


[< Go back](#)

## Bare metal programming with RISC-V guide

 September 9, 2023 | 12:05 PM

Follow @popovicu94

Today we're going to explore how to write a bare metal program for a RISC-V machine. For reproducibility, the target is a QEMU `riscv64 virt` machine.

We will briefly cover the initial stages of the RISC-V machine bootup and where you can plug in your custom software to program the bare metal machine!

At the end of this article, we will write a bare metal program for our RISC-V machine and send a string 'hello' to the user, without depending on **any** supporting software on the running machine whatsoever (OS kernel, libraries, anything).

## Table of contents

## ▼ Open Table of contents

- Machine bootup and running the initial software
  - General concepts
  - QEMU bootup
    - The Zero Stage Bootloader (ZSBL)
    - QEMU -bios flag
    - A note on the -kernel flag
    - How can ELF files be used during the bootup??
- Writing a custom "BIOS" for RISC-V
  - Interacting with the user via UART
  - Coding it all together!
  - Running the "fake BIOS" on QEMU
- I just want to run the code!

## Machine bootup and running the initial software

### *General concepts*

*Feel free to skip the section on general concepts if you are familiar with how computers boot*

When a real machine is powered on, the hardware first runs the health checks and then loads the first instructions to run into its memory. Once the instructions are loaded, the processor core initializes its registers and the program counter points to the first instruction. From that point on, the software can run.

In simpler setups like small microcontrollers, this is all the software there is, just a single binary blob of instructions. The processor will execute just that going forward. In a more complex setup like a laptop or a phone, there are more stages to the startup.

In those more complex setups, traditionally, the first instructions are the BIOS, whose task is to subsequently load the bootloader into the memory and hand-off the control to it. The bootloader is usually small and easy to load into the running memory and the processor can easily start running its code. It proceeds to load the operating system kernel into the memory (implementing the bootloader though is a science of its own).

Each machine loads the initial software in its own way. For example, the BIOS can be stored on a separate storage chip and upon powerup, the contents of the storage are simply filled into the memory at a fixed address and the processor just executes starting from that address.

## ***QEMU bootup***

`riscv64 virt` machine, even though its virtualized, still has its own boot sequence. It goes through multiple stages, and at the moment, we will not be exploring them all. Please stay tuned for the follow-up articles with those details.

The key to understanding this virtual machine is that, obviously, it has no chip attached to it from which to read the software (it is virtual) so QEMU simulates this in some way. You might have seen the flag `-bios` for QEMU examples before and hopefully now you have a strong intuition what it could be. If you're guessing this is passing the very first instructions that your virtual RISC-V core is executing upon the startup, you are ***almost correct***.

### **The Zero Stage Bootloader (ZSBL)**

Once you power on this virtual machine, QEMU fills the memory at `0x1000`

with a few instructions and sets the program counter right to that address. This is the equivalent of a real machine having some hardcoded ROM firmware on the board (tucked away in some chip) and just dumping the contents into the RAM upon the bootup. You **do not have the control** over these instructions, i.e. they are not a part of your software image, and generally, I do not see a reason why you would want to override those, and they are actually quite useful for more complicated setups (I promise we will cover them in a follow up article). For the curious ones, these few instructions are the **Zero Stage Bootloader (ZSBL)**. The ZSBL sets up a few registers for reasons we'll explore in the future (right now, you can basically ignore this register setup) and jumps to the address `0x80000000` which is where the action truly begins!

### **QEMU -bios flag**

`0x80000000` is where the first user-provided instructions to QEMU are running, and they are loaded there as soon as the virtual machine starts. If you don't pass anything, QEMU will use the default and load up a piece of software called **OpenSBI**. The next article in this blog will be exactly what is the concept behind SBI in RISC-V and what exactly OpenSBI is. It's important to note that SBI on RISC-V isn't **really** BIOS, but something very similar. My personal guess is that the QEMU authors simply recycled the flag that was available and representing BIOS on other architectures like `x86`. Anyway, something to keep in mind is that SBI is generally very similar to BIOS in terms of what it does, and more importantly, it is something you can customize.

The `-bios` flag is the **ELF** a binary file containing instructions and potentially some other data, organized in sections. **ELF** is the standard binary format for Linux, and the details of the **ELF** file format are way outside the scope of this article, but a sufficient mental model here is that it is simply a key-value map where key is the starting address of a section, and the value is the bunch of bytes that need to be loaded into the memory at that address. Therefore, the **ELF** file

provided to the `-bios` flag should fill out the memory starting at `0x80000000` (and this is indeed what QEMU's default OpenSBI image does).

### A note on the `-kernel` flag

If you have been booting an operating system with QEMU before (e.g. Linux), you have likely used the `-kernel` flag. It is basically the same thing as the `-bios` flag: you can pass it an ELF image which covers some other memory region, and conceptually it will just dump the bytes in the memory. We won't be using this flag today, we'll cover its usage in the following articles.

### How can ELF files be used during the bootup??

Even though conceptually ELF files represent just ways to fill in the memory, they are definitely not super simple that you can write a quick parser in one afternoon. A careful reader may wonder how does the machine then know to parse out the contents mapped to some address `0x12345678` from the ELF file and load the memory with those. This would be a great observation – in our case, we are using a virtual machine and we are basically simulating a machine which conceptually has such intelligent digital circuitry or amazingly complex initial software bootloader that is available in the machine's memory right upon the powerup. That is, of course, not what happens in the real machines. The software that is loaded upon the powerup is stored on the machine storage as a flat binary blob that is blindly just dumped into the memory upon the powerup, there is really no parsing involved, but since we're dealing with a **virtual** machine here, the sky is the limit, we are not bound by the complexities of manufacturing the hardware that does any of this.

## Writing a custom "BIOS" for RISC-V

We have established that `0x80000000` is the location of the first user-provided instruction that the machine executes. I provided it as just a

fact, and if you really want a little more background as to why this might be so, you can start from here. Basically, what we see here is that DRAM is mapped to start at the `0x80000000` in the address space (if you don't know what this means, don't worry, it will not be too relevant for the rest of this article).

Let's begin by building an `ELF` file that will lay some processor instructions at address `0x80000000` that will give the user a message 'hello'!

## *Interacting with the user via UART*

Those who have done embedded systems programming in the past are surely familiar with the concept of UART. UART is a very simple device used for the most basic form of input/output: there is one wire for input (receiving, known as `RX`) and one wire for output (transmit, known as `TX`), and one bit goes onto the wire at the time. If you're connecting two devices to speak to each other over UART, one device's `TX` is the other device's `RX`, and the other way around. If you're reading this article and have not done anything with UART before, I strongly suggest at least getting the cheapest possible Arduino and having it speak to your computer through a USB-to-UART cable. The concept would be identical to what we're doing here, but you would be doing it for real, and it will make more sense, since the scenario here is entirely virtualized.

QEMU virtualizes an UART device on the virtual machine, and our software can access it. When you open the QEMU's serial port (UART) section, what happens is basically when you press a keyboard button, the code for that button is sent out of your host's `TX` to the VM's `RX` and when the VM outputs something on its `TX`, it will be rendered to you graphically in the terminal (so you don't have to otherwise decode the electrical signals from the simulated board :)), e.g. if the VM sends out 8 bits representing `65`, your QEMU will render the character `a`, since that is its ASCII code.

We know that QEMU maps UART at the address `0x10000000` (you can check it in their source code) and the device that is virtualized here is `NS16550A`. The details do not matter here: for the purposes of the article, what this means is if you send an 8-bit value to that address from your software, that will be sent out on the `TX` wire of the virtualized UART device. Practically, that means if you go to QEMU's serial port, the character you wrote to `0x10000000` will be rendered in your console.

## *Coding it all together!*

With all this knowledge in mind, now we can write the code. The `ELF` file we are about to build will lay out a few instructions at `0x80000000` to print the characters, `'h'`, `'e'`, `'l'`, `'l'` and `'o'` in succession to the address `0x10000000`. Finally, the code should then get stuck in an infinite loop (so that QEMU doesn't crash for any strange reason and we can inspect the output)

```
.global _start
.section .text.bios

_start: addi a0, x0, 0x68
        li a1, 0x10000000
        sb a0, (a1) # 'h'

        addi a0, x0, 0x65
        sb a0, (a1) # 'e'

        addi a0, x0, 0x6C
        sb a0, (a1) # 'l'

        addi a0, x0, 0x6C
        sb a0, (a1) # 'l'

        addi a0, x0, 0x6F
        sb a0, (a1) # 'o'
```

```
loop:    j loop
```

You can save this file as `hello.s`. Let's assemble this file into the machine code. In my case (and likely in yours), I am using a cross-platform toolchain, meaning that I am developing on a platform different from the target platform. Concretely, I am developing this software on an `x86` machine and building for a `riscv64` machine.

To assemble this file, I run the following:

```
riscv64-linux-gnu-as -march=rv64i -mabi=lp64 -o hello.o -c hello.s
```

The exact command may be different depending on what kind of assembler you have for `riscv64`, this is the tool I have obtained through my system's package manager for Debian. I leave it to the reader to obtain the correct toolchain for building `riscv64` software, it should generally be a matter of just obtaining the right software package from the Internet.

Now, the code is only assembled, meaning that we have the software instructions in the machine code format, but this binary is still not ready to go and act as our fake BIOS. We need to use a **linker** and drive its behavior with a **linker** script to ensure that the instructions we have generated will be laid out at `0x80000000` as we intended. Let's write the linker script.



```

MEMORY {
    dram_space (rwx) : ORIGIN = 0x80000000, LENGTH = 128
}

SECTIONS {
    .text : {
        hello.o(.text.bios)
    } > dram_space
}

```

We won't be covering what all this means, but tl;dr is we now have a way to put those instructions exactly where we want them. Let's verify with **objdump**.

```
riscv64-linux-gnu-objdump -D hello
```

Disassembly of section .text:

```

0000000080000000 <.text>:
    80000000: 06800513          li  a0,104
    80000004: 100005b7          lui a1,0x10000
    80000008: 00a58023          sb  a0,0(a1) # 0x10000000
    8000000c: 06500513          li  a0,101
    80000010: 00a58023          sb  a0,0(a1)
    80000014: 06c00513          li  a0,108
    80000018: 00a58023          sb  a0,0(a1)
    8000001c: 06c00513          li  a0,108
    80000020: 00a58023          sb  a0,0(a1)
    80000024: 06f00513          li  a0,111
    80000028: 00a58023          sb  a0,0(a1)
    8000002c: 0000006f          j   0x8000002c

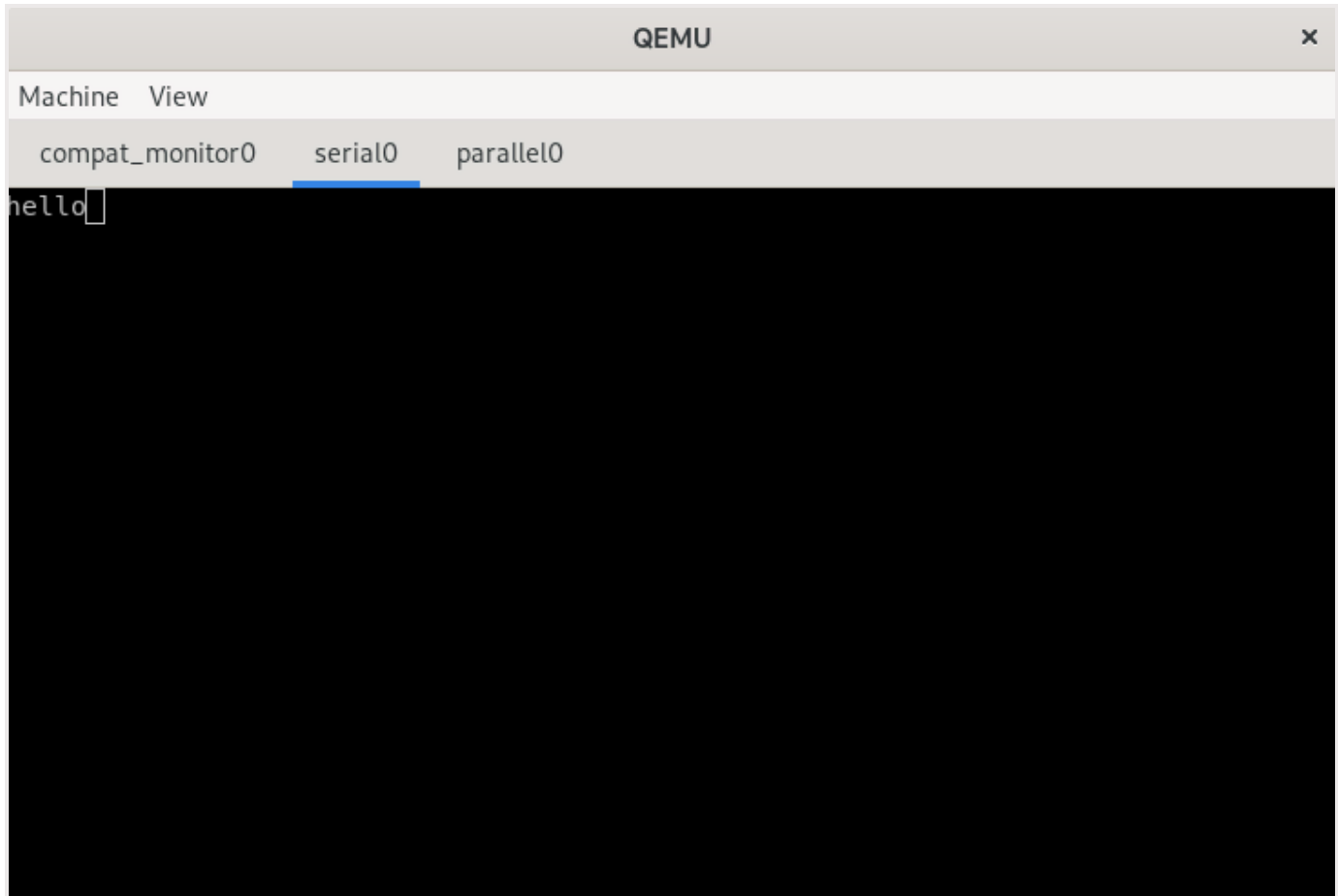
```

## Running the “fake BIOS” on QEMU

QEMU can now be fired up by running the following command:

```
qemu-system-riscv64 -machine virt -bios hello
```

To see what happens on UART, click the **View** button in the top menu and switch to the serial port view. The output should be like this:



## I just want to run the code!

Head over to the [Github repo](#) for this article, run the `make` command and that will do everything we described above. You can then launch QEMU.

[#risc-v](#) [#bare-metal](#)

Copyright © 2025 | All rights reserved.

