RIOT-rs

Rust-based Configurations for RIOT

RIOT Summit – September 2023

Kaspar Schleiser *proxied by Emmanuel Baccelli*

Agenda

1. RIOT: Why? How? What?
2. Rust: Why?
3. Rust-based RIOT: How?
4. What is RIOT-rs?
5. RIOT-rs performance? A preliminary evaluation
6. Debating embedded Rust and RIOT: where to now?

RIOT: Why? How?

At the start, our goals were to provide

- An alternative to exotic programming (e.g. TinyOS) or closed-source (e.g. Zephyr)
- The 1st OS designed for low-power IPv6 (6LoWPAN/CoAP) standard network stack
- Prevention of vendor lock-in, empowering low-power IoT end-users

RIOT: Why? How?

At the start, our goals were to provide

- An alternative to exotic programming (e.g. TinyOS) or closed-source (e.g. Zephyr)
- The 1st OS designed for low-power IPv6 (6LoWPAN/CoAP) standard network stack
- Prevention of vendor lock-in, empowering low-power IoT end-users

Our approach has been

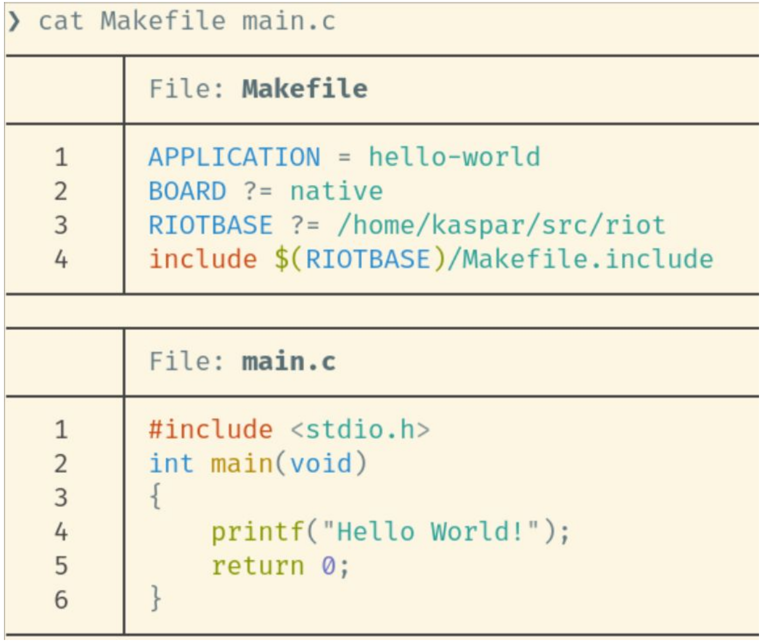
- OS architecture: microkernel & threading
- Standard coding: ANSI C
- Fully open source: rewrite vendor blobs
- Implementing – and contributing to – open network standards (IETF)
- Grassroots open source community processes

What is RIOT?

- **`core`**/: scheduling, mutex, ipc
- **`sys`**/: timers, networking, fs, …
- **`cpu`**/: MCU architecture support
- **`drivers/periph`**: peripheral drivers
- **`drivers`**/: sensor/network/misc drivers
- **`pkg`**/: third party code
- **`boards`**/*: board configuration
- **build system** (make, Kconfig…)

➔ *(A well-known general-purpose OS)*
➔ *(A lively open source community)*

Awesome Fact: this runs on 99% of our supported HW, just by changing BOARD



```
❯ cat Makefile main.c

File: Makefile

APPLICATION = hello-world
BOARD ?= native
RIOTBASE ?= /home/kaspar/src/riot
include $(RIOTBASE)/Makefile.include


File: main.c

#include <stdio.h>
int main(void)
{
    printf("Hello World!");
    return 0;
}
```

Ceilings with RIOT now

Hitting limits w.r.t. security

- Making mem protection + MPU first class citizens
- Providing configuration(s) with “defensive” code
- Catching errors: Graceful shutdown / restart of threads

Ceilings with RIOT now

Hitting limits w.r.t. security

- Making mem protection + MPU first class citizens
- Providing configuration(s) with “defensive” code
- Catching errors: Graceful shutdown / restart of threads

Hitting limits w.r.t. programming & maintenance

- Bound to the limits of C (API design, safety, abstractions, tooling, …)
- Dealing with the toolchain mess
- Peoplepower for CI & maintenance of system’s (un)controlled growth

Ceilings with RIOT now

Hitting limits w.r.t. security

- Making mem protection + MPU first class citizens
- Providing configuration(s) with “defensive” code
- Catching errors: Graceful shutdown / restart of threads

Hitting limits w.r.t. programming & maintenance

- Bound to the limits of C (API design, safety, abstractions, tooling, …)
- Dealing with the toolchain mess
- Peoplepower to CI & maintain the system’s growth (uncontrolled?)

If left unaddressed, RIOT could drift into becoming subpar

➔ Which long-term direction should explore from here ??

Agenda

1. RIOT: Why? How? What?
2. Rust: Why?
3. Rust-based RIOT: How?
4. What is RIOT-rs?
5. RIOT-rs performance? A preliminary evaluation
6. Debating embedded Rust and RIOT: where to now?

Enter Rust

The “new” kid on the block, challenging C…

… with a different trade-off combining:

- High-level ergonomics;
- Built-in memory safety;
- Low-level control;

With modern tooling (build with *cargo*, import *crates*)...

Recent Rust rant: see this post on Google Open Source Blog

Enter Rust

The “new” kid on the block, challenging C…

… with a different trade-off combining:

- High-level ergonomics;
- Built-in memory safety;
- Low-level control;

With modern tooling (build with *cargo*, import *crates*)...

Recent Rust rant: see this post on Google Open Source Blog

➔ What we need to fix our problems on embedded ??

(We already have Rust wrappers)

(We already use Rust drivers on some boards)

What Expectations with (Much) More Rust?

Technical

- Inherent memory safety, without (much?) performance loss
- Workflow changes (stop chasing whole categories of mean bugs)
- More modern tooling (lean & mean)

What Expectations with (Much) More Rust?

Technical

- Inherent memory safety, without (much?) performance loss
- Workflow changes (stop chasing whole categories of mean bugs)
- More modern tooling (lean & mean)

Non-technical

- Further differentiate from (deep-pocketed) Zephyr / FreeRTOS
- Potential synergy with (lively) embedded Rust movement

Embedded Rust: What’s Out There Already?

Quite a bit, and growing:

- Drivers, crypto libs…
- Hardware abstraction (e.g. *embedded-hal*)
- Network abstraction (e.g. *embedded-nal*)
- Network stack (e.g. *smoltcp*)
- Framework for **embedded async** Rust (e.g. *Embassy*)
- Full-fledged operating system (e.g. *Tock-OS*)

Intermediate Summary

Fact: Rust picked up steam, for good reasons

➔ not only in Linux & unconstrained, but also on embedded & constrained devices!

Question: Could (much more) Rust fix our problems?

➔ What would (much more) Rust look like ??

Agenda

1. RIOT: Why? How? What?
2. Rust: Why?
3. Rust-based RIOT: How?
4. What is RIOT-rs?
5. RIOT-rs performance? A preliminary evaluation
6. Debating embedded Rust and RIOT: where to now?

Alternatives for (much more) Rust

In the context of a research project RIOT-fp * we considered different experiments

1. ~~Prototype RIOT scheduler + RIOT apps on top of TockOS~~
2. Incremental rewrites of core RIOT modules in Rust
3. Prototype RIOT over async Rust framework (Embassy)

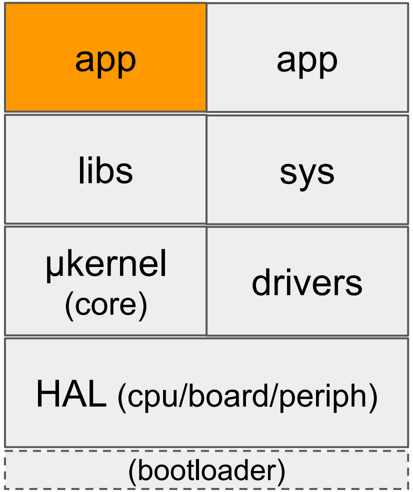
No standard Rust async,
Not lib-oriented,
100% MPU-dependent,
can’t replicate RIOT
scheduling semantics…

* See online https://future-proof-iot.github.io/RIOT-fp/about

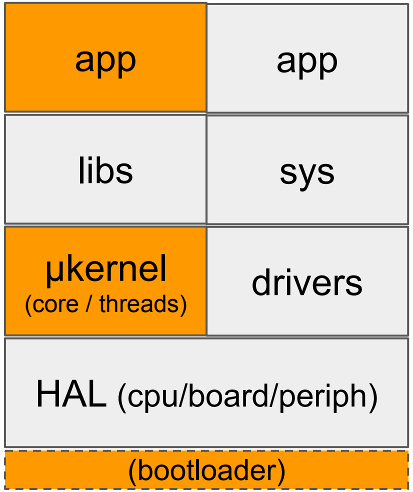
Re-write of core RIOT in Rust

⬜ *C*

🟧 *Rust*



RIOT + Rust wrappers
(**C configs**)



Cargo-built RIOT
(**with Rust core**)

Re-write of core RIOT in Rust

After several rewrites of core (task switching) in Rust… we observe that

- Build system modification is the big chunk
  - rabbit hole starts with driving the build with cargo and Rust needing LLVM…
  - leads to even more messy than RIOT current build system…

Long story short, based on our experience during our research project:

- *Not worth it* for just “a Rust core” (*vs* Rust wrappers for select modules)

Re-write of core RIOT in Rust

After several rewrites of core (task switching) in Rust… we observe that

- Build system modification is the big chunk
  - rabbit hole starts with driving the build with cargo and Rust needing LLVM…
  - leads to even more messy than RIOT current build system…

Long story short, based on our experience during our research project:

- *Not worth it* for just “a Rust core” (*vs* Rust wrappers for select modules)

➔ But perspectives include proofs* on functional Rust (RIOT module rewrites)

* e.g. based on Hax, see https://github.com/hacspec/hacspec-v2 (collaboration during RIOT-fp project)

Alternatives for (much more) Rust

In the context of a research project RIOT-fp * we considered different options

1. ~~Prototype RIOT scheduler + RIOT apps on top of TockOS~~
2. Incremental rewrites of core RIOT modules in Rust
3. Prototype RIOT over async Rust framework (Embassy)

* See online https://future-proof-iot.github.io/RIOT-fp/about

About Embassy (and smoltcp)

Significant community active at https://github.com/embassy-rs/embassy

***What does it provide** we care about?*

- Based on async Rust => naturally concurrent, no need for main loop
- HAL, timers, real-time, low-power, bluetooth, LoRa, USB, Bootloader + DFU, …

About Embassy (and smoltcp)

Significant community active at https://github.com/embassy-rs/embassy

***What does it provide** we care about?*

- Based on async Rust => naturally concurrent, no need for main loop
- HAL, timers, real-time, low-power, bluetooth, LoRa, USB, Bootloader + DFU, …

***What does it not provide** that we *really* care about?*

- Implementation
  - 6LoWPAN/CoAP/OSCORE/RPL/… (IPv6 low-power stack)
  - Multiple timers (e.g., low-power *and* high frequency)
  - Threading
  - Secure standard OTA (SUIT?)
  - …
- Architectural / Integration
  - Application portability – even blinky code with Embassy is board-specific…
  - (On an arbitrary board, a relatively small time-to-hacking)
- Policy / Community Processes
  - Blob avoidance (e.g., drop softdevice?)

About Embassy (and smoltcp)

Significant community active at https://github.com/embassy-rs/embassy

What does it provide we care about?

- Based on async Rust => naturally concurrent, no need for main loop
- HAL, timers, real-time, low-power, bluetooth, LoRa, USB, Bootloader + DFU, …

*What does it not *really* do that we *really* care about?*

- Implementation
  - 6lowpan/CoAP/OSCORE/RPL/… (IPv6 low-power stack)
  - Multiple timers (e.g., low-power *and* high frequency)
  - Threading
  - Secure standard OTA (SUIT?)
  - …
- Architectural / Integration
  - Application portability – even blinky code with Embassy is board-specific…
  - (On an arbitrary board, a relatively small time-to-hacking)
- Policy / Community Processes
  - Blob avoidance (e.g., drop softdevice, port nimBLE?)

So what about Embassy + RIOT threads?

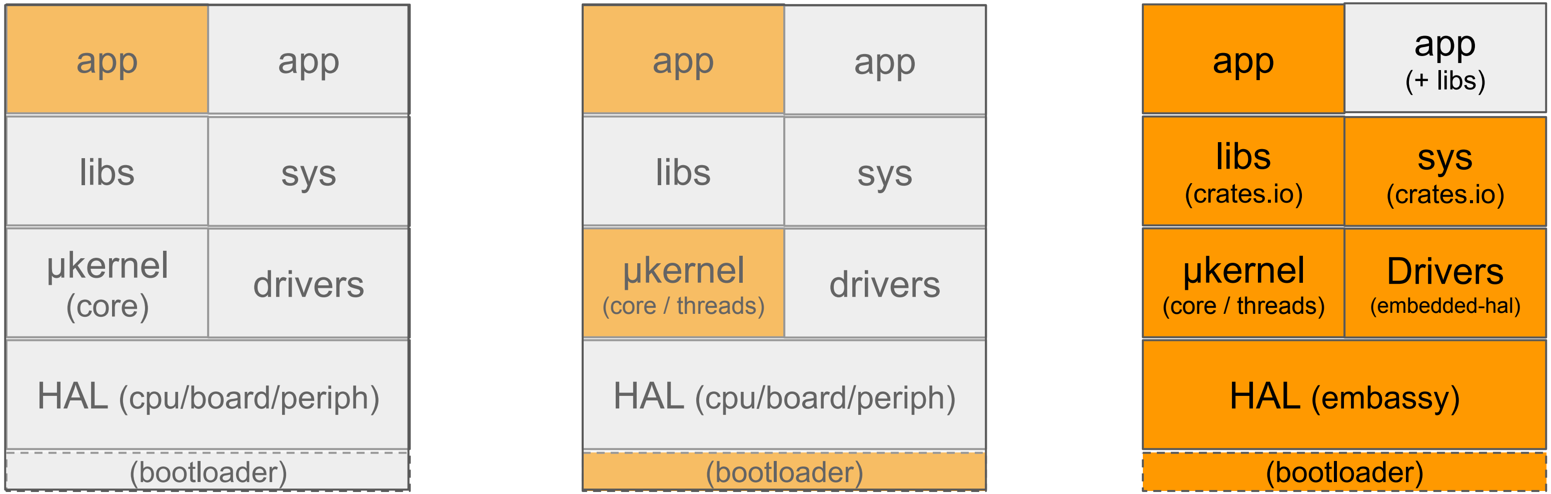
Agenda

1. RIOT: Why? How? What?
2. Rust: Why?
3. Rust-based RIOT: How?
4. What is RIOT-rs?
5. RIOT-rs performance? A preliminary evaluation
6. Debating embedded Rust and RIOT: where to now?

RIOT based on Async Rust

⬜ *C*

🟧 *Rust*



app | app
libs | sys
µkernel (core) | drivers
HAL (cpu/board/periph)
(bootloader)

RIOT + Rust wrappers
(C configs)

app | app
libs | sys
µkernel (core / threads) | drivers
HAL (cpu/board/periph)
(bootloader)

Cargo-built RIOT
(C with Rust core)

app | app (+ libs)
libs (crates.io) | sys (crates.io)
µkernel (core / threads) | Drivers (embedded-hal)
HAL (embassy)
(bootloader)

RIOT-rs
(Rust-based configs)

RIOT-rs prototype, in other words

`core`/	riot-rs-core
`sys`/	embassy-time, embassy-net+smoltcp
`cpu`/	embassy-nrf, -rp, -esp, ...
`drivers`/`periph`	embedded-hal
`drivers`/	embedded-hal
`pkg`/	crates.io + pkg to integrate 3rd party
`boards`/*	–
build system	Cargo-driven

RIOT-rs Prototype

- Re-used RIOT Rust scheduler rewrite providing RIOT semantics
  - Embassy HAL kicks in at initialisation, RIOT threads then run on the side
  - C API bindings

- Main challenges addressed with the build system:
  - Matching ~10 lines for build system & code for RIOT basic application!
    - Cargo doesn’t do “BOARD=...”, only “--target thumbv7em-none-eabi”, needing the application Cargo.toml to specify board specifics
    - Embassy has arch specific initialization (nrf, rp, rsp)
  - 1st shot at integration:
    - riot-rs crate: going through standard hoops to select correct cpu/board/embassy setup
    - wrapped Cargo in laze, allows “`laze build --builder nrf52840dk`” to nudge Cargo right

Agenda

1. RIOT: Why? How? What?
2. Rust: Why?
3. Rust-based RIOT: How?
4. What is RIOT-rs?
5. RIOT-rs performance? A preliminary evaluation
6. Debating embedded Rust and RIOT: where to now?

What We Can Say about RIOT-rs Prototype (so Far)

The implemented prototype works on a couple of different Cortex-M boards

➔ see code at https://github.com/future-proof-iot/RIOT-rs
➔ ready for porting to other cpu (RISC-V) and other boards

Preliminary micro-benchmarks of **RIOT-rs *vs* RIOT-C**

➔ core/threads have **almost identical RAM/ROM/perf**
➔ e.g., “thread_flags” has same performance

Some remarks/observations:

1. Rust needs LTO, code size otherwise huge
2. RIOT-c GCC+lto optimizes *very* well (bar is high ;)
3. Non-trivial code size comparison difficult due to issues with LLVM-only RIOT-C, which is necessary for XLTO

What We Can Say about RIOT-rs Prototype (so Far)

- based on RIOT-rs core
  - close-at-hand: implement MPU-based sandboxing for threads
  - also within reach: multicore support (prototype has initial multicore support for raspi-pico)

- based on prototype integration
  - close-at-hand: board specific (sensor) driver selection

Agenda

1. RIOT: Why? How? What?
2. Rust: Why?
3. Rust-based RIOT: How?
4. What is RIOT-rs?
5. RIOT-rs performance? A preliminary evaluation
6. Debating embedded Rust and RIOT: where to now?

The Horizon with Rust ?

(**from our perspective**, based on RIOT-rs experiments)

- **We could retain the awesome sides of RIOT!**
  - Application portability, “batteries-included”
  - Smooth transition seems possible, without loss of our (rich) functionalities
- **We can improve embedded Rust**
  - Provide fully integrated system and distrib. (building on a decade of RIOT experience)

The Horizon with Rust ?

(**from our perspective**, based on RIOT-rs experiments)

- **We could retain the awesome sides of RIOT!**
  - Application portability, “batteries-included”
  - Smooth transition seems possible, without loss of our (rich) functionalities
- **We can improve embedded Rust**
  - Provide fully integrated system and distrib. (building on a decade of RIOT experience)
- **We could fix some critical RIOT bottlenecks**
  - Better share burden of HAL, periph/driver devel. & maintenance
  - Rationalize our broad, but uneven HW support
  - More modern tooling & ergonomics **: increased productivity in the long-run?**
- **We can gain security guarantees**
  - Memory safety
  - (Proofs “for free” on a perimeter of critical modules e.g., “core/thread is panic-free”)

A Step Back, Up for Debate

Is C is the future? Most probably not.

Is Rust the future? Could be!

Independently: memory safety is not a SHOULD. It’s a MUST.

Do we have the resources to tend towards memory safe RIOT-C? Most probably not.

What should we do about that ?

A Step Back, Up for Debate

Is C is the future? Most probably not.

Is Rust the future? Could be!

Independently: memory safety is not a SHOULD. It's a MUST.

Do we have the resources to tend towards memory safe RIOT-C? Most probably not.

What should we do about that ?

We already support + partly depend on Rust.

Should we embrace (much) more Rust ?

- If so how?
- Where do we want to be in 3-5 years from now?

That’s all folks! Time for Q&A

(The key questions are in the previous slide ;)

RIOT-rs prototype code

More info on the RIOT-fp research project