A Language-Neutral Intermediate Representation for API Reference and User Documentation

(+ a configuration agnostic API documentation extractor)

Lasse Rosenow, 26.08.2026

PART 1

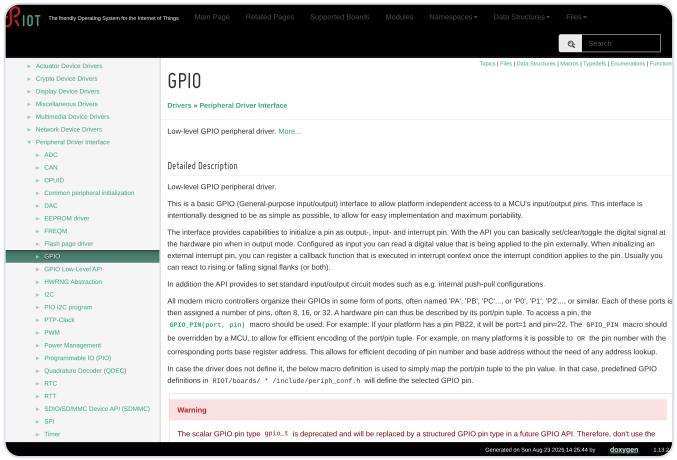
1 Motivation

The problems with existing documentation systems

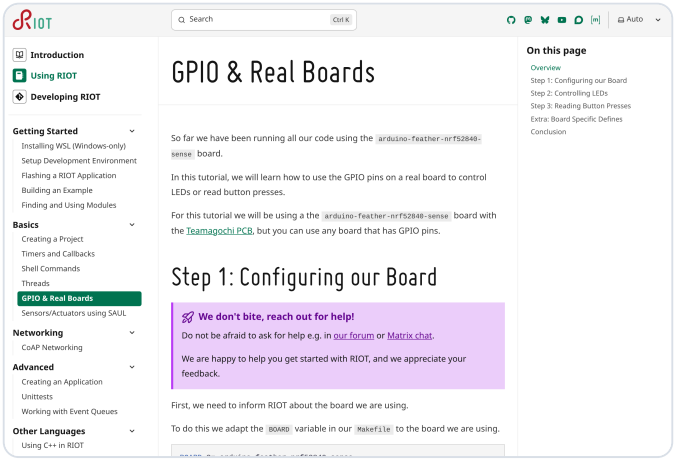
3 Documentation Sites for only 1 Project

RIOT is written in **C**, has **Markdown** user guides and a **Rust** API Wrapper:

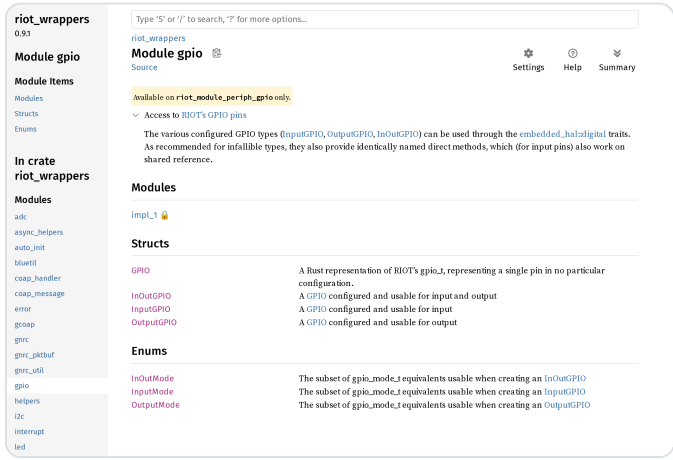
C API



GUIDES



RUST API



Problem: No shared information structure

- No global search across pages
- No reference links across pages

MOTIVATION

Configurations Are Not Exposed

The API surface in Doxygen differs depending on compile time configurations

SIMPLIFIED RIOT C CODE

```c
#if MODULE_ZTIMER_NOW64
  uint64_t ztimer_now(ztimer_clock_t *clock);
#elif MODULE_ZTIMER || DOXYGEN
  uint32_t ztimer_now(ztimer_clock_t *clock);
#else
  /* ztimer not compiled in */
  static inline uint32_t
  ztimer_now(ztimer_clock_t *clock)
  { (void)clock; return 0; }
#endif
```

DOXYGEN OUTPUT

`uint32_t ztimer_now(ztimer_clock_t *clock)`

Get the current time of a clock.

Parameters

`[in]` **clock** ztimer clock to operate on

Returns

Current count on `clock`

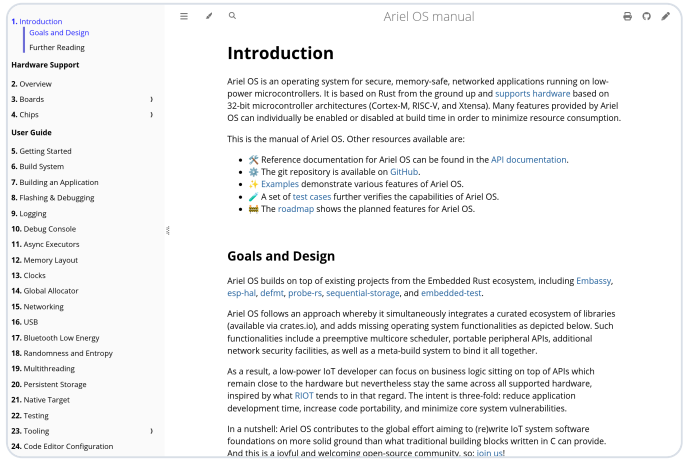
Doxygen only renders one configuration

- On a **64-bit** configuration the documented return type is **wrong**
- And there is also **no information** on how to **enable configurations**

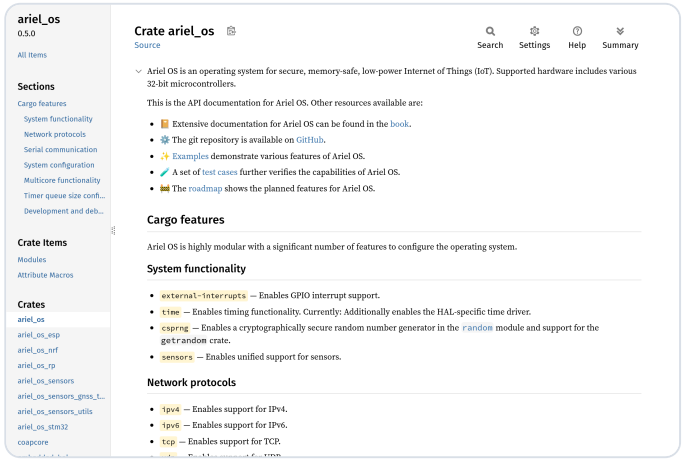
Second Example: Ariel OS: 2 Documentation Sites for only 1 Project

Ariel OS is pure **Rust**, so there is no C API, but the user guides still ship separately

GUIDES



RUST-API

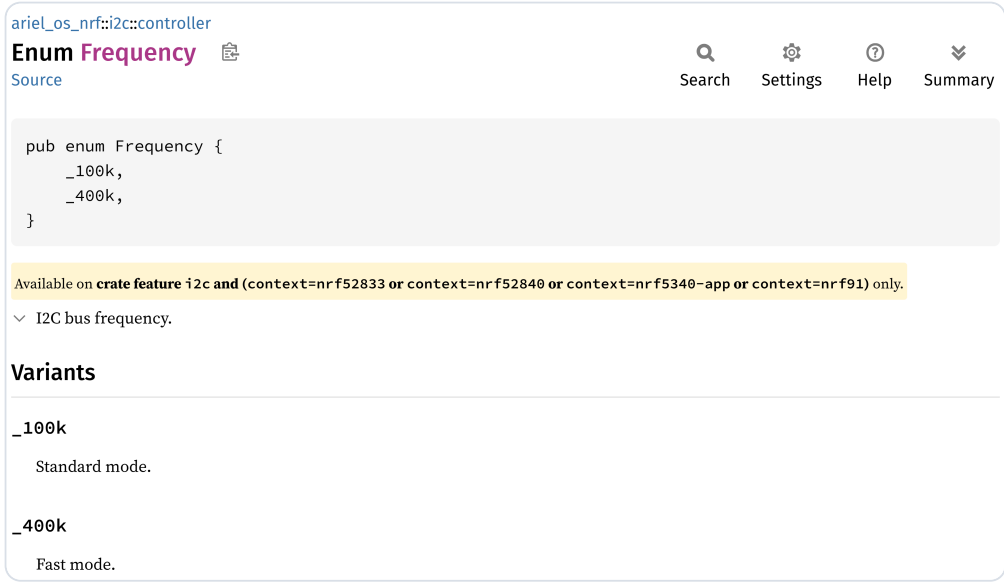


Ariel OS: Configurations Are Only Partially Exposed

ARIEL-OS-NRF/SRC/I2C/CONTROLLER/MOD.RS

```rust
/// I2C bus frequency.
pub enum Frequency {
    _100k,
    #[cfg(any(context = "nrf52833",
              context = "nrf5340-app",
              context = "nrf91"))]
    _250k,
    _400k,
}
```

WHAT ARIEL-OS.GITHUB.IO SHOWS



The published docs are built with `context = "nrf52840"` only

=> **`_250k`** is missing

2

PART 2

Why not Doxygen?

(Doxygen does support multiple input languages after all)

Doxygen vs. "The Problems"

Could the problems be fixed inside Doxygen?

PROBLEM 1: UNIFIED DOCUMENTATION

GUIDES IN DOXYGEN

RIOT OS

- RIOT Documentation
- Creating modules
- Creating an application
- Porting boards
- Writing a Device Driver in RIOT
- Getting started
- Flashing via RIOT's Build System
- Build In Docker
- Running and creating tests
- Build System Basics
- Kconfig in RIOT
- Using C++ in RIOT
- Using Rust in RIOT
- Advanced build system tricks
- Debugging Tools
- Emulators
- Release cycle
- Changelog
- Removed Features and Modules

Deprecated List

Todo List

Supported Boards

- Modules
- Namespaces
- Data Structures
- Files

C OUTPUT LOOKS LIKE C++

kernel_pid_t msg_t::sender_pid

uint16_t msg_t::type

void* msg_t::ptr

No Rust support in Doxygen at all

PROBLEM 2: EXPOSED CONFIGURATIONS

`ENABLE_PREPROCESSING = NO`

```
uint64_t ztimer_now(ztimer_clock_t *clock)
```

Get the current time of a clock.

Parameters

`[in]` **clock** ztimer clock to operate on

Returns

Current count on `clock`

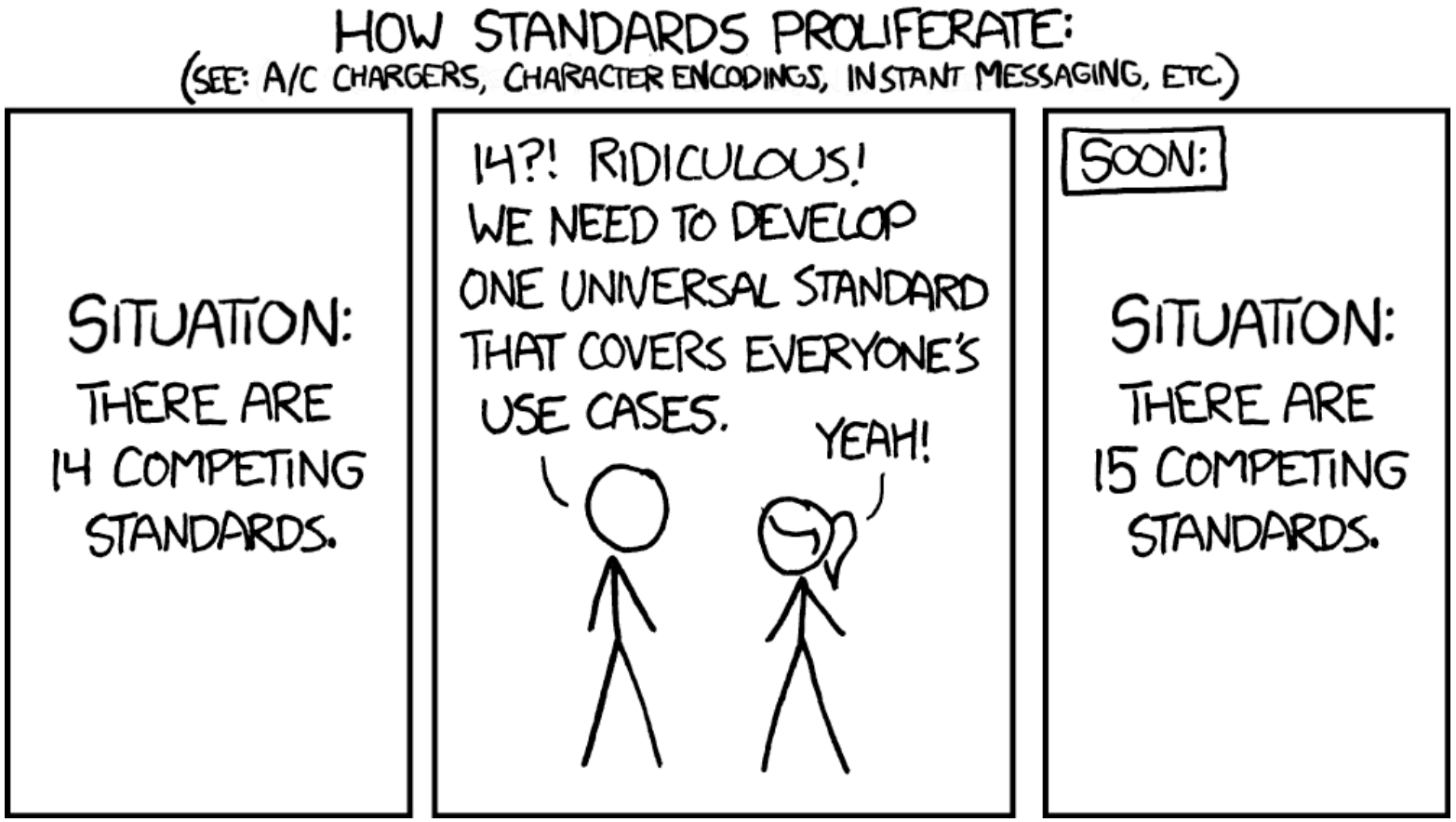
- Now only the `uint64_t` variant is documented
- The `uint32_t` variant is missing

PART 3

3

The Solution: A New Language-Neutral Intermediate Representation

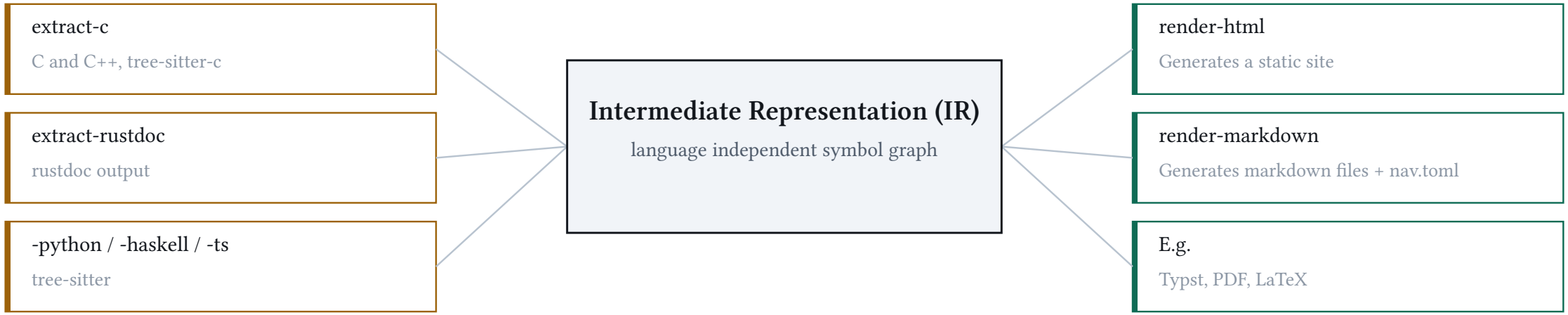
Also Known As



Author: xkcd, Source: https://xkcd.com/927/

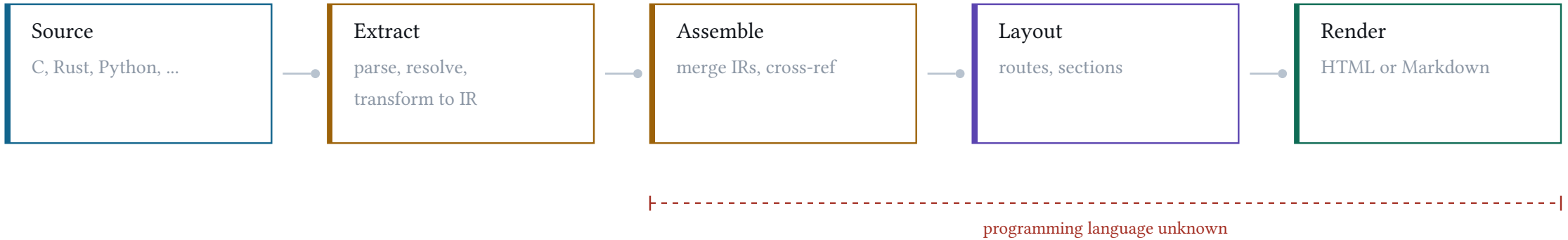
Architecture

- Extraction of code using tree-sitter and rustdoc output for Rust
- All extracted symbols share one IR
- Rendering is fully decoupled from the language specific extractors



The IR Does Not Know What Programming Language Is Used

After extraction the programming language is unknown



- Language specific terms e.g. `crate`, `module`, `namespace` are specified in the schema and filled by the extractor

Extract: Convert code to IR nodes

Assemble: Merge nodes of various extractors, resolve cross-refs

Layout: Transform IR into a more opinionated and configurable output format (generate nav, show or hide public fields, create routes)

Render: Transform layout to a markup language such as HTML, Markdown etc.

4

PART 4

Support Multiple Languages

SUPPORT MULTIPLE LANGUAGES

C and Rust Parsing and Merge

The parsed nodes land in a shared symbol graph, prefixed by the name of their collection (c, rust)

C AND RUST SOURCE CODE

```c
/** @brief Set a pin */
void gpio_write(gpio_t pin,
int value);
```

```rust
/// Drives the wrapped pin
/// via [`gpio_write`].
fn set_high(&mut self);
```

→

SYMBOL GRAPH

```
symbol ▸ collection "c"
 ├ c RIOT gpio_write()
 └ kind function

symbol ▸ collection "rust"
 ├ rust RIOT riot_wrappers/gpio
 │      /OutputGPIO#set_high()
 ├ kind method
 └ doc link ▸ gpio_write
```

→

RUST SET_HIGH HTML OUTPUT

```html
<h1>set_high</h1>
<pre>fn set_high(&mut self)</pre>
<p>
  Drives the wrapped pin via
  <a href="/c/…/periph_gpio/
#gpio_write">
    gpio_write
  </a>.
</p>
```

A Rust-to-C link is a normal reference, because they share the same IR graph

Language Specific Vocabulary

The IR defines various concepts that need translations per language:

KIND	C	RUST	PYTHON	HASKELL	TYPESCRIPT
`Package`	library	crate	package	package	package
`Module`	— (uses `Group`)	module (`mod`)	module	module	module
`Struct`	struct	struct	—	—	—
`Class`	—	—	class	—	class
`Interface`	—	trait	protocol (`Protocol`, `ABC`)	typeclass (`class`)	interface
`Instance`	—	trait impl (`impl`)	—	instance	—
`Enum`	enum	enum	enum (`class X(Enum)`)	data type (`data`, `newtype`)	enum
`Function`	function	function (`fn`)	function (`def`)	function	function
`Macro`	macro (`#define`)	macro (`macro_rules!`)	—	—	—
`Typedef`	type alias (`typedef`)	type alias (`type`)	type alias (`TypeAlias`)	type synonym (`type`)	type alias (`type`)

More Language Specific `Stuff`

The programming language is unknown to the output renderer, but it needs to know how to visualize different languages

IR FIELD	DECIDES	C	RUST
`Signature::return_position`	where the return type is written	`leading`	`trailing`
`Language::member_separator()`	how a member is joined to its owner	`.`	`::`
`Language::tag()`	the info string of a rendered code block	`c`	`rust`

SUPPORT MULTIPLE LANGUAGES

Guide Pages (Markdown / reStructuredText)

Any markup parses into the same typed tree and gets its own ID

MARKDOWN / RESTRUCTUREDTEXT SOURCE CODE

```
Set it with [`gpio_write`],
read the [pin state]
(sym:gpio_read).
```

OR

```
Set it with :func:`gpio_write`,
read the :func:`pin state
<gpio_read>`.
```

→

SYMBOL GRAPH

```
paragraph
 ├ text "Set it with "
 ├ link ▸ gpio_write
 │    └ code "gpio_write"
 ├ text ", read the "
 └ link ▸ sym:gpio_read
      └ text "pin state"
```

→

HTML OUTPUT

```
<p>
  Set it with
  <a href="/c/…/#gpio_write">
    <code>gpio_write</code>
  </a>,
  read the
  <a href="/c/…/#gpio_read">
    pin state
  </a>.
</p>
```

PART 5

5

Expose Configurations

EXPOSE CONFIGURATIONS

Extract Configurations from the Source Code

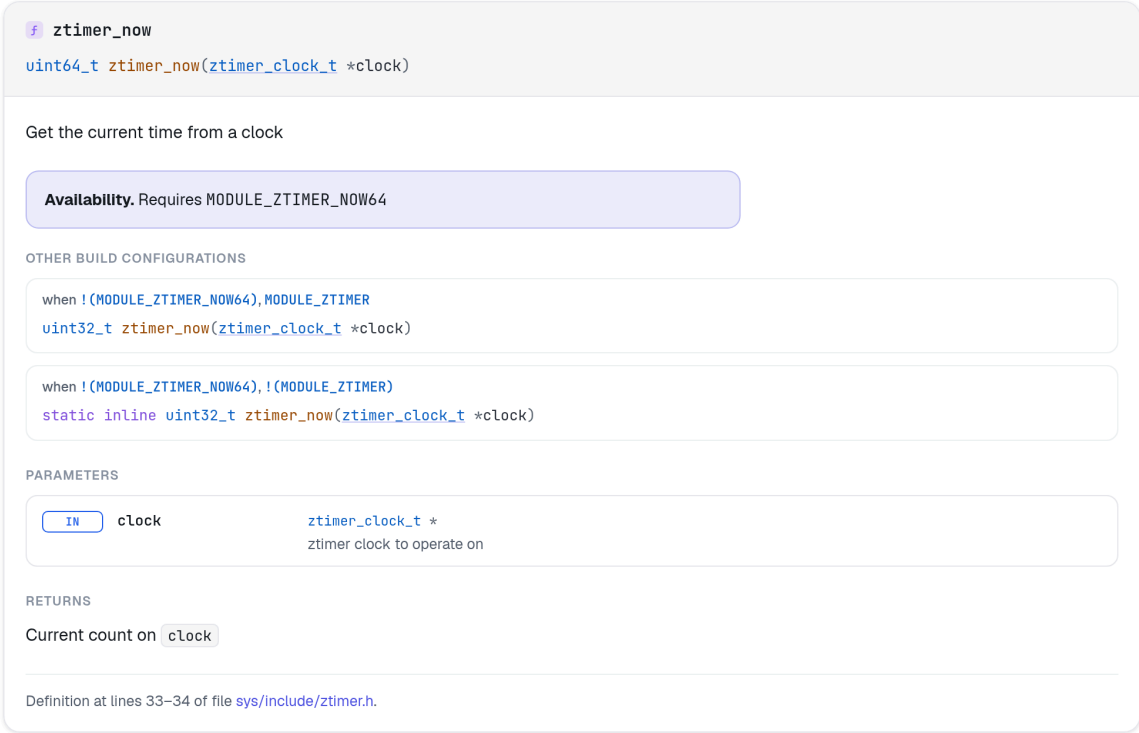
The C-parser converts the `#ifdef` macros into `conditions` in the new IR

This way the renderer can visualize them in the final output

IR SYMBOL

```yaml
id: c RIOT ztimer_now()
kind: function
declarations:
  - conditions: [MODULE_ZTIMER_NOW64]
    fragments: uint64_t ztimer_now(...)
  - conditions: ["!(MODULE_ZTIMER_NOW64)",
                 "MODULE_ZTIMER"]
    fragments: uint32_t ztimer_now(...)
  - conditions: ["!(MODULE_ZTIMER_NOW64)",
                 "!(MODULE_ZTIMER)"]
    fragments: static inline uint32_t
               ztimer_now(...)
```

HTML OUTPUT



PART 6

6 IR Structure

A short overview of the general structure of the new IR

A Graph of Symbols

The graph is implemented via flat lists

```
bundle:
    version: 1
    meta: { ... }          # project name, git-repo URL, etc.
    collections: [ ... ]   # the API spaces: Guides, C API, Rust API
    symbols: [ ... ]       # every documentable entity: function, method, type, etc.
    relationships: [ ... ] # member-of, in-group, inherits, overrides, etc.
    pages: [ ... ]         # guides, etc.
```

symbols: All documentable entities with unique IDs etc. (functions, methods, types)

relationships: Relations between symbols

pages: Markup files encoded as rich-text nodes with unique IDs

Stable IDs for Symbols

A grammar for building stable identifiers for symbols

```
<id>         ::= <language> ' ' <package> ' ' <descriptor>
<package>    ::= project or package name, or '.' if unknown
<descriptor> ::= (<name> <suffix>)+
<suffix>     ::= '/'    module or namespace
               | '#'    type: struct, class, union, enum, typedef
               | '()'   function or method
               | '.'    variable, constant, field
               | '!'    preprocessor macro
               | '%'    instance
               | ':'    group, file
```

E.g.:

SYMBOL ID	LANGUAGE	PACKAGE	NAME	SUFFIX	NAME	SUFFIX
`c RIOT gpio_init()`	`c`	`RIOT`	`gpio_init`	`()`		
`rust RIOT riot_wrappers/gcoap/`	`rust`	`RIOT`	`riot_wrappers`	`/`	`gcoap`	`/`
`c RIOT gpio_mode_t#GPIO_IN.`	`c`	`RIOT`	`gpio_mode_t`	`#`	`GPIO_IN`	`.`

7

PART 7

Schema to HTML

Collections

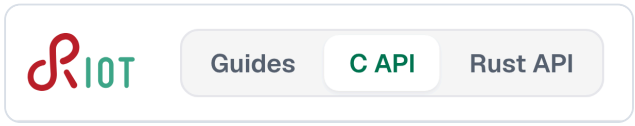
A collection is the documentation for a specific language such as markdown, rust, c

IN THE IR

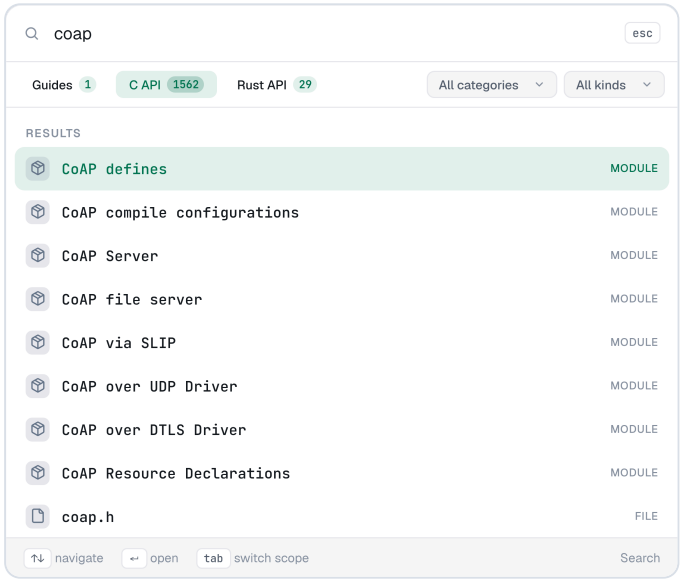
```
collections:
  - { id: guides, title: Guides,   position: 1 }
  - { id: c,      title: C API,    position: 2 }
  - { id: rust,   title: Rust API, position: 3 }

# and on every symbol:
collection: c
```

HTML: TABS



HTML: SEARCH



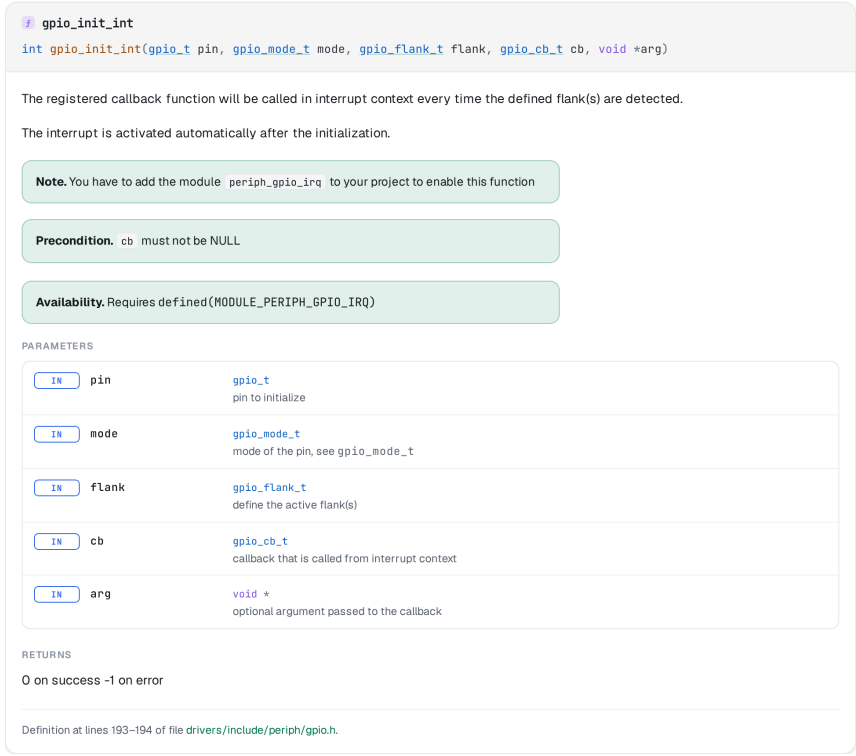
Symbols

Documentation of the `gpio_init_int(...)` function

IR SYMBOL

```yaml
id: c RIOT gpio_init_int()
kind: function
name: gpio_init_int
declarations:
  - fragments:
      - { kind: type_ref, text: int }
      - { kind: identifier,
          text: gpio_init_int }
      - ...
    conditions:
      - defined(MODULE_PERIPH_GPIO_IRQ)
    location:
      line: 193
      file: drivers/include/periph/gpio.h
```

HTML OUTPUT



ƒ **gpio_init_int**

```
int gpio_init_int(gpio_t pin, gpio_mode_t mode, gpio_flank_t flank, gpio_cb_t cb, void *arg)
```

The registered callback function will be called in interrupt context every time the defined flank(s) are detected.

The interrupt is activated automatically after the initialization.

Note. You have to add the module `periph_gpio_irq` to your project to enable this function

Precondition. `cb` must not be NULL

Availability. Requires `defined(MODULE_PERIPH_GPIO_IRQ)`

PARAMETERS

IN	pin	gpio_t pin to initialize
IN	mode	gpio_mode_t mode of the pin, see gpio_mode_t
IN	flank	gpio_flank_t define the active flank(s)
IN	cb	gpio_cb_t callback that is called from interrupt context
IN	arg	void * optional argument passed to the callback

RETURNS

0 on success -1 on error

Definition at lines 193–194 of file drivers/include/periph/gpio.h.

PART 8

8

Future Work

FUTURE WORK

Extract `USEMODULE += my_module` Requirements

Every RIOT API page should open with the `USEMODULE += my_module` line that turns it on

```
id: c RIOT gpio_init_int()
declarations:
  - conditions:
      - defined(MODULE_PERIPH_GPIO_IRQ)
    # derived from the condition above:
    enablement: USEMODULE += periph_gpio_irq
```

The extractor can not be RIOT specific, so either:

- A plugin API for special extraction rules
- A general `Makefile` parser that can be configured in the project config

FUTURE WORK

Document `dist/tools` in RIOT

RIOT: `dist/tools` is a mixed collection of small tools that also needs documentation

```
dist/tools/                      # ~100 small tools
├── ethos/                       # C + shell script + README.md
├── benchmark_udp/               # C + README.md
├── uhcpd/                       # C only, no README
├── pyterm/                      # Python
├── usb-serial/                  # Python + shell + README.md
├── tapsetup/                    # shell script
├── compile_and_test_for_board/  # Python + README.md
└── ...
```

A reader looks for **the tool**, not for its implementation language

- The IR is language independent => Collections can hold symbols with mixed languages
- Extractors need to get more flexible so that multiple can together extract from a shared folder

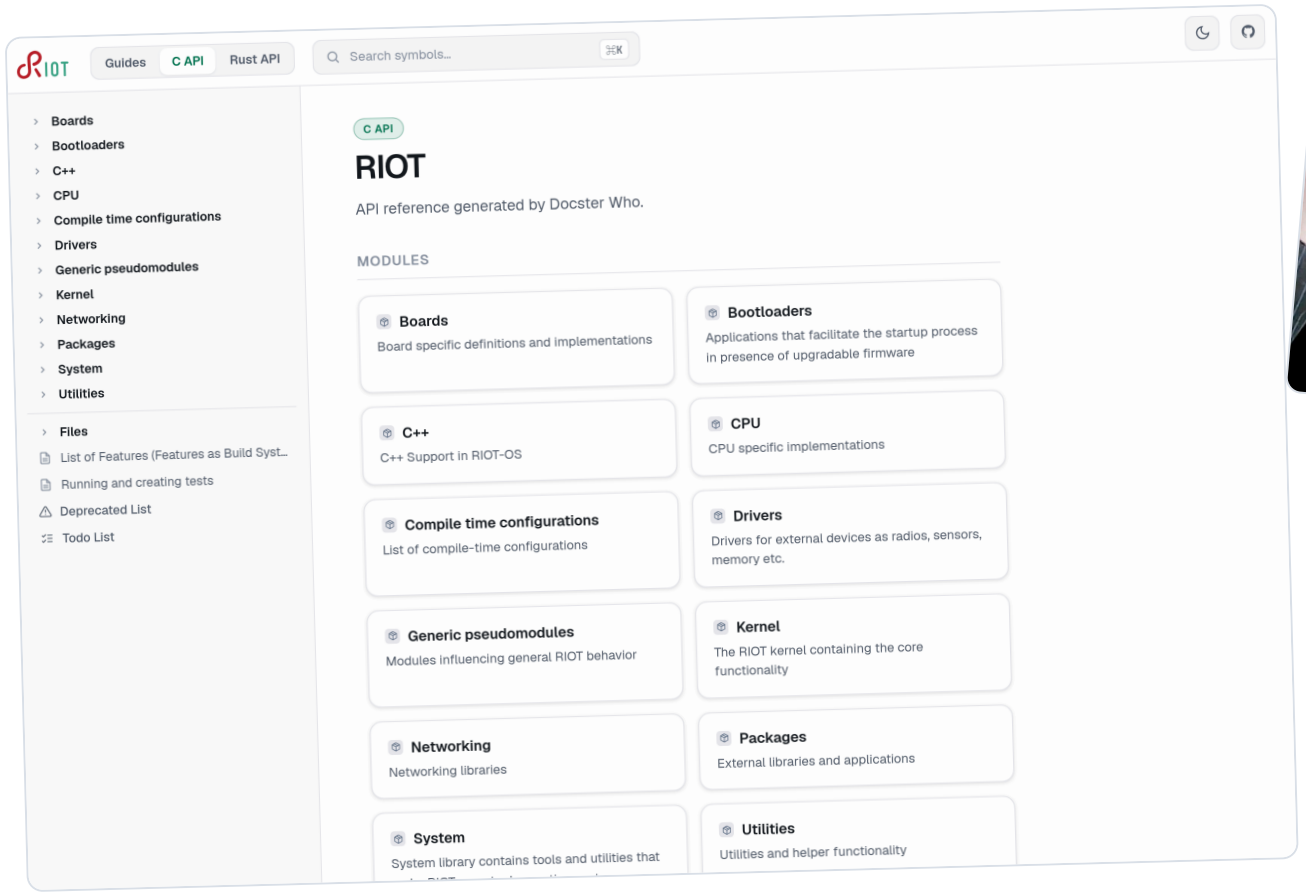
PART 9

9

Prototype Demo

PROTOTYPE DEMO

Introducing: `Docstor Who`



Source: Doctor Who (BBC)