note: Abbreviations of this term include reST, or rst. When using its full name, `S` and `T` should be capitalized, while `r` remains lowercase.

slug

Definition: A unique identifier for a project, version, or target. This value comes from the project, version name, or target name, such as `esp-idf`, `release-v5.0`, or `esp32` in https://docs.espressif.com/projects/esp-idf/en/release-v5.0/esp32/index.html.

Sphinx

Definition: Sphinx is a powerful documentation generator that has many great features for writing technical documentation.
Note: When placed in documentation, `S` should be capitalized.

target

Definition: Represents a series of Espressif products for which you build documentation, e.g., ESP32, ESP32-S2, ESP32-C3.

WaveDrom

Definition: WaveDrom draws your Timing Diagram or Waveform from a simple textual description.

Index

A

A-term, **38**
add-ons, **103**

B

B-term, **38**
blockdiag, **103**

D

dependency, **103**
Docutils, **103**
Doxygen, **103**

E

ESP-Docs, **103**

G

Graphviz, **103**

I

interactive shell, **103**

L

LaTeX, **103**

M

Markdown, **103**

R

Read the Docs, **103**
reStructuredText, **103**

S

slug, **104**
Sphinx, **104**

T

target, **104**
Term A, **38**
Term B, **38**

W

WaveDrom, **104**