Pilot Project

Rust for FPBGA-SoC with RISC-V: Evaluation on security and performance

Jonas Möwes

12mojo1bif@hft-stuttgart.de

2025-03-21

1. Introduction

Building Rust for RISC-V in a bare-metal environment requires a clear understanding of the necessary configurations, toolchains, and runtime considerations. This document provides an overview of key aspects involved in setting up Rust for RISC-V without an operating system and in combination with RTEMS, a real-time operating system.

Rather than serving as a step-by-step tutorial, this document outlines:

- Essential configurations required to compile Rust for RISC-V.
- Target specifications and toolchain setup for bare-metal execution.
- Memory management and allocator considerations in no_std environments.
- QEMU as a simulation platform for RISC-V targets.
- RTEMS integration, including BSP selection and linking Rust with RTEMS libraries.

By exploring these topics, this document aims to provide a high-level understanding of what is required to build and run Rust in these environments.

2. Building Rust for RISC-V

2.1. RISC-V

RISC-V is an open standard instruction set architecture (ISA) defined by the RISC-V Foundation. It is designed with a modular philosophy, consisting of a small base integer instruction set plus optional extensions like M (multiply/divide), A (atomic operations), F and D (floating-point), and more. This flexibility makes RISC-V suitable for everything from microcontrollers to high-performance computing. Because the ISA is open and free to implement, it has spurred innovation and adoption in academic, industrial, and hobbyist contexts. When building Rust for RISC-V, it is critical to select the correct variant of the base ISA and extensions supported by your target device or emulator.

2.2. Rustc platform support

Rustc is the compiler for the programming language Rust. It supports various platforms like Linux, Windows, Mac and as well to some extend RISC-V. The platforms also called targets are seperated in tiers that guarentee different aspects. The target names follow the LLVM target triple naming convention defined in llvm::Triple class.

The syntax is: `<arch><sub_arch>-<vendor>-<sys>-<env>`

For example: `riscv64gc-unknown-linux-gnu` would translate to:

- **riscv64gc:** 64-bit RISC-V architecture with the G (I+M+A+F+D) + C (compressed) instruction-set extensions
- **unknown:** vendor is unspecified (could be any)
- **linux:** the target operating system is Linux
- **gnu:** indicates the GNU C library (glibc) environment and toolchain conventions for Linux

Each target is put into a tier. There are 3 tiers:

- **Tier 1 ("guaranteed to work")**: This means the Rust project runs automated tests (including all applicable unit tests and integration tests) on these platforms in continuous integration (CI). The compiler and standard library are expected to build successfully, pass tests, and function reliably. Pre-built binaries are officially provided for Tier 1 targets, and regressions are treated as high-priority issues.
- **Tier 2 ("guaranteed to build")**: The compiler and standard library should build successfully for these targets, and the Rust project will accept bug reports. However, not all tests may be run in CI,

or they may not all pass. Official binaries may or may not be provided. While Tier 2 targets receive some level of support and oversight, they do not receive as rigorous a testing process as Tier 1.
- **Tier 3 (“may work”):** These targets are minimally supported. They are generally community-maintained, and there is no official guarantee they will build or pass tests. The Rust project does not run CI for these platforms, and pre-built binaries are not typically provided.

2.3. RISC-V targets

RISC-V embedded targets can be categorized into 32-bit and 64-bit architectures:

- 32-bit targets:
  - riscv32{e, em, emc}-unknown-none-elf ⟶ Optimized for ultra-constrained devices (16 GPRs, minimal power/area).
  - riscv32{i, im, ima, imc, imac, imafc}-unknown-none-elf ⟶ Higher performance with 32 GPRs and optional extensions for multiplication (M), atomic operations (A), and floating-point (F).

- 64-bit targets:
  - riscv64gc ⟶ Full-featured (integer, floating-point, atomic, compressed).
  - riscv64imac ⟶ Similar but omits floating-point (F, D), balancing performance and hardware constraints.

Tier Support:
- Tier 3: riscv32{e, em, emc} and riscv32ima.
- Tier 2: Most riscv32{i, im, imc, imac, imafc} targets and both 64-bit targets.
- Note: All are no_std (no standard library support).

2.4. The Rust Allocator

In `no_std` environments, such as embedded systems, Rust’s standard memory allocator is not available. This makes it necessary to define a custom allocator or use an external allocator provided by the operating system. Without a proper allocator, heap allocation features like `Box`, `Vec`, and `String` cannot be used.

Rust provides the `#[global_allocator]` attribute to specify a custom global memory allocator. This is useful in environments where the default allocator is unavailable or needs to be replaced.

The `#[global_allocator]` attribute sets a static instance of a type implementing `GlobalAlloc` as the global allocator. For no_std environments, a simple custom allocator can be implemented as follows:

```rust
use core::alloc::{GlobalAlloc, Layout};

struct SimpleAllocator;

unsafe impl GlobalAlloc for SimpleAllocator {
    unsafe fn alloc(&self, layout: Layout) -> *mut u8 {
        // Allocator logic will go here
    }

    unsafe fn dealloc(&self, _ptr: *mut u8, _layout: Layout) {
        // Deallocator logic will go here
    }
}

#[global_allocator]
static ALLOCATOR: SimpleAllocator = SimpleAllocator;
```

This basic example demonstrates how a custom allocator could be structured. In real scenarios, the `alloc` function would point to a memory pool, and `dealloc` would properly release memory.

It is also possible to use `extern functions` to import functions in the linking process. This is done by declaring them with the `extern` keyword:

```
extern "C" {
    fn malloc(size: usize) -> *mut u8;
    fn free(ptr: *mut u8);
}
```

3. QEMU

QEMU is an open-source machine emulator and virtualizer that can simulate a wide range of CPU architectures, including RISC-V. By running your program on QEMU, you can develop and test embedded applications without needing the actual hardware. For a RISC-V 32-bit setup (e.g., riscv32i), you typically invoke QEMU with a command like:

```
qemu-system-riscv32 \
    -nographic \
    -machine virt \
    -kernel path/to/your_program.elf
```

Here, -machine virt selects the generic RISC-V "virt" machine, while -nographic routes I/O through the terminal rather than a graphical console. You can then observe your program's output and interact with it as though it were running on physical hardware. This approach is especially helpful during early development, debugging, and continuous integration on embedded targets.

4. Bare Metal

!DISCLAIMER: Code is from the popovicu/risc-v-bare-metal-rust-dynamic-memory repository see Section 8!

4.1. Build

Here is a simple `Hello World` program in Rust that can be build for RISC-V and then be run by QEMU.

```
#![no_std] // 1
#![no_main] // 2

use core::arch::global_asm;
use core::panic::PanicInfo;
use core::ptr;

global_asm!(include_str!("entry.s")); // 3

const UART: *mut u8 = 0x10000000 as *mut u8; //4
fn uart_print(message: &str) {
    for c in message.chars() {
        unsafe {
            ptr::write_volatile(UART, c as u8);
        }
    }
}

// 5
#[no_mangle]
```

```
pub extern "C" fn main() -> ! {
    loop {
        uart_print("Hello, world!\n");
        for _ in 0..5000000 {}
    }
}
// 6
#[panic_handler]
fn panic(_info: &PanicInfo) -> ! {
    uart_print("Something went wrong.");
    loop {}
}
```

There are 6 key points that are important.

1. `#![no_std]`

This attribute tells the Rust compiler not to use the standard library (std). In embedded and bare-metal environments, you typically don't have an operating system or the usual platform features that std depends on, so you opt for no_std instead.

2. `#![no_main]`

Rust normally expects a main() function as the entry point, but in a bare-metal environment, you have full control over what "entry" means (often defined by a custom linker script and/or assembly). Marking the crate with `#![no_main]` ensures that Rust does not generate the usual runtime setup and main() boilerplate.

3. `global_asm!(include_str!("entry.s"));`

This embeds the assembly code from `entry.s` at compile time. The `entry.s` file contains critical startup instructions for our RISC-V system:

```
.global _start
.extern _STACK_PTR

.section .text.boot

_start:
  la sp, _STACK_PTR  # Load stack pointer
  jal main           # Jump to Rust's main function
  j .                # Infinite loop (if main ever returns)
```

- `_start` is the is the entry point, setting the stack pointer (`sp`) before jumping to `main`.
- `_STACK_PTR` is defined later in the linker script (see below).

4. `const UART: *mut u8 = 0x10000000 as *mut u8;`

This line defines a memory-mapped I/O address for a UART (Universal Asynchronous Receiver/Transmitter). In many embedded systems, hardware registers for peripherals like UART are accessed at specific memory addresses. By casting `0x10000000` to `*mut u8`, you can write bytes directly to that address to send characters.

5. `#[no_mangle] pub extern "C" fn main() -> ! { ... }`

`#[no_mangle]`: Tells the compiler not to rename (mangle) the function symbol, so the linker sees it as main.

`extern "C"`: Ensures the function uses the C calling convention, which is typically expected in low-level contexts.

`-> !`: Indicates the function never returns (an infinite loop). In this function, the code repeatedly prints "Hello, world!" via the UART and does a busy-wait delay.

6. `#[panic_handler] fn panic(_info: &PanicInfo) -> ! { ... }`

When a Rust panic occurs in a no_std context, you need a custom panic handler because there is no standard library to handle panics. This function logs an error message ("Something went wrong.") to the UART and enters an infinite loop, preventing any further execution.

In a bare-metal environment, there is no operating system to manage memory and load executables. Instead, we use a linker script to define how the program is placed in memory, ensuring that everything is correctly positioned for execution. Below is the linker script used for this RISC-V program, followed by an explanation of its purpose.

```
MEMORY {
  program (rwx) : ORIGIN = 0x80000000, LENGTH = 2 * 1024 * 1024
}

SECTIONS {
  .text.boot : {
    *(.text.boot)
  } > program

  .text : {
    *(.text)
  } > program

  .data : {
    *(.data)
  } > program

  .rodata : {
    *(.rodata)
  } > program

  .bss : {
    *(.bss)
  } > program

  . = ALIGN(8);
  . = . + 4096;
  _STACK_PTR = .;
}
```

The linker script defines the memory layout of the program, specifying where different sections of code and data should be placed in RAM. It assigns the `text` (code), `data`, read-only data(`rodata`), and `BSS` (uninitialized data) sections to the memory region starting at `0x80000000`. Additionally, it aligns memory properly and defines `_STACK_PTR`, which sets up the stack pointer for the program. This ensures that the compiled binary is correctly structured for execution on the target RISC-V hardware.

There are several ways to tell cargo to use the link script but one simple way is to add a build script `build.rs` in the root of the project next to the link script.

The `build.rs` looks like this:

```
fn main() {
    println!("cargo:rustc-link-arg-bin=<rust-project-name>=-Tlink_script.ld");
}
```

The build process is started with the command:

```
cargo build --target <target>
```

4.2. Run in QEMU

To run the program in QEMU choose the correct risc-v architecture and run

```
# for 32-Bit architectures
qemu-system-riscv32 -machine virt -bios <path/to/executable> -nographic
# or for 64-Bit architectures
qemu-system-riscv64 -machine virt -bios <path/to/executable> -nographic
```

5. Building Rust for RISC-V RTEMS

!DISCLAIMER: Code is from the RTEMS Documentation see Section 8!

5.1. RTEMS

RTEMS (Real-Time Executive for Multiprocessor Systems) is a free, open-source real-time operating system (RTOS) designed for embedded systems. It provides a POSIX-compliant API, supports multiprocessing, and is used in aerospace, automotive, industrial, and military applications. RTEMS is lightweight, highly configurable, and optimized for deterministic real-time performance, making it ideal for mission-critical systems.

The RTEMS Documentation (see Section 8) has a guide to Build Bare Metal Rust for RTEMS. It explains the general way on how to build Rust for RTEMS and then shows how to execute it in QEMU.

What is needed:

- RTEMS Tools to build BSP and the linking process.
- Configuration file `init.c` for RTEMS.
- A Board Support Package.
- A prepared Rust Project for RTEMS.

5.2. Board Support Package (BSP)

A Board Support Package (BSP) in RTEMS is a collection of low-level software components that enable RTEMS to run on a specific hardware platform. It includes CPU initialization, clock setup, interrupt handling, memory management, and device drivers. The BSP abstracts hardware differences, making RTEMS portable across multiple architectures.

RTEMS supports RISC-V architecture with BSPs for platforms such as QEMU RISC-V emulation. These BSPs provide essential support for booting, handling interrupts, and managing peripherals, allowing RTEMS applications to run on real and emulated RISC-V hardware.

A BSP can be configured for more specific cases. The configuration is done when building the specific BSP via a a `ini` file. Configuration options can be listed by running the command line tool `waf` from the RTEMS repository with the `bspdefaults` subcommand. Here is a snippet of the command output for the target `riscv/rv32i`:

```
λ ./waf bspdefaults --rtems-bsp=riscv/rv32i
# boot hartid (processor number) of risc-v cpu (default 0)
RISCV_BOOT_HARTID = 0
# Defines the build label returned by rtems_get_build_label().
```

```
RTEMS_BUILD_LABEL = DEFAULT
# Default size in CAN frames of FIFO queues.
RTEMS_CAN_FIFO_SIZE = 64
# Number of available priorities for CAN priority queues.
RTEMS_CAN_QUEUE_PRIO_NR = 3
# Enable the RTEMS internal debug support
RTEMS_DEBUG = False
# Enable the Driver Manager startup
RTEMS_DRVMGR_STARTUP = False
# Enable the Newlib C library support
RTEMS_NEWLIB = True
# Enable the para-virtualization support
RTEMS_PARAVIRT = False
```

5.3. Rust Project for RISC-V RTEMS

The setup for a Rust project in this environment is similar to the Bare Metal from Section 4. Except there is now no linker script and no start up assmeby file needed.

RTEMS provides an extensive library that includes allocator functions, which can be linked to Rust to enable dynamic memory management. Additionally, the library offers a `printk()` function for console output. Similar to the pure bare-metal approach, Rust requires a panic handler in this environment, which can be forwarded to the RTEMS-provided `rtems_panic()` function. Beyond these, the RTEMS library offers a wide range of additional system functions, including task management, synchronization primitives, and real-time scheduling capabilities, further supporting embedded system development.

The entry point to the program is a function that has to be implemented as

```
#[unsafe(no_mangle)]
extern "C" fn Init() {...}
```

Before exiting the program the RTEMS `rtems_shutdown_executive()` function is called to shutdown the RTMES system in a controlled manner.

To specify some build flags a `.cargo/config.toml` is defined where the configuration for cargo is defined. In it targets can be defined and configured. Here is a example to build for a `riscv64gc-unknown-none-elf` target:

```
[target.riscv64gc-unknown-none-elf]
# Either kind should work as a linker
linker = "riscv-rtems7-gcc"
# linker = "riscv-rtems7-clang"
rustflags = [
    # See `rustc --target=riscv64gc-unknown-none-elf  --print target-cpus`
    "-Ctarget-cpu=generic-rv64",
    # The linker is a gcc compatible C Compiler
    "-Clinker-flavor=gcc",
    # Pass these options to the linker
    "-Clink-arg=-march=rv64imafdc",
    "-Clink-arg=-mabi=lp64d",
    "-Clink-arg=-mcmodel=medany",
    # Rust needs libatomic.a to satisfy Rust's compiler-builtin library
    "-Clink-arg=-latomic",
]
runner = "qemu-system-riscv64 -M virt -nographic -bios"
```

The project can then be build with `cargo build`. The target of the BSP and the rustc target have to be same to assure compatibility. Rust lets you define more than one target in the `.cargo/config.toml`. `cargo build` will then create binaries for each target. To only build for one specific target cargo has the `--target` flag.

5.4. Linking and running in QEMU

Linking is done by compiling the RTEMS init.c and then using the RTEMS toolchain to link it with the Rust static library. In this process linker flags that are defined by the used BSP are used.

This produces a `<project-name>.exe` that then can be run in QEMU with:

```
rtems-run --rtems-bsp=<bsp_name> <project_name>.exe
```

6. Milestone Plan

1. Technical Onboarding (2 Weeks)
   - Familiarization with QEMU, RTEMS, and RISC-V
   - Understanding toolchain configurations, BSP setup, and target hardware requirements
2. Application on Hardware (4 Weeks)
   - Running a basic "Hello World" program on the hardware
   - Implementing a register interface via the FPGA-APB bus
3. Performance Comparisons (1 Week)
   - Comparing assembly code in C and Rust
   - Evaluating a minimal functional program in both C and Rust (memory usage & performance analysis)
4. Abstraction Layer Development (2 Weeks)
   - Translating C code into a Rust-based abstraction layer
   - Abstracting RTEMS functionalities (multi-tasking, timers, etc.)
   - Writing documentation for the abstraction layer
5. Rust Coding Guidelines (1 Week)
   - Documenting best practices for Rust vs. C in terms of quality and performance (building on findings from Step 3)
6. (Potential Future Step: Rewriting application software modules in Rust)

7. Conclusion

This document lays the foundation for implementing Rust on RISC-V in a bare-metal and RTEMS environment, providing insights into toolchain setup, memory management, and simulation via QEMU. While essential groundwork has been completed, the next step involves translating these findings into actual hardware implementation on a RISC-V instantiated FPGA. This transition will allow for deeper evaluation of performance, security, and real-world applicability, forming the basis for the bachelor thesis on Rust for FPBGA-SoC with RISC-V.

8. Sources

- The rustc book
- 16.1. Bare Metal Rust with RTEMS — RTEMS User Manual 7.e8e6f12 (8th March 2025) documentation
- Bare Metal Rust on RISC-V With Dynamic Memory. (Simplified to a plain Hello-World Program.)
- Source files for this project