QEMU emulation detection in bare metal Arm Cortex-M3 firmware

Author: Amir Sabbag | **Date:** 18 October 2026

1. Introduction

The basic goal of the assignment was to create a bare-metal (no standart library) firmware for the Arm Cortex-M3, which would be emulated in QEMU, with the goal of detecting a running emulator. Cortex-M3 is a 32-bit reduced instruction set chip made by Arm.

I chose the Rust language and the Cargo ecosystem for writing the firmware, since the set up was really similar to another project that I am currently working on. Nevertheless, the firmware is #![no_std] and #![no_main], and I did not use any crates that provide external features either, all of the code is self contained.

2. Project Layout

To compile the project, you will need a working rustup installation. The project pins the compiler version, namely nightly-2026-10-16, so it should be reproducable. We use the nightly compiler, since the binary dependencies feature of Cargo is currently unstable, although it is very useful for our case.

The project uses the Cargo workspaces feature, so the code in src/ is simply a QEMU runner, and the actual firmware code is in firmware/src/. When building the project, Cargo first builds the firmware (of course compiling to target thumbv7m-none-eabi) and creates the ELF, and passes its path along to src/main.rs.

In src/main.rs, we simply call the QEMU command with the specified arguments and the ELF source. A Makefile is provided as well, which will compile and run the firmware in QEMU.

3. Boot Sequence

The Arm Cortex-M3 chip constantly runs in a loop, and when it recieves a reset, it will load the stack pointer from 0x00000000, and the reset handler from 0x00000004 (we are in 32-bit after all). So, we will have to put the stack and the reset handler function in those respective places. We use a linker script for that.

The linker script is in the root of the project, at linker.ld. To know exactly what to put in the linker script, I had to go around and find the specific memory layout of the Arm Cortex-M3 chip. A nice script was in the chemuRTOS source code, which I have linked below. In the linker script, we use two memory regions, mainly the 4 MiB ZBT SSRAM1 starting at 0x00000000 (denoted CODE), and ZBT SSRAM2/3 starting at 0x20000000 (denoted RAM).

The CODE segment is denoted read and execute, and RAM is read, write and execute. We start our stack at the very limit of the RAM region, at 0x20400000, since the stack grows towards lesser memory regions. In our rust code, we have a _start() function, which is the default entry that linkers search for. That will be our entrypoint. We store the function pointer of _start() in a static variable, and place it in the section “vector_table.reset_vector”, which the linker script places right after the stack pointer.

We register the eight exception handlers via the static variable EXCEPTIONS, which is placed in “vector_table.exceptions”. For now, all of them point to a default handler dh(), which loops forever. Not much “handling” there. After that, we initialize `.text` and `.rodata` in CODE, since those are read-only, and `.bss` and `.data` in RAM. We also provide the start and end pointers of `.bss` and `.data`, which we will need later. That wraps it up for the linker script!

Our _start() function will do the necessary setup and then call main. According to the Embedonomicon (linked below), it is not advised to write the start-up code in Rust, yet adding assembly in the middle was going to complicate everything too much, so I just did what they told me not to do (it seems to work so far!). Simply, with the section pointers that we get from the linker script (I had written about this last year on my website, actually), we zero out the `.bss`, and copy `.data` from CODE to RAM. That is about it, after that, we can finally call our Rust function main(). So far it was just dealing with extern “C” fn’s.

Finally, in firmware/src/uart.rs, with the init() function, we can initialize the UART interface. We only use the UART0, and the source file pretty much explains how to set it up.

4. Detection Checks

Both of the detection checks implemented are in the source file firmware/src/em_check.rs. They both follow from the same logic, mainly finding gaps in between QEMU’s implementation, and ARM’s specification. The way this search was done was straightforward:

1. Find the source for the QEMU board for Cortex-M3, at https://github.com/qemu/qemu/blob/master/hw/arm/mps2.c
2. Find all the calls of create_unimplemented_device(), which creates and maps a dummy device.
3. Find address regions where there are unimplemented devices.
4. Refer to the ARM documentation on whether there are reset value registers in that range, or other R/W registers.

Step 4 was done via ChatGPT, where I asked the LLM to search through documentation and find me possible candidates.

4.1. CMSDK AHB GPIO 0 PIDN and CIDN reset value

For our first check, we used the following information from the documentation: CMSDK AHB GPIO 0 lays between start address 0x40010000, and end address 0x40010FFF (Application Note AN385). It contains the registers PID0, PID1, PID2, and CID0, CID1, CID2, CID3, which according to the documentation, all advertise reset values (Arm Cortex-M3 System Design Kit).

Reg.	Offset	Expected	QEMU
PID0	0xFE0	0x20	0x00
PID1	0xFE4	0xB8	0x00
PID2	0xFE8	0x1B	0x00
CID0	0xFF0	0x0D	0x00
CID1	0xFF4	0xF0	0x00
CID2	0xFF8	0x05	0x00
CID3	0xFFC	0xB1	0x00

Reset values are values that the registers have after booting, and since these registers are read-only, reading them should provide the values advertised in the documentation. However, QEMU maps those regions to a placeholder, and simply returns 0 for everything.

The firmware reads each register with a volatile access and compares its low byte with the documented value. Under QEMU 11.1.2, every listed register returned `0x00`. I did not test the firmware on a physical AN385 board.

An attacker could bypass this check by modifying the emulator to return the expected ID bytes at these addresses.

4.2. APB test slave register

For the second check, we used the APB test slave register, which lays between the start address 0x4000B000, and end address 0x4000BFFF (Application Note AN385). The CMSDK documentation defines a 32-bit read/write data register at offset 0x000. The firmware saves the register's initial value, writes 0xDEADBEEF using a volatile 32-bit access, reads it back, and then restores the initial value.

On a physical board, one would expect the read to result in 0xDEADBEEF. When testing this on QEMU emulator version 11.1.2, we get back 0, which is inline with what we saw from the source code. An attacker could bypass this by implementing this functionality on their emulator.

5. Bypasses

With respect to this category of checks, I devised three ways to bypass these checks. They are listed in increasing order of difficulty for the attacker.

5.1. State altering via debugging

If the attacker can get the firmware to reliably run in a debugger, the attacker has access over all the defined variables, registers, and the state of the machine. An attacker could alter the stack, memory locations, variables, skip over certain instructions. This kind of control allows the checks to be bypassed easily.

5.2. Firmware patching

If the checks are in a location that is obvious to the attacker, the attacker can use decompilation and reverse engineering tools like Ghidra to patch the firmware bytes directly, either bypassing the detection branches or overwriting expected values. Since these tools are extremely powerful nowadays, with the assistance of LLMs these modifications are made trivial.

5.3. Extending the emulator

These checks depend on the fact that there are gaps between the emulator environment and bare metal hardware. If the attacker can deduce the nature of the checks, and extend the emulation environment, the checks can be bypassed. Again, with the assistance of LLMs extending these interfaces are now much easier than before.

6. Conclusion

I was quite surprised by the difficulty of having a reliable way of checking whether one is running in an emulator, and I would have expected there to be some very easy way to do so. I am guessing, with the addition of hardware devices, the surface area emulators have to cover to be undetected rises sharply, so it would be quite hard to have an emulator without any gaps. My other hopes lay in hardware execution time checking, which would check execution time and compare it to an expected value.

7. Links

- https://github.com/qemu/qemu/blob/master/hw/arm/mps2.c
- https://github.com/calebhultmann/chemuRTOS/blob/main/board/mps2-an385/linker.ld
- https://docs.rust-embedded.org/embedonomicon/memory-layout.html
- https://www.qemu.org/docs/master/system/arm/mps2.html
- Application Note AN385 ARM Cortex-M3 SMM on V2M-MPS2
- Arm Cortex-M3 System Design Kit