Building and Integrating Rust libraries for RIOT-OS, with Micropython support
(and FreeRTOS)

Michael Richardson
<mcr@sandelman.ca>
Ottawa, Canada

Yoichi J. Nakaguro
<ynaka96@gmail.com>
Thailand

ESP32
Overview of Talk

Goals

Technologies

Target Audience

RUST Challenges

Micropython Challenges

Demo github(s)
Goals of Project

1) get some secure onboarding upstream into RIOT-OS and FreeRTOS
2) Rust-based RFC8366 voucher library for embedded (no_std) and regular (std) use
3) Integrate with micropython as the top-level control loop, so that policy is in the end-user space
4) The next revolutionary product will come from amateurs, get them good tools
Technologies

- RFC8366 (vouchers)
- RFC8995 (BRSKI)
  - See https://brski.org/
- RUST
  - For memory safety,
  - Thread safety
  - It’s the future
- ESP32
  - https://www.adafruit.com/product/3405
  - Specs:
    - 240 MHz dual core Tensilica LX6 microcontroller with 600 DMIPS
    - Integrated 520 KB SRAM
    - Integrated 802.11b/g/n HT40 Wi-Fi transceiver, baseband, stack and LWIP
    - Integrated dual mode Bluetooth (classic and BLE)
    - 4 MByte flash include in the WROOM32 module
- FreeRTOS
  - Because it’s there
- RIOT-OS
  - Because it’s cool
Target Audience

- The “Hello World” application
  - Batteries included security
  - Onboarding included
- Needs to be upstreamed everywhere, always available, always tested, compiled.
- The garage/small enterprise IoT experimenter
  - Doing a “proof of concept”
  - Winds up shipping/in production
- Target system code must be Royalty-Free (RF) license to reduce friction
- Seats, support, Registrar, onboarding, provisioning systems will cost money
  - But, open standards, people can roll their own, if they do not like ours.
Project Architecture

- Provide a series of libraries to do heavy lifting
- Keep top-level policy in python, where end developers can see and change the policy
- Provide reference python code to do onboarding
  - Provide to call home, softAP, and other fallback methods
ESP32 challenges (opportunities)

- Provisioning of IDevID into devices.
  - Looking for standardized way to store private key, certificates
  - Trust Anchor for SUIT
- Needs to be compatible with industry trends for provisioning.
  - Many different methods, but needs to result in same layout and content
- First firmware burn should be SUIT based.
- Provisioned anchors and configuration space perhaps should be CBOR?
- Xtensa has some secured/measured boot mechanisms which need further examination
Project Status

The Good

- RUST voucher library 90%
- Front-end-loaded integration to FreeRTOS and RIOT-OS
- Many libraries adapted and made no_std
- Using QEMU-Xtensa, but also RIOT-OS native build for testing

The Bad

- Micropython not connected to gnrc network on RIOT-OS
  - (yes on FreeRTOS)
- Cargo build not integrated into RIOT-OS, done by linking in Cargo “staticlib”, that is, a lib.a
RUST integration challenges

- Dependencies (no_std!). Many useful crates in Crates.io are not no_std aware; we often need to adapt them with a clear scope.

- Looking for good compromise between no_std and std: "semi_std" for e.g. dynamic containers, smart pointers, basic IO operations.

- We made available an interface crate "mcu-if" (details in Case Study)

```rust
#![no_std]
use mcu_if::println;
use mcu_if::alloc::{boxed::Box, vec, vec::Vec};
use mcu_if::core2::io::{self as io, Cursor, Seek, SeekFrom, Write};
```

- Cargo builds a static library, and then RIOT-OS links from this.

```bash
INCLUDES += -I$(CURDIR)/../include
ARCHIVES += $(CURDIR)/../target/xtensa-esp32-none-elf/release/librustmod.a
```
<table>
<thead>
<tr>
<th>RIDT-OS vs FreeRTOS</th>
<th>ANIMA Minerva perspective</th>
</tr>
</thead>
<tbody>
<tr>
<td>Framework</td>
<td></td>
</tr>
<tr>
<td>RIOT 2021.07</td>
<td>ESP-IDF 4.2 (by Espressif)</td>
</tr>
<tr>
<td>Build type</td>
<td></td>
</tr>
<tr>
<td>esp32, native</td>
<td>esp32</td>
</tr>
<tr>
<td>Networking</td>
<td></td>
</tr>
<tr>
<td>GNRC</td>
<td>LwIP</td>
</tr>
<tr>
<td>in micropython</td>
<td></td>
</tr>
<tr>
<td>We plan to working on this!</td>
<td>Works (WIFI, LAN)</td>
</tr>
<tr>
<td>esp32.bin</td>
<td></td>
</tr>
<tr>
<td>~100 KB (small and build is fast)</td>
<td>~300 KB</td>
</tr>
<tr>
<td>C libraries (libc)</td>
<td></td>
</tr>
<tr>
<td>Compatible: io – printf, mem – malloc/free, etc.</td>
<td></td>
</tr>
<tr>
<td>C libraries (crypto)</td>
<td></td>
</tr>
<tr>
<td>mbedtlss coming</td>
<td>Has mbedtlss</td>
</tr>
<tr>
<td>Rust</td>
<td></td>
</tr>
<tr>
<td>Compatible! We verified no_std/”semi_std” Rust works fine</td>
<td></td>
</tr>
<tr>
<td>RIOT ‘native’ board and testing</td>
<td>MCU emulators</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>RIOT board type</td>
<td>esp32</td>
</tr>
<tr>
<td>Arch</td>
<td>xtensa</td>
</tr>
<tr>
<td>MCU</td>
<td>real device</td>
</tr>
<tr>
<td>Memory emulation</td>
<td>~300 KB</td>
</tr>
<tr>
<td>SPI emulation</td>
<td>Perfect (of course)</td>
</tr>
<tr>
<td>dev/debug cycle</td>
<td>SLOW (need flashing)</td>
</tr>
<tr>
<td>Notes</td>
<td>- net if: WIFI, (LAN)</td>
</tr>
<tr>
<td></td>
<td>- CI: near impossible</td>
</tr>
<tr>
<td></td>
<td>- Production test: Yes!</td>
</tr>
<tr>
<td>ESC emulators</td>
<td>esp32</td>
</tr>
<tr>
<td>x86</td>
<td>qemu-xtensa</td>
</tr>
<tr>
<td>32-bit Linux</td>
<td>~300 KB; OK</td>
</tr>
<tr>
<td>?</td>
<td>OK (dev/debug)</td>
</tr>
<tr>
<td>N/A</td>
<td>OK</td>
</tr>
<tr>
<td>Great</td>
<td>OK</td>
</tr>
<tr>
<td>Great</td>
<td>Great</td>
</tr>
<tr>
<td>- net if: LAN</td>
<td>- net debug tap: OK</td>
</tr>
<tr>
<td>- CI: Great</td>
<td>- Rust toolchain setup: Hard</td>
</tr>
<tr>
<td>- GNRC experiments: Yes!</td>
<td>- Rust toolchain: Easy</td>
</tr>
<tr>
<td>- CI: Great</td>
<td>- net if: LAN</td>
</tr>
<tr>
<td>- net debug tap: Great</td>
<td>- net debug tap: Great</td>
</tr>
<tr>
<td>- GNRC experiments: Yes!</td>
<td>- Rust toolchain: Easy</td>
</tr>
</tbody>
</table>
Case study: 0) reference implementation repo

Name will probably change

Robust IoT development with Rust and RIOT-OS.

Languages
- Rust 60.0%
- Makefile 32.7%
- Shell 7.3%

Repository map

/  
  | README.md
  |    crates/
  |     mcu-emu
  |     mcu-if
  |     examples/
  |      esp32-no_std
  |      xbd-base
  |      xbd-micropython
  | ...

... currently supports mcu's specific to esp32 (and Linux native) only
... emulator runner ('gemu-system-xtensa' or RIOT native board binary)
... "semi_std" interface on top of bare `no_std`
... bare `no_std` firmware with a Rust module
... cross-`BOARD` (esp32/native) firmware with minimal 'librustmod.a'
... cross-`BOARD` firmware featuring MicroPython with 'libvoucher.a'

Environments

Ubuntu 20.04 is supported and also being used for CI.

https://github.com/AnimaGUS-minerva/iot-rust-module-studio
Case study: 0) clone and init the repo

Getting started

After cloning the repo, first, set up the pre-configured RIOT/ESP32 toolchain:

$ make init
Case study: 1) examples/esp32-no_std – build and run `esp32` firmware

We can flash this onto the real ESP32 device too!

```bash
make
```

```make
Cargo.toml
Makefile
include
├── rustmod.h
│
├── riot
│   ├── Makefile
│   └── main.c
├── src
│   └── lib.rs
```
mcu-if implements "semi_std" features
- ![panic_handler]
- ![global_allocator]
- ![alloc_error_handler]
- `println!()`
- `core2::io::*` (substitute of `std::io::*`)

```
use mcu_if::println;
use mcu_if::alloc::boxed::Box, vec, vec::Vec;
use mcu_if::core2::io::(self as io, Cursor, Seek, SeekFrom, Write);

#[no_mangle]
pub extern fn square(input: i32) -> i32 { }
```
Case study: 2) examples/xbd-base – build and run `native` firmware

```
<table>
<thead>
<tr>
<th>text</th>
<th>data</th>
<th>bss</th>
<th>dec</th>
<th>hex</th>
<th>filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>53329</td>
<td>1460</td>
<td>47792</td>
<td>102581</td>
<td>190b5</td>
<td>/home/runner/work/iot-rust-module-studio/iot-rust-module-studio/examples/xbd-base/riot/bin/native/riot.elf</td>
</tr>
</tbody>
</table>

make[5]: Leaving directory `/home/runner/work/iot-rust-module-studio/iot-rust-module-studio/examples/xbd-base/riot`

```/riot/bin/native/riot.elf & file ./riot/bin/native/riot.elf
linux-gate.so.1 (0x7f7f97000)
libdl.so.2 => /lib32/libdl.so.2 (0x7f7f5e000)
libgcc_s.so.1 => /lib32/libgcc_s.so.1 (0x7f7f3f000)
libc.so.6 => /lib32/libc.so.6 (0x7f7d54000)
/lib/ld-linux.so.2 (0x7f7f90000)
./riot/bin/native/riot.elf: ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV), dynamically linked, interpreter /lib32/ld-linux.so.2, BuildID[sha1]=bf1a758c79a77f4adffbb7277c6835fe3aa0b57, for GNU/Linux 3.2.0, with debug_info, not stripped

```

```
RIOT native interrupts/signals initialized.
LED_RED_OFF
LED_GREEN_ON
RIOT native board initialized.
RIOT native hardware initialization complete.

main(): [This is RIOT] (Version: 2021.04)
riot: RIOT_BOARD: native
riot: RIOT MCU: native
riot: before calling rustmod
[src/lib.rs] square() input: 4
Box: new(42): 42
Box: new([0; 10]): [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Vec: from([0, 1, 2]): [0, 1, 2]
vec([0, 1, 2]): [0, 1, 2]
buf: Cursor { inner: [0, 0, 0, 0, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9], pos: 15 }
check: Cursor { inner: [0, 0, 0, 0, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
riot: after calling rustmod
riot: square(4) => 16
```
Case study: 2) examples/xbd-base - build and run `esp32` firmware

```
$ make build-esp32
$ make run-esp32
```

```
I (747) boot: End of partition table
I (749) esp_image: segment 0: paddr=0x00010020 vaddr=0x3f40000 size=0x1000 size=0x0100 (10464) map
I (760) esp_image: segment 1: paddr=0x00012908 vaddr=0x3f80000 size=0x1134 (4404) load
I (769) esp_image: segment 2: paddr=0x0001a44 vaddr=0x4000000 size=0x4328 (1792) load
I (782) esp_image: segment 3: paddr=0x00017d74 vaddr=0x400c000 size=0x00064 (100) load
I (788) esp_image: segment 4: paddr=0x00017de0 vaddr=0x400e000 size=0x008230 (33328)
I (805) esp_image: segment 5: paddr=0x00020018 vaddr=0x400d000 size=0x09284 (37508) map
I (828) boot: Loaded app from partition at offset 0x1000
I (829) boot: Disabling RNG early entropy source...

main(): This is RIOT! (Version: 2021.04)
riot: RIOT_BOARD: esp32-wroom-32
riot: RIOT_MCU: esp32
riot: before calling rustmod
[src/lib.rs] square(): input: 4
Box::new(42): 42
Box::new([8; 10]): [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Vec::from([0, 1, 2]): [0, 1, 2]
vec![0, 1, 2]: [0, 1, 2]
buf: Cursor { inner: [0, 0, 0, 0, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9], pos: 15 }
check: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
riot: after calling rustmod
riot: square(4) -> 16
```
Case study: 3) examples/xbd-micropython - structure

- Voucher (RFC8366) Library Integration
- MicroPython Integration

1. Cargo.toml
2. makefile
3. mcu-emu
4. xbd-base

- Example directory structure
- Makefile
- Cargo.toml
- Include directory
- MicroPython module integration
- Voucher library integration
MicroPython status and our actions

- [upstream] micropython/ports/riot is missing; while [kaspar030] micropython/ports/riot is under dev

  >>> Updated kaspar's branch for latest v1.16 micropython
  >>> Extended kaspar's branch to be multi-arch (esp32 tested)
  >>> Probably PR to kaspar and help merge ports/riot into upstream micropython!

- Recent v1.16 release has ESP32 ethernet support that works well in FreeRTOS with ESP-IDF v4.2

  >>> Planning to connect RIOT's GNRC layer to Micropython's ethernet LAN api
  >>> For this to happen, we made sure our codebase is compatible with both esp32 AND native.
  RIOT's native board is awesome for dev/debug purpose!
Case study: 3) examples/xbd-micropython – Voucher (RFC8366) Library integration

```bash
$ cargo tree

xbd-micropython
  └── cose v0.1.4 (fork of franziskuskiefer/cose-rust)
      └── moz_cbor v0.1.2 (fork of franziskuskiefer/cbor-rust)
          └── core2 v0.3.0-alpha.1 (fork of technocreatives/core2)
              └── memchr v2.4.1

mcu-if
  └── core2 v0.3.0-alpha.1 (fork of technocreatives/core2)
      └── memchr v2.4.1
          └── libc-print v0.1.16 (fork of mmastrac/rust-libc-print)
```

- Using awesome 3rd-party crates
- Retaining necessary std-like features
Case study: 3) examples/xbd-micropython – run test and REPL (native build)

- **Makefile**
  - crates/
    - mcu-emu
    - mcu-if
  - examples/
    - esp32-no_std
    - xbd-base
    - xbd-micropython

- **Cargo.toml**
  - include
    - voucher.h

- **riot/Makefile**
  - boot.py
  - custom
  - modvel

- **main.c**
  - `main`: This is RIOT! (Version: 2021.04)
  - `sizeof(mp_heap)`: 131072
  - Executing `boot.py`

- **Debug in REPL**
  - `vch = voucher.get_voucher_F2_00_02()`

- **Rust-based COSE decoder in action!**
  - `vch = voucher.get_voucher_F2_00_02()`
Case study: 3) examples/xbd-micropython – run test (esp32 build)

We can flash this onto the real ESP32 device too!
Conclusions

- Ongoing work
  - Connect GNRC to micropython in `native` (then `esp32`)
  - Compact no_std Rust interface/binding to the *latest* mbedtls library (we recently saw mbedtls pkg coming to RIOT!)

- Please expect pull requests

- Looking for discussion of security architecture
  - How much can we do on multiple boards?
  - Can we do variations with some abstraction API?
  - Some boards have Bootloaders, others do not