


[< Go back](#)

RISC-V SBI and the full boot process

 September 9, 2023 | 09:00 PM

Follow @popovicu94

In the last article, we covered bare metal programming on RISC-V. Please familiarize yourself with that material before proceeding with the rest of this article, as this article is a direct continuation of the aforementioned one.

This time we are talking about RISC-V **SBI (Supervisor Binary Interface)**, with **OpenSBI** as the example. We'll look at how SBI can assist us with implementing operating system kernel primitives and we'll end the article with a practical example using **riscv64 virt** machine.

Table of contents

▼ Open Table of contents

- RISC-V and “BIOS”
 - Machine modes
 - SBI
 - Fancy abstractions
 - Binary interface
- Practical example with OpenSBI
- Bootng the OS kernel after SBI and calling into OpenSBI
 - What really happens in the ZSBL?
 - 3 flavors of OpenSBI
 - FW_PAYLOAD
 - FW_JUMP
 - FW_DYNAMIC
 - Exploring the fw_dynamic_info struct
 - Building an “infinite-loop fake kernel”
 - Intentionally skipped details
- Hello world fake kernel
- Conclusion
- Code pointers

RISC-V and “BIOS”

In the article mentioned above, we talked extensively about the very

first stages of the RISC-V bootup process. We mentioned that first the ZSBL (Zero Stage Bootloader) runs, initializes a few registers and jumps directly to some address hardcoded by ZSBL. In the case of QEMU's `riscv64 virt`, the hardcoded address is `0x80000000`. This is where the first user-provided code runs, and if left to default, QEMU will load `OpenSBI` there.

Machine modes

So far we have avoided talking about different machine modes, and now is the perfect time to introduce them. The concept with machines modes is that not every piece of software should be able to access just about any memory address on the machine, or even execute just about any instructions available with the CPU. Traditionally, in a textbook example, the two main divisions are made here:

1. Privileged mode
2. Unprivileged mode

The *privileged mode* is where the machine starts at the boot time. Any instruction is permitted and no address access is considered an access violation. Once the operating system takes over the control of the system and starts launching the user code (aka userspace code), the modes start switching. When the user code is running on the CPU core, it is running within the *unprivileged mode* where not everything is accessible. Going back to the kernel mode means switching back to the privileged mode.

This is a very textbook and simplistic view at the permissions of operations and the question arises: why only 2 modes?

In systems, more than 2 modes typically exist, forming a protection ring with multiple access modes. RISC-V specification does not necessarily prescribe exactly which modes must be implemented for a core, except the **M (Machine)** mode. This is the most privileged mode.

Typically, the processors with M mode only are simple embedded systems, moving over more secure systems (M and S modes), all the way to full systems that can run Unix-like operating systems (M, S and U modes).

SBI

The official docs provide a formal definition, and I will try to water it down here with the goals of making it more intuitive.

RISC-V's SBI spec defines the layer of software that sits at the bottom of the RISC-V software stack. This is very similar to BIOS, which is traditionally the first bit of software that runs on a machine. You might have seen some of the guides for developing a simple kernel from scratch, and they typically involve something similar to what we did in the initial guide for bare metal programming on RISC-V, with a small twist – they are very often actually depending on the pre-existing software to do some I/O. The similarity to our previous guide is that they also carefully align the first instructions to the correct address to ensure that the processor's execution flow goes as intended and the simple kernel takes over at the right time, however, what I have typically observed in those short guides is that the goal is typically to print something like 'Hello world' to **the VGA screen**. This last bit sounds like a fairly complex operation, and it really is.

How is printing to the VGA then done easily then? The answer is that BIOS is here to assist with the most basic I/O operations such as printing some characters to the screen, hence its name – **Basic Input Output System**! Please pay attention to the opening section of the bare metal programming guide: we were achieving interaction with the user *without* depending on *any* existing software on the machine (well, almost true, we still went through the Zero Stage Bootloader, but we didn't depend on any outcome from it, nor we really had any control over it; it's simply hardcoded into the system). If we were to print something on the VGA screen, instead of sending characters out through UART, we would have to do a lot more than send an ASCII code to a single

address. VGA involves setting up the display device into the right mode, by sending multiple values over, setting up different parameters, etc. It's a fairly elaborate operation.

So how does BIOS traditionally help with tasks like these? The main concept is that whatever operating system ends up installed on the machine, it would anyway need some basic functionality, such as printing some information to the VGA screen. Thus, the machine can have these standard operations simply baked into it and ready to consume by whatever operating system ends up on the machine. Conceptually, we can think of these procedures as an everyday library we write our applications against.

Additionally, if an operating system is written against such a "library", it automatically becomes more portable. The "library" should have all the low level details, such as "outputting to UART means writing to `0x10000000`" (as is the case with QEMU's `riscv64 virt` VM), vs. "outputting to UART means writing to `0x12345678`", and the operating system simply needs to invoke "outputting to UART" procedure, while this "library" will know exactly how to interact with the hardware.

Fancy abstractions

This is all just a lot of talk for a very simple concept we have been using in programming since day 1: we apply **layers of abstractions** in our coding. Think of something like a Python function that does something like "sending a local file to an email address". From a high level perspective, we simply call a function `send_file_to_email(file, email)` and the underlying library opens up the network connection and starts pumping the bytes. This could be just another Python library. At some point, that will likely move down the software stack, and the Python library will depend on the Python runtime written in something like C to make a system call to the operating system (for example, to perform a core operation such as opening a network socket). The operating system has a network driver

somewhere deep down, which knows to which address in the address space does it need to send the individual bytes in order to send the bytes over the wire to the network and so on. The main concept here is that we have an established way of hiding the complexity of operations by delegating them to the lower layers of the software stack. We built the larger system not from the atomic parts, but out of “molecules”.

If we’re delegating the complexity to the underlying library, it probably just means a function call. However, once it’s time to delegate the complexity to the operating system and lower, this happens through a **binary interface**.

Binary interface

Since basically forever, the **x86** has been the dominant architecture for the computers we use, be it desktops or laptops. Things have been changing a lot lately, and other architectures are entering the picture, but let’s focus on just **x86**. What then, makes an application built for Linux incompatible with the application for Windows? If it’s written for **x86**, and both Linux and Windows run on **x86**, what could possibly be the differentiator here? The CPU instructions are not different from one platform and the other, so what could it be? The answer is the **interface between the application and the operating system**. This particular link between the user software and the operating system is called the **application binary interface (ABI)**. ABI is just a definition that says how the services from the operating system are invoked from the user application.

Therefore, when we say something like “this software is written for platform X”, it’s not enough to just say that X is **x86** or **RISC-V**, we must say **x86/Linux** or **x86/Windows** or **RISC-V Linux** etc. The platform definition may be even more complex than that if things like dynamic linking are involved, but let us not go there for now.

Let’s take a quick example at a program written in assembly for

x86/Linux that just prints a 'Hello' string to the standard output.

```
.global _start

.section .text

_start: mov $4, %eax ; 4 is the code for the 'write' system call
        mov $1, %ebx ; We are writing to file 1, i.e. the 'standard output'
        mov $message, %ecx ; The data we want to print is at the address defined
        mov $5, %edx ; The length of the data we want to print is 5
        int $0x80 ; Invoke the system call, i.e. ask kernel to print the data to

        mov $1, %eax ; 1 is the code for the 'exit' system call
        mov $0, %ebx ; 0 is the process return code
        int $0x80 ; Invoke the system call, i.e. ask the the kernel to close th

.section .data
message: .ascii "Hello"
```

Assemble this program with:

```
as -o syscall.o syscall.s
```

Link it with:

```
ld -o syscall syscall.o
```

Run with:

```
./syscall
```

You should see the output "Hello". If you're on Bash and you also want to double check the process return code, simply run:

```
echo $?
```

And you should see `0`.

Tip: If you want to try out this example from above, but you do not have access to an x86/Linux machine, you can do this through a JavaScript VM that emulates an x86 system in-browser [here](#); that's a really cool website!

And there we have it: a program which prints a message to the standard output when run on an x86 machine with a Linux kernel. C standard library **was not used**. The final **ELF** binary should run on Linux with no dependencies other than it is run on the correct platform.

Now back to the question, what makes this binary incompatible with Windows (potentially)? **Another operating system encodes the system calls differently (e.g. writing isn't code 4, but code 123, or the parameters are passed through different CPU registers)**. And now you have a good idea of how to directly interface with the kernel, without the assistance of the standard library (although you probably almost never want to do it). This means you have uncovered the layer at which software does things like opening files, allocates memory, sends signals, etc. The C standard library can be thought of as a wrapper which hides this complexity of invoking software interrupts through the **int** instruction to communicate with the kernel, and instead makes it look like a normal call to a C function, and then under the hood, this is what it is. To be fair, the library does a lot more than that, but for the purposes of this article, it can be thought of simply as a wrapper.

And now in the RISC-V world, we have the same thing: the user application interfaces with the kernel through software interrupt CPU instructions, and passing the parameters through the CPU registers. And the kernel basically does **the same thing** with the SBI in order to

invoke its services! It's just that this final layer of logic invocation is called the **SBI**, not the **ABI**. A way to think about it is that it is not the **application** that works in the lower layer, but rather the **supervisor** of the applications. The difference, however, is in the name only, and the concept remains absolutely the same.

Practical example with OpenSBI

At this point we have established that SBI, much like ABI, is just a way of invoking a functionality in the lower layers of the software stack. Furthermore, we also established the SBI sits at the bottom of the software stack on a RISC-V machine, and runs in the most privileged M mode. Let's add some more details to this picture.

It should also make sense at this point why the QEMU developers chose the `-bios` flag in order to accept the SBI software image (because the functionality is basically the same as BIOS). As a reminder, the `-bios` flag should point to an **ELF** file that will lay out the SBI software out in memory starting from address `0x80000000`.

Let's start the QEMU's VM with just OpenSBI loaded, and see what happens. We shouldn't really have to pass anything to QEMU since it defaults to loading OpenSBI at `0x80000000`.

```
qemu-system-riscv64 -machine virt
```

This is the output (on the serial port, not VGA):

```
OpenSBI v0.8
```

```

  ----
 /  __ \
| | | | _ __   ___ _ __ | (___ | |_) | | | | | | |
| | | | ' _ \ / _ \ ' _ \ \___ \| | | |
| |__| | |_) | |_) | |_) | |_) | | |

```

```

\____/| .__/ \___|_| |_|_____/|____/____|
| |
|_|

```

```

Platform Name      : riscv-virtio,qemu
Platform Features  : timer,mfdeleg
Platform HART Count : 1
Boot HART ID       : 0
Boot HART ISA      : rv64imafdcsu
BOOT HART Features : pmp,scounteren,mcounteren,time
BOOT HART PMP Count : 16
Firmware Base      : 0x80000000
Firmware Size      : 96 KB
Runtime SBI Version : 0.2

```

```

MIDELEG : 0x00000000000000222
MEDELEG : 0x0000000000000b109
PMP0     : 0x00000000080000000-0x0000000008001ffff (A)
PMP1     : 0x00000000000000000-0xffffffffffffffff (A,R,W,X)

```

The machine keeps spinning in place, presumably because it is set up to do so by default since there is no other piece of software passed to QEMU to take over the control after OpenSBI. At this point, things look good, it seems like OpenSBI has been set up properly (and its output confirms that it sits right at `0x80000000`).

How do we keep going up the software stack, how do we add a new layer? The new layer could be something like an operating system kernel, so similarly to how we have previously built an `ELF` file containing instructions to be placed at `0x80000000`, we will build another `ELF` file for QEMU to load into its memory, but this time the instructions will come to another address, since the portion starting at `0x80000000` has already been taken over by OpenSBI.

Which address should we load our fake “kernel” at, then?

Booting the OS kernel after SBI and calling into OpenSBI

When we loaded the BIOS/SBI/whatever you want to call it, the address was basically burnt into the machine's logic. The first few instructions were Zero Stage Bootloader (ZSBL) and the final instruction from there was jumping to the hardcoded address `0x80000000`. As we previously mentioned, this is an immutable fact of the platform we're working with, it's just simply what it does. However, that's all it really hardcodes at this point: it just hardcodes that you will have to start from `0x80000000`, and now we have OpenSBI placed there, so where does OpenSBI take us next?

Now enters the importance of the **ZSBL** again and now it really matters how it initializes those registers before performing that hardcoded jump to `0x80000000`. What ZSBL really does is two things:

1. Ensures that the software running **after** OpenSBI's initialization can run, and this is basically the OS kernel bootloader, or it could be the kernel itself directly (which is what you typically see in QEMU guides where you launch Linux, bootloader is skipped and the memory is immediately loaded with the kernel).
2. Jumps to the OpenSBI.

We have covered the second point in great detail so far, so let's now dig deeper into how does it accomplish point #1.

What really happens in the ZSBL?

We have mentioned before that ZSBL execution starts at the address `0x1000`. Let's trace the execution through QEMU and see what's going on. To do that, we'll add 2 flags to the QEMU CLI command: `-s` and `-S`. These flags ensure that QEMU exposes a **gdb** debug port, and additionally, the VM pauses immediately upon creation, waiting for us to drive it manually (which we will do through **gdb**).

Let's begin this reverse engineering process. We're starting QEMU as so:

```
qemu-system-riscv64 -machine virt -s -S
```

In another terminal, we connect to the `gdb` server nested in QEMU, so we can drive the VM forward. I am doing this on an `x86` machine, so I will use `gdb-multiarch` so I can do a cross-platform debug for `riscv`. So in this new terminal, I just run:

```
gdb-multiarch
```

I want to set up a few things before I connect into the VM to drive it forward:

```
set architecture riscv:rv64
```

It should be obvious what the line above does. Next, I want to get the actual running instruction printed to my terminal each time I move one instruction:

```
set disassemble-next-line on
```

It's time to connect to the QEMU `gdb` server (port `1234` is I believe hardcoded by QEMU, though it *may* be configurable by the `-s` flag somehow; I never tried it and I don't think you'll need to change this behavior)

```
target remote :1234
```

And right there, `gdb` is waiting for us at `0x1000`, exactly where the very first instruction after power on happens. We will use `si` a few

times to step through instructions one by one, until we get to the jump to SBI at `0x80000000`.

```
(gdb) target remote:1234
Remote debugging using :1234
warning: No executable has been specified and target does not support
determining executable automatically. Try using the "file" command.
0x0000000000000100 in ?? ()
=> 0x0000000000000100: 97 02 00 00 auipc    t0,0x0
(gdb) si
0x0000000000000104 in ?? ()
=> 0x0000000000000104: 13 86 82 02 addi    a2,t0,40
(gdb) si
0x0000000000000108 in ?? ()
=> 0x0000000000000108: 73 25 40 f1 csrr    a0,mhartid
(gdb) si
0x000000000000010c in ?? ()
=> 0x000000000000010c: 83 b5 02 02 ld     a1,32(t0)
(gdb) si
0x0000000000000110 in ?? ()
=> 0x0000000000000110: 83 b2 82 01 ld     t0,24(t0)
(gdb) si
0x0000000000000114 in ?? ()
=> 0x0000000000000114: 67 80 02 00 jr     t0
(gdb) si
0x0000000080000000 in ?? ()
=> 0x0000000080000000: 33 04 05 00 add    s0,a0,zero
```

There were only 6 instructions in ZSBL before handing the control over to the OpenSBI, including the jump itself. However, what are these few instructions that happened, what is their significance?

It turns out that all this is part of the SBI specification too, it's a part of the boot sequence. However, with OpenSBI, there are 3 different flavors of this dance, and let's look at those flavors first before getting into a lot of details on what happens after the ZSBL.

3 *flavors of OpenSBI*

You can build OpenSBI in 3 different ways:

1. `FW_PAYLOAD` (official docs)
2. `FW_JUMP` (official docs)
3. `FW_DYNAMIC` (official docs)

`FW_PAYLOAD`

This one is probably the easiest to understand conceptually. When building this flavor of OpenSBI, you will literally point the `make` tool to your kernel/"whatever you want to run after OpenSBI" image and you will get a single binary payload that you can directly load wherever your first CPU instructions start from (in QEMU's VM case, `0x80000000`). As I understand, it is possible to tweak the exact location of your software in relation to the OpenSBI blob in the memory, but for simplicity, the mental model we can apply here is that OpenSBI and your software blob are spliced together into a single blob and once the OpenSBI initialization finishes, the very next instruction is your software (you basically slide right into your software after OpenSBI).

The way to achieve this is:

1. Make sure `FW_PAYLOAD=y` is set in the `make` process, this will ensure a file called `fw_payload` is generated.
2. Point `FW_PAYLOAD_PATH` in your `make` process to the software you want to run after OpenSBI.

Per the docs linked above, if you skip the second flag, a very simple piece of software will be spliced with OpenSBI: a blank infinite loop. That explains why when we just launched QEMU with no flags, basically with OpenSBI only, the machine kept spinning in place – OpenSBI was likely built this way (since you can't just keep executing random contents of the memory) and it was just busy waiting in place.

The upside of this approach is that now you have a single, spliced, monolithic software image to load into your machine. You don't have to deal with multiple floating pieces, just one monolith. If your build process for the software is straightforward, you may even end up with a really easy way to manage all the software on the target machine, while getting all the upside of having OpenSBI do some work for you.

The downside is that you are now responsible for building everything together, including OpenSBI. What's worse, if the machine already had OpenSBI, let's imagine, burnt into some ROM, it already has OpenSBI to boot up, having it twice on a machine likely won't cut it.

FW_JUMP

This one is fairly simple too: you basically hardcode the address of your software that comes after OpenSBI. Similarly to above, 2 steps are needed.

1. Make sure `FW_JUMP=y` is set in the `make` process, this will ensure a file called `fw_jump` is generated.
2. Set `FW_JUMP_ADDR` in the `make` process to the address where OpenSBI should jump once its done.

This is quite similar to what we had in the previous scenario, only the jump address is hardcoded. It seems like in this case you are still necessarily responsible for building the OpenSBI image, but it's easy to rebuild it and point to different addresses for different machines (let's say different machines with varying memory layouts).

FW_DYNAMIC

This one is the most generalized flavor and that's why we leave it for last. This is where the importance of the register set up in ZSLB shines.

In this flavor, the boot stage that happens before OpenSBI is in charge

of passing a few pointers to OpenSBI. In this case, we're of course talking about the ZSBL. If we play close attention, we see that it touches the register `a2`.

At this point, I would like to encourage the reader to also read the section on ZSBL from [this article](#). The whole article is great, I just initially found it a little tough to go through, so consider this article a warmup for understanding that article, it's really worth going through.

Anyway, keeping this article watered down still – what is the significance of setting up the register `a2` in ZSBL? **It points to a `struct fw_dynamic_info`** which gives the dynamic OpenSBI flavor a way to continue going through the boot process! In fact, one of the piece of data in this struct is the address of the next piece of software running after OpenSBI! A good question to ask is: on a real machine, who populates this struct? Based on what we'll see below, it's obvious that QEMU hardcodes this content into the memory, and that logic is not part a of the ZSBL, but I can definitely imagine a device where ZSBL actually populates this struct and passes it on to OpenSBI.

Slide 17 of [this presentation](#) by an engineer from Western Digital (presumably a core contributor to OpenSBI) outlines the contents of this `struct`:

1. Magic number
2. Version
3. Next address
4. Next mode
5. Options

All of these are unsigned longs (I guess that means 64 bit, 8 bytes?).

Exploring the `fw_dynamic_info struct`

At this point, let's take a quick detour to make sure we're on the same page. Let's quickly make sure we're all looking at the same version of the OpenSBI because different systems have different version of QEMU which may come with a different version of OpenSBI. Building OpenSBI from source is really straightforward, so let's quickly do it. First, we need to clone the Git repo (time of writing of this article is 10th Sept 2023; if you want to achieve full reproducibility, build at a commit at this date):

```
git clone https://github.com/riscv-software-src/opensbi.git
cd opensbi
make ARCH=riscv CROSS_COMPILE=riscv64-linux-gnu- PLATFORM=generic
```

The build should be fairly fast and lightweight. The output file we're interested in is `build/platform/generic/firmware/fw_dynamic.bin`. We'll pass this through the `-bios` flag to QEMU. Starting QEMU with (from the `opensbi` folder we just cloned with Git):

```
qemu-system-riscv64 -machine virt -s -S -bios build/platform/generic/firmware/fw
```

After a few `sis` in `gdb`, we get back to where we were before. Let's poke QEMU's memory to see what's going on there at the end of ZSBL. At the last instruction of ZSBL, we look at the register dump (we use `i r` for this).

```
=> 0x0000000080000000: 33 04 05 00 add s0,a0,zero
```

```
(gdb) i r
```

ra	0x0	0x0
sp	0x0	0x0
gp	0x0	0x0
tp	0x0	0x0
t0	0x80000000	2147483648
t1	0x0	0
t2	0x0	0

```

fp          0x0  0x0
s1          0x0  0
a0          0x0  0
a1          0x87e00000  2279604224
a2          0x1028  4136
a3          0x0  0
a4          0x0  0
a5          0x0  0
a6          0x0  0
a7          0x0  0
s2          0x0  0
s3          0x0  0
s4          0x0  0
s5          0x0  0
s6          0x0  0
s7          0x0  0
s8          0x0  0
s9          0x0  0
s10         0x0  0
s11         0x0  0
t3          0x0  0
t4          0x0  0
t5          0x0  0
t6          0x0  0
pc          0x80000000  0x80000000

```

a2 is therefore pointing to **0x1028**. As we said, let's poke that memory with **gdb**. We'll ask it to read 10 successive 8-byte values starting from **0x1028**, and display them in hex format.

```
(gdb) x/10xg 0x1028
```

The **g** flag prints out the memory contents in 8-byte (giant) chunks.

```

(gdb) x/10xg 0x1028
0x1028: 0x000000004942534f 0x0000000000000002
0x1038: 0x0000000000000000 0x0000000000000001

```

```

0x1048: 0x0000000000000000  0x0000000000000000
0x1058: 0x0000000000000000  0x0000000000000000
0x1068: 0x0000000000000000  0x0000000000000000

```

This roughly seems to match Vysakh's article. We definitely see the magic described in that article, followed by the `0x02` info version. Next should be the address for the next jump, but there are all zeroes... This is strange, but let's keep looking. Next value is `0x01` which again, according to the article, should correspond to the next mode of execution which is `S`. This is correct, we're going from `M` mode running SBI to the `S` mode running the OS kernel bootloader, or the kernel itself, whatever we want. Why is the address of the next jump all zeroes though? At this point, I'll just let QEMU run without interference from `gdb`. I run the following in `gdb`:

```
continue
```

Everything is sort of hanging, but I got a newer OpenSBI output on UART since I am now running a newer version of OpenSBI:

```
OpenSBI v1.3-54-g901d3d7
```

```

      ----
    /  __ \
  | | | | _ __   ___ _ __ | (___ | |_) | | | | | | | |
  | | | | ' _ \ / _ \ ' _ \ \___ \| | _< | |
  | |__| | |_) | |__| | |_) | |_) | | |
  \___/| | .__/ \___| | | |___/|___/|___|
      | |
      | |

```

```

Platform Name       : riscv-virtio,qemu
Platform Features   : medeleg
Platform HART Count : 1
Platform IPI Device : aclint-mswi
Platform Timer Device : aclint-mtimer @ 100000000Hz
Platform Console Device : uart8250

```

```
Platform HSM Device      : ---
Platform PMU Device      : ---
Platform Reboot Device   : syscon-reboot
Platform Shutdown Device : syscon-poweroff
Platform Suspend Device  : ---
Platform CPPC Device     : ---
Firmware Base           : 0x80000000
Firmware Size           : 322 KB
Firmware RW Offset      : 0x40000
Firmware RW Size        : 66 KB
Firmware Heap Offset    : 0x48000
Firmware Heap Size      : 34 KB (total), 2 KB (reserved), 9 KB (used), 22 KB (
Firmware Scratch Size   : 4096 B (total), 768 B (used), 3328 B (free)
Runtime SBI Version      : 1.0

Domain0 Name            : root
Domain0 Boot HART       : 0
Domain0 HARTs           : 0*
Domain0 Region00        : 0x0000000002000000-0x000000000200ffff M: (I,R,W) S/U
Domain0 Region01        : 0x00000000080040000-0x0000000008005ffff M: (R,W) S/U:
Domain0 Region02        : 0x00000000080000000-0x0000000008003ffff M: (R,X) S/U:
Domain0 Region03        : 0x00000000000000000-0xfffffffffffffff M: ( ) S/U: (R,
Domain0 Next Address     : 0x0000000000000000
Domain0 Next Arg1        : 0x00000000087e00000
Domain0 Next Mode        : S-mode
Domain0 SysReset         : yes
Domain0 SysSuspend       : yes

Boot HART ID            : 0
Boot HART Domain        : root
Boot HART Priv Version   : v1.10
Boot HART Base ISA      : rv64imafdc
Boot HART ISA Extensions : zicntr
Boot HART PMP Count      : 16
Boot HART PMP Granularity : 4
Boot HART PMP Address Bits: 54
Boot HART MHPM Info      : 0 (0x000000000)
Boot HART MIDELEG        : 0x0000000000000222
Boot HART MEDELEG        : 0x000000000000b109
```

This matches what we saw above, the next address is all zeroes... This is strange, there's no way that could be true. I now ran QEMU without the initial pause, just letting it run and connecting with `gdb` asynchronously. I'll spare you the details, but inspecting the registers on that "live run" definitely showed to me that nothing is executing in the `0x0000000000000000` area. The CPU seems to be spinning around some other address.

This likely has something to do with the fact that I actually didn't pass any software to QEMU to load other than OpenSBI, so that's probably what's throwing it off. QEMU likely populated the struct in memory with all zeroes, and OpenSBI identifies it as an illegal edge case, so it just keeps spinning in OpenSBI forever – this is my educated guess.

How do we pass some software to run other than OpenSBI? **The same way we passed OpenSBI, just a different flag name!** This time, we're using the `-kernel` QEMU flag. And how are we going to build this software? The same way we built the "fake BIOS" in our previous article, we'll just map it to a different memory location. Let's give it a shot at `0x80200000`.

Building an "infinite-loop fake kernel"

Our OS kernel will just spin in place. It will be a single jump instruction at `0x80200000` that just stays there infinitely. Here's the assembly source code:

```
.global _start
.section .text.kernel

_start: j _start
```

The linker script is the following:

```
MEMORY {
```

```

    kernel_space (rwx) : ORIGIN = 0x80200000, LENGTH = 128
}

SECTIONS {
    .text : {
        infinite_loop.o(.text.kernel)
    } > kernel_space
}

```

*For details on how to use these files to build an **ELF** image that can be loaded into QEMU, please see the original bare metal programming article.*

Once we build it, we end up with the **infinte_loop** **ELF** file that can serve as our fake kernel. We now run QEMU

```
qemu-system-riscv64 -machine virt -s -S -bios build/platform/generic/firmware/fw
```

Again, I connect **gdb** and **si** my way to the end of ZSBL. Now when I read the infamous struct at **0x1028**, things look a lot better, which confirms the theory that QEMU was populating that struct weirdly.

```

=> 0x0000000008000000:  33 04 05 00 add s0,a0,zero
(gdb) x/10xg 0x1028
0x1028: 0x000000004942534f  0x0000000000000002
0x1038: 0x0000000008020000  0x0000000000000001
0x1048: 0x0000000000000000  0x0000000000000000
0x1058: 0x0000000000000000  0x0000000000000000
0x1068: 0x0000000000000000  0x0000000000000000

```

We now see that the new address is populated in this struct, as is expected. This is also reflected in the OpenSBI output on UART. Let's continue to our fake kernel with **gdb** and see if everything is OK there.

```
(gdb) break *0x080200000
```

```
Breakpoint 1 at 0x80200000
```

```
(gdb) continue
```

```
Continuing.
```

```
Breakpoint 1, 0x0000000080200000 in ?? ()
```

```
=> 0x0000000080200000: 6f 00 00 00 j 0x80200000
```

Everything looks good here. Let's recap:

1. ZSBL is the first thing that runs after the power-on. It initializes a few registers. The key register is `a2`, which points to a `fw_dynamic_info` struct containing the crucial info for the `FW_DYNAMIC` flavor of OpenSBI to operate. In QEMU case, this struct is somehow populated during the power-on, magically by the virtualization engine, but in reality, this is **likely** the job of the ZSBL. Either way, OpenSBI now knows what to do after it's done.
2. OpenSBI provides an interrupt-based interface for the software up on the stack (presumably OS kernel bootloader and kernel itself) to invoke it. This interface is called SBI and it's conceptually the same as ABI for the application software on top of an operating system.
3. We pass the kernel image to QEMU as yet another ELF which just populated another section of the memory. QEMU populates the struct in such way that OpenSBI can pass the control to there, and before it switches there, it enters the `S` mode of execution.

Intentionally skipped details

ZSBL also touched the `a0` and `a1` registers.

`a0` has something to do with RISC-V harts, but let's not get into those details, they are not relevant for the rest of this article. Besides, this particular step in the boot process doesn't seem to be particularly relevant, per comments from Github.

`a1` is an interesting pointer because it points to the **device tree** data

structure in memory. For the rest of this article, this data structure is not relevant, so we can disregard this piece of information. However, the device tree is really useful for real kernels like Linux. Linux is able to scan the device tree from memory and understand the structure of the machine it's running on, rather than having to run a lot of `if/else` branches in its programming for every hardware combination. The Wikipedia article should give a decent idea of how this is used in Linux. As mentioned, however, we won't be concerned with the details of device tree in the rest of this article.

Hello world fake kernel

Now we have all the knowledge we need to code a fake OS kernel that just prints "Hello world" to the UART device. The functionality is not at all different from the bare metal program we looked at in the previous guide, but the way we'll get there is significantly different. We'll be using an SBI call to print to UART, instead of directly interacting with the UART device (we're using a more privileged lower layer of software to do this work for us). This could have serious consequences, even on a trivial example such as a "hello world" one: **we delegate the responsibility of interacting with the UART hardware to the SBI layer, thus achieving portability across different machines that conform to this SBI interface.**

How do we call into RISC-V SBI layer? Conceptually, it's exactly the same as invoking a print to standard output in x86 Linux – we'll populate some registers and invoke a software interrupt/trap to pass the control down the software stack to OpenSBI. OpenSBI offers a lot of services in the SBI layer, and many of them can be extremely useful for developing a portable operating system kernel, such as interaction with the timers (relevant for time slicing and enabling multiple threads to share the same CPU core). For the full list of functionality exposed through the SBI layer, please take a look here.

In this guide, we'll be focusing on the debug console functionality, i.e. we'll be writing out to UART through SBI. Let's code!

First, we need to know how do we encode the functionality we want OpenSBI to execute through registers. This is well documented [here](#). tl;dr is that SBI functionality is grouped into "extensions". Register **a7** contains the extension ID (EID), while **a6** encodes the individual function ID (FID) within that extension. The parameters are then passed through **a0**, **a1**, **a2**, ...

For printing to the console, the EID we are looking for is **0x4442434E** (a rather interesting value) and the FID is simply **0x00**.

This time, instead of printing one by one character as we did in the initial bare metal programming guide, we'll invoke the printing as a single operation. After all, we should be benefiting from the high level functionality that the SBI layer offers. Therefore, our binary should store the output string somewhere in the memory, and ideally we want to do something like invoking the SBI to print from that address. We'll do just that:

```
.global _start
.section .text.kernel

_start: li a7, 0x4442434E
        li a6, 0x00
        li a0, 12
        lla a1, debug_string
        li a2, 0
        ecall

loop:   j loop

.section .rodata
debug_string:
        .string "Hello world\n"
```

A couple of things to note here:

1. We use PC-relative addressing here for the output string. As a reminder, the kernel is stored at an address represented by a very large unsigned integer. This value is too high to be encoded within any RISC-V 32-bit instruction word. That's not a problem, we simply use a short sequence of **AUIPC** and **ADDI** instructions to get there (check out [this article](#) for more information on this). If you do not understand what this point is all about, please make sure to revise different memory addressing modes and the differences between them: this is crucial for any sort of bare metal programming. And instead of using **AUIPC** and **ADDI** in sequence, since this is a common pattern, the RISC-V assembler has a **pseudoinstruction LLA**, which we are using here.
2. SBI for some reason asks for the pointer to the string to be printed to be broken down into two pieces. One piece, as you can see is just 0. I am not entirely sure why this is needed, but that's the API.

So our SBI call is defined by several registers:

1. **a7** identifies the SBI extension
2. **a6** identifies the function within the extension (in this case, debug console extension)
3. **a0** contains the length of the string that needs to go to the debug console output
4. **a1** and **a2**, when joined together, contain the 64-bit pointer to the address of the string that needs to be printed

The SBI call is now invoked through an **ecall** instruction, which activates a CPU trap. At this point, OpenSBI takes over and writes to UART, in exactly the same way as we did in the initial bare metal programming guide. If you are wondering how a simple **ecall** invocation takes us to OpenSBI, that is because OpenSBI set up the trap handling mechanism in such way that when our kernel gets into a trap, the

program counter will jump into the OpenSBI software section. The details of this are way outside the scope of this article, but we may cover this in some other article.

For now, just check out the QEMU serial port and confirm that “Hello world” is printed properly:

```
qemu-system-riscv64 -machine virt -s -S -bios build/platform/generic/firmware/fw
```

```
OpenSBI v1.3-54-g901d3d7
```

```

      ----
    /  __ \
| | | | | _ _ _ _ _ _ | ( _ _ | | ) | | | | | |
| | | | | ' _ \ / _ \ ' _ \ \ _ _ \ | _ < | |
| | _ | | | ) | _ _ / | | | _ _ ) | | ) | | |
 \ _ _ / | . _ _ \ _ _ | | | _ _ _ / | _ _ / _ _ _ |
      | |
      | _ |

```

```

Platform Name           : riscv-virtio,qemu
Platform Features       : medeleg
Platform HART Count     : 1
Platform IPI Device     : aclint-mswi
Platform Timer Device   : aclint-mtimer @ 100000000Hz
Platform Console Device : uart8250
Platform HSM Device     : ---
Platform PMU Device     : ---
Platform Reboot Device  : syscon-reboot
Platform Shutdown Device: syscon-poweroff
Platform Suspend Device : ---
Platform CPPC Device    : ---
Firmware Base           : 0x80000000
Firmware Size           : 322 KB
Firmware RW Offset      : 0x40000
Firmware RW Size        : 66 KB
Firmware Heap Offset    : 0x48000
Firmware Heap Size      : 34 KB (total), 2 KB (reserved), 9 KB (used), 22 KB (

```

```

Firmware Scratch Size      : 4096 B (total), 768 B (used), 3328 B (free)
Runtime SBI Version        : 1.0

Domain0 Name               : root
Domain0 Boot HART          : 0
Domain0 HARTs               : 0*
Domain0 Region00           : 0x0000000002000000-0x000000000200ffff M: (I,R,W) S/U
Domain0 Region01           : 0x00000000080040000-0x0000000008005ffff M: (R,W) S/U:
Domain0 Region02           : 0x00000000080000000-0x0000000008003ffff M: (R,X) S/U:
Domain0 Region03           : 0x00000000000000000-0xffffffffffffffff M: () S/U: (R,
Domain0 Next Address       : 0x00000000080200000
Domain0 Next Arg1          : 0x00000000087e00000
Domain0 Next Mode          : S-mode
Domain0 SysReset           : yes
Domain0 SysSuspend         : yes

Boot HART ID               : 0
Boot HART Domain           : root
Boot HART Priv Version     : v1.10
Boot HART Base ISA         : rv64imafdc
Boot HART ISA Extensions   : zicntr
Boot HART PMP Count        : 16
Boot HART PMP Granularity  : 4
Boot HART PMP Address Bits : 54
Boot HART MHPM Info        : 0 (0x000000000)
Boot HART MIDELEG          : 0x00000000000000222
Boot HART MEDELEG          : 0x0000000000000b109
Hello world

```

As an exercise, I suggest probing the base extension (0x10) with `gdb` to investigate what the QEMU machine + OpenSBI you build are capable of offering.

Conclusion

We ended up with an entirely portable fake kernel that prints “Hello world” to UART! This may seem like nothing special, but the concept

here is very powerful. Without rebuilding, you can drop the same kernel image on a different RISC-V 64-bit machine with OpenSBI that supports the debug console extension.

In fact, I played a little trick here. :) One of the main reasons I suggested building OpenSBI from source is that some QEMU versions provided by the Linux distro package managers do not support the debug console extension (they're simply old). This was the case with my default OpenSBI which came with Debian's version of QEMU.

Finally, I would like to remind the reader that we have extensively focused on the QEMU virt machine with a RISC-V core and all the fine details of this article are related to it. That said, my hope is that the reader has learned enough about the boot sequence concepts and bare metal programming that adapting this knowledge to a particular real-world scenario becomes easy.

In the next posts, we'll talk about taking this further and booting up a full blown Linux kernel. We'll expand that step by step until we reach a Linux deployment that can handle I/O with keyboard, mouse, screen and Ethernet network.

I hope you enjoyed this lengthy writeup!

Code pointers

If you prefer not to copy/paste, the code is available on [this GitHub repo](#).

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

Copyright © 2025 | All rights reserved.

