



## Tools for RISC-V SoC Bring-up

Nick Kossifidis (FORTH)

*Disclaimer:*

*"Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Health and Digital Executive Agency (HaDEA). Neither the European Union nor the granting authority can be held responsible for them."*



Call: Open source for cloud-based services, GA Nr: 101092993 (HaDEA)

- Design of individual CPU units
  - ALU, FPU, VPU, MMU, ...
- Verification of individual units
  - e.g. directed/random tests, in a simulator
- Verification of the whole core
  - e.g. RISC-V ACT suite, checks against the SAIL model/reference simulator
- Post-synthesis co-simulation tests
- Integration with other IPs
  - Each IP with its own set of pre/post-synthesis tests
- ...

- More advanced bare-metal tests for verification of the core
  - e.g. parts of the RISC-V spec not covered by ACT, custom extensions
- Progressively more complex bare-metal platform-level tests
  - e.g. interrupt delivery/delegation, communication between peripherals, peripheral operation
- Memory subsystem tests
  - e.g. litmus, cache-coherency with peripherals, IOMMU
- Security-related tests
  - e.g. constant-time requirements, TRNG operation, MTT, xPMP
- Stress testing/profiling/benchmarking
- ...

- Even after the whole process, things can still go wrong !
- Booting a full-blown Linux distro greatly expands test coverage ... and complexity
- Tracking down HW bugs in such a setup is a nightmare!
- We need a strategy to progressively expand test coverage
- Also a strategy that may be used in constrained environments, e.g. in case parts of the SoC / PCB end up not working after tape-out/assembly (e.g. no DRAM)

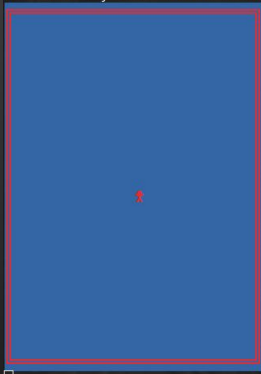
- Custom C environment, some libc functions, no OS
- Very small tools mainly focused on {stress-}testing
- Also used for writing simple drivers to test peripherals
  - No device tree parsing, simple headers instead
- Custom linker script to support running from ROM
  - Our BootROM is also built using this process
- Generates binaries also for QEMU so that we can directly compare results between the QEMU and HW
  - Can also be done for other emulators
  - We can even run the same binaries with HW by writing a QEMU model for our HW
- As simple as possible, makes it easy to port simple C tools
- Also use it for educational purposes

```
#define BM_CPU_CYCLES_PER_SEC 1000000
#define BM_UART_BASE_ADDR 0x10000000
#define BM_UART_CLOCK_HZ 3686400
#define BM_UART_BAUD_RATE 115200
#define BM_UART_REG_SHIFT 0
#define BM_UART_IRQ 0xa
#define BM_CLINT_BASE_ADDR 0x2000000
#define BM_TIMEBASE_FREQ_HZ 10000000
#define BM_PLIC_BASE_ADDR 0xc000000
#define BM_PLIC_NUM_SOURCES 0x35
```



# Some examples...

```
BareMetal loader (c) FORTH/CARV 2019
-----
Hart 0 active
===== INTERACTIVITY TEST =====
Press 1 to start...
Welcome to interactivity test
Use arrow keys to move the cursor inside the box
```



```
mick@Gazofonias ~/Workspace/BareMetal $ ./test_on_qemu.sh build/bm_coremark.qemu.bin
BareMetal loader (c) FORTH/CARV 2019
-----
Hart 0 active
2K performance run parameters for coremark.
CoreMark Size : 666
Total ticks : 12612063
Total time (secs): 12.000000
Iterations/Sec : 5000.000000
Iterations : 60000
Compiler version : GCC9.2.0
Compiler flags : -O2
Memory location : STATIC
seedcrc : 0xe9f5
[0]crclist : 0xe714
[0]crcmatrix : 0x1fd7
[0]crcstate : 0x8e3a
[0]crcfinal : 0xbd59
Correct operation validated. See README.md for run and reporting rules.
CoreMark 1.0 : 5000.000000 / GCC9.2.0 -O2 / STATIC
```

```
mick@Gazofonias ~/Workspace/BareMetal $ ./test_on_qemu.sh build/bm_spmv.qemu.bin
BareMetal loader (c) FORTH/CARV 2019
-----
Hart 0 active
Test status: PASS, cycles: 1880082, instruction count: 18723816
Test status: PASS, cycles: 1817374, instruction count: 18170712
Test status: PASS, cycles: 1790042, instruction count: 17899200
Test status: PASS, cycles: 1799351, instruction count: 17992008
Test status: PASS, cycles: 1796083, instruction count: 17959500
Test status: PASS, cycles: 1802624, instruction count: 18024264
Test status: PASS, cycles: 1794380, instruction count: 17942184
Test status: PASS, cycles: 1790301, instruction count: 17901504
```

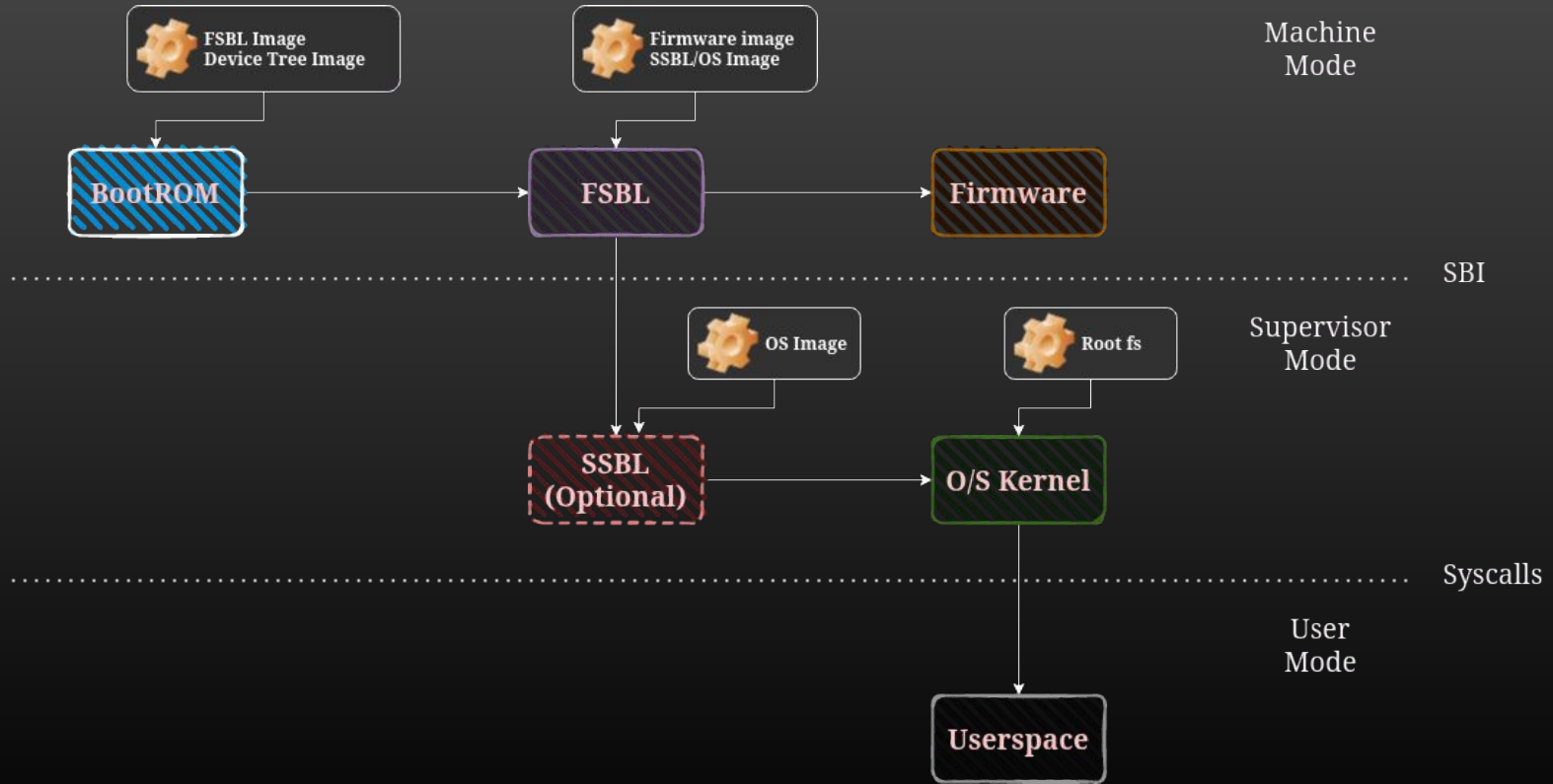
```
mick@Gazofonias ~/Workspace/BareMetal $ ./test_on_qemu.sh build/bm_uart_test.qemu.bin
BareMetal loader (c) FORTH/CARV 2019
-----
===== UART WRITE TEST =====
! "#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNQPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
===== UART LOOPBACK TEST =====
```

```
BareMetal loader (c) FORTH/CARV 2019
-----
Hart 0 active
[EmacLite] using MAC address: 3B:B1:78:2C:F8:85
[DHCP] Sending discovery...
[Net] Got ip: 10.0.2.15 subnet mask: 255.255.255.0 gateway: 10.0.2.2
[TFTP] Initialized with server ip: 10.0.2.2 and boot filename: test
[TFTP] Downloading sdv3.bin
[TFTP] negotiated block size: 1428
```

```
mick@Gazofonias ~/Workspace/BareMetal $ ./test_on_qemu.sh build/bm_sha3_test.qemu.bin
BareMetal loader (c) FORTH/CARV 2019
-----
Hart 0 active
Test start

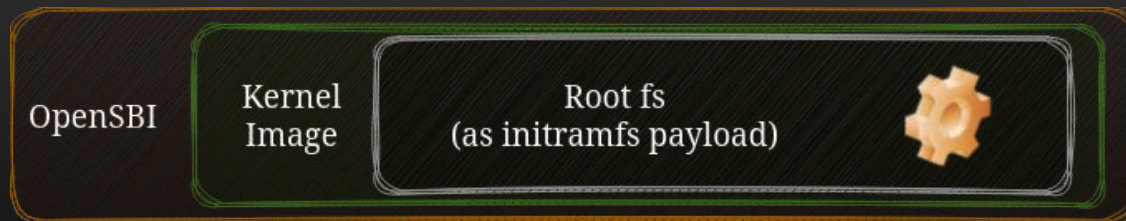
Test round: 0
SHA3-256 of empty string: A7FFC6F8BF1ED76651C14756A061D662F580FF40E43849A82D80A4B00F8434A
SHA3-512 of empty string: A69F73CA2A2A9AC5C8B567DC185A756E97C982164FE25859E0D1DC1475C80A615B2123AF1F5F94C11E3E9402C3AC558F500199D95B6D3E301758586281DCD26
SHA3-256 of "abc": 3A985DA74FE225B2045C172D6BD390B0855F086E3E9D525B46BF24511431532
SHA3-512 of "abc": B751850B1A57168A5693CD924B6B096E08F621827444F70D884F5D0240D2712E10E116E9192AF3C91A7EC57647E3934057340B4CF408D5A56592F8274EEC35F0
SHA3-256 of "test": 36F028580BB02CC8272A9A020F4200E346E276AE664E45EE80745574E2F5AB80
SHA3-512 of "test": 180F0C0DDA5A20D5722A06A75C3D56FC1FD183036E238ED7477343EEC75E68BF5B882C10E37CE4C987B6EED3DFEEB7F685CEBA8CC3E41502F6E49ECFB954
SHA3-256 of 1m1l 'a's: 5C8875AE47A3634BA4FD55EC858FDF661F32ACA75C6D69D0DCB6C115891C1
SHA3-512 of 1m1l 'a's: 3C3A876DA14034AB60627C077B98F7E120A2A5370212DFB3385A18D4F38859ED311D0A9D5141CE9C5C66EE689B266A8AA18AC8E282A0E0DB596C90B0A7887
Round took 260725 cycles
```

# The way to full Linux boot



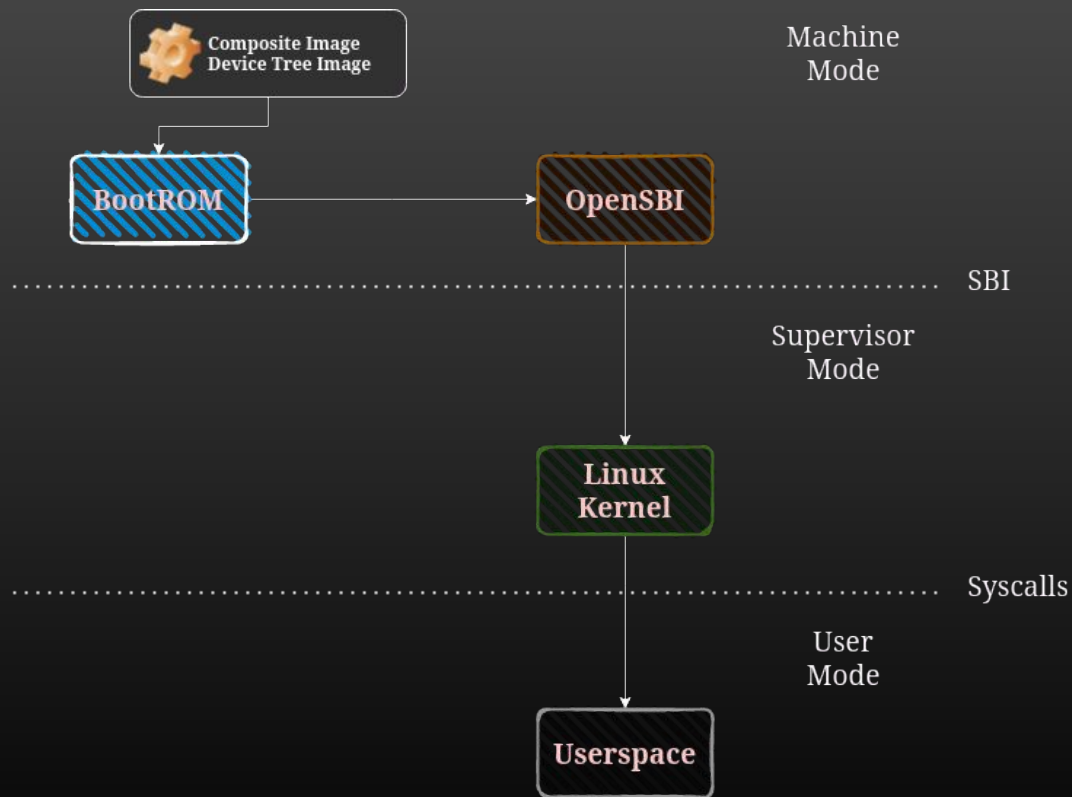
# Simplifying the Linux boot process

- Use OpenSBI, a firmware implementation that also acts as FSBL
- Get rid of SSBL and jump to Linux kernel directly
- Reduce number of external images
  - Kernel image as an OpenSBI payload
  - Root FS included as initramfs in the kernel image

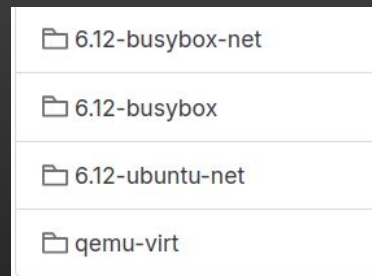




# Simplifying the Linux boot process



- Start with a bare minimum kernel configuration
  - No networking, no storage, NOMMU
  - Limited functionality
- Move on to a more complex kernel configuration
  - With networking, storage, multiple users, ...
- Finally, a full-blown kernel configuration
  - With systemd support and everything needed to boot a fully-featured Linux distro.
- We automate the build process using yarvt targets:
  - <https://gitlab.riser.cloudsigma.com/riser/riser-os>



- Start with a tiny rootfs with only busybox
  - statically linked
- Add more tools and networking support
  - e.g. iperf, ssh
- Use an off-the-shelf rootFS of a full-blown Linux distro
  - Start with Alpine, also built around busybox (and it uses musl like we do for our small rootfs)
  - Move to ubuntu
- Yarvt can build / initialize rootFSes automatically

```
mick@Gazofonias ~/Workspace/riser-os $ ./yarvt 6.12-busybox help
----- Yet another RISC-V tool v0.9 -----
TARGET: 6.12-busybox
6.12-busybox commands:
  help/usage: Print this message
  bootstrap: (Re)Build unified image (osbi + Linux + rootfs)
  run_on_qemu: Test unified image on QEMU

mick@Gazofonias ~/Workspace/riser-os $ ./yarvt 6.12-ubuntu-net help
----- Yet another RISC-V tool v0.9 -----
TARGET: 6.12-ubuntu-net
6.12-ubuntu-net commands:
  help/usage: Print this message
  bootstrap: (Re)Build unified image (osbi + Linux)
  run_on_qemu: Test unified image on QEMU
    <arg> Rootfs path on host's NFS server
  boot_node: Run unified image on QEMU, in a multi-instance scenario
    <arg> Rootfs path on host's NFS server
    <arg> Node ID (1 - 253)
  run_installer: Build and run the installer script on the rootfs
    <arg> Distro to install: ubuntu / alpine
    <arg> Rootfs path on host's NFS server
```

# Why NOMMU Linux

- MMU is a common source of HW bugs in our experience
  - Microarchitectural bugs that are hard to reproduce in simple tests we previously did
  - Especially when we go multicore
- Why not go for a simple RTOS (e.g. FreeRTOS, Zephyr)
  - Using standard tools (e.g. busybox, iperf) would be harder (different syscall API)
  - Building the image would be more complicated (need to go through an SDK etc)
  - Usually support only M-mode/U-mode setups
  - Would be harder to compare behavior between MMU/NOMMU

# NOMMU Linux basics

- Different memory allocators: mm/nommu.c
- Limitations on mmap: Documentation/nommu-mmap.txt
  - **No memory protection**
  - **No fork() support**
    - fork() relies on COW, but vfork() is supported
  - **No overcommit / lazy binding**
  - **No swap**
  - **No dynamic heap/stack**
    - avoid using alloca(), brk(), sbrk(), use malloc()/free() instead
  - **No MAP\_SHARED on files**
    - in general MAP\_SHARED functionality is limited
  - **No MAP\_FIXED**
  - **Limitations on MAP\_PRIVATE**
    - no COW/paging
  - **Excessive fragmentation, avoid large mappings**

- When MMU is available, `BINFMT_ELF` loader is used to load executables / shared libraries.
- Without MMU, alternative loaders/binary formats are used
  - `BINFMT_FLAT`
    - Stripped down ELF (through `elf2flt`)
    - No dynamic loading (`libld`)
    - No shared libraries
    - Limitations on executable's size
  - `BINFMT_ELF_FDPIC`
    - Position Independent (PIC/PIE) ELF, no `ET_EXEC` support
    - Support for shared libraries through function descriptors (hence FD)
    - Support for dynamic loading (`libld`)
    - May also be used when MMU is enabled
- Alternative toolchains also required
  - based on `µClibc` or `musl`



# NOMMU Linux basics

```
mick@Gazofonias ~/Workspace/yarvt-carv/build/6.7-busybox/RV64I/rootfs $ file bin/busybox
bin/busybox: ELF 64-bit LSB executable, UCB RISC-V, RVC, double-float ABI, version 1 (SYSV), statically linked, stripped
mick@Gazofonias ~/Workspace/yarvt-carv/build/6.7-busybox/RV64I/rootfs $ readelf -h bin/busybox
```

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00
  Class:   ELF64
  Data:    2's complement, little endian
  Version: 1 (current)
  OS/ABI:  UNIX - System V
  ABI Version: 0
  Type:    EXEC (Executable file)
  Machine: RISC-V
  Version: 0x1
  Entry point address: 0x10172
  Start of program headers: 64 (bytes into file)
  Start of section headers: 550376 (bytes into file)
  Flags:   0x5, RVC, double-float ABI
  Size of this header: 64 (bytes)
  Size of program headers: 56 (bytes)
  Number of program headers: 5
  Size of section headers: 64 (bytes)
  Number of section headers: 14
  Section header string table index: 13
```

```
mick@Gazofonias ~/Workspace/yarvt-carv/build/6.7-busybox/RV64I/r
```

```
mick@Gazofonias ~/Workspace/yarvt-carv/build/6.7-busybox-nommu/RV64I/rootfs $ file bin/busybox
bin/busybox: ELF 64-bit LSB pie executable, UCB RISC-V, RVC, double-float ABI, version 1 (SYSV), dynamically linked, interpreter /lib/ld-uClibc.so.0, stripped
mick@Gazofonias ~/Workspace/yarvt-carv/build/6.7-busybox-nommu/RV64I/rootfs $ readelf -h bin/busybox
```

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00
  Class:   ELF64
  Data:    2's complement, little endian
  Version: 1 (current)
  OS/ABI:  UNIX - System V
  ABI Version: 0
  Type:    DYN (Position-Independent Executable file)
  Machine: RISC-V
  Version: 0x1
  Entry point address: 0x47fe
  Start of program headers: 64 (bytes into file)
  Start of section headers: 609208 (bytes into file)
  Flags:   0x5, RVC, double-float ABI
  Size of this header: 64 (bytes)
  Size of program headers: 56 (bytes)
  Number of program headers: 9
  Size of section headers: 64 (bytes)
  Number of section headers: 22
  Section header string table index: 21
```

```
mick@Gazofonias ~/Workspace/yarvt-carv/build/6.7-busybox-nommu/RV
```

```
mick@Gazofonias ~/Workspace/yarvt-carv/build/6.7-busybox-nommu/RV64I/rootfs $ file lib/ld-uClibc.so.0
lib/ld-uClibc.so.0: ELF 64-bit LSB shared object, UCB RISC-V, RVC, double-float ABI, version 1 (SYSV), static-pie linked, stripped
mick@Gazofonias ~/Workspace/yarvt-carv/build/6.7-busybox-nommu/RV64I/rootfs $ readelf -h lib/ld-uClibc.so.0
```

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00
  Class:   ELF64
  Data:    2's complement, little endian
  Version: 1 (current)
  OS/ABI:  UNIX - System V
  ABI Version: 0
  Type:    DYN (Shared object file)
  Machine: RISC-V
  Version: 0x1
  Entry point address: 0xdec
  Start of program headers: 64 (bytes into file)
  Start of section headers: 20712 (bytes into file)
  Flags:   0x5, RVC, double-float ABI
  Size of this header: 64 (bytes)
  Size of program headers: 56 (bytes)
  Number of program headers: 7
  Size of section headers: 64 (bytes)
  Number of section headers: 17
  Section header string table index: 16
```

```
mick@Gazofonias ~/Workspace/yarvt-carv/build/6.7-busybox-nommu/RV64I/rootfs $ █
```

- Initial support added on Linux 5.5
  - Only M-mode/U-mode scenario
  - Mainly to support the Kendryte K210 that had a non-compliant MMU
- Almost declared deprecated on Feb. 2024
  - But after community feedback, it was not deprecated
  - Instead, new patches came up and support keeps getting better
  - Support for running NOMMU Linux on S-mode
  - Still needs further work though



- FLAT binaries supported, but won't work for us
  - Due to our custom memory layout
- ELF psABI for FDPIC support is still WiP
  - But we can at least run busybox
  - ... and support for 64bit is also there
- $\mu$ Clibc added support for RISC-V
  - Recently FORTH contributed  $\mu$ Clibc-ng support to the official RISC-V toolchain repo:  
<https://github.com/riscv-collab/riscv-gnu-toolchain/pull/1475>
  - And also fixed the CI to provide pre-built toolchains:  
<https://github.com/riscv-collab/riscv-gnu-toolchain/pull/1608>



# Testing MMU vs NOMMU

```

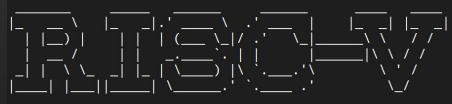
0.000000[ T0] Linux version 6.7.12-busybox-dirty (root@Gazofonias) (riscv64-unknown-linux-gnu-gcc (gc891d8dc23e) 13.2.0, GNU ld (
0.000000[ T0] random: crng init done
0.000000[ T0] Machine model: eupilot-qemu
0.000000[ T0] SBI specification v2.0 detected
0.000000[ T0] SBI implementation ID=0x1 Version=0x10004
0.000000[ T0] SBI TIME extension detected
0.000000[ T0] SBI IPI extension detected
0.000000[ T0] SBI RFENCE extension detected
0.000000[ T0] SBI SRST extension detected
0.000000[ T0] earlycon: ns16550a0 at MMIO 0x0000040010000000 (options '')
0.000000[ T0] printk: legacy bootconsole [ns16550a0] enabled
0.000000[ T0] Disabled 4-level and 5-level paging
0.000000[ T0] OF: reserved mem: 0x0000800000400000..0x000080000043ffff (256 KiB) nomap non-reusable mmode_resv188000,400000
0.000000[ T0] OF: reserved mem: 0x0000800000440000..0x000080000045ffff (128 KiB) nomap non-reusable mmode_resv088000,440000
0.000000[ T0] Zone ranges:
0.000000[ T0] DMA32 empty
0.000000[ T0] Normal [mem 0x0000800000400000-0x0000800000803fffff]
0.000000[ T0] Movable zone start for each node
0.000000[ T0] Early memory node ranges
0.000000[ T0] node 0: [mem 0x0000800000400000-0x000080000045fffff]
0.000000[ T0] node 0: [mem 0x0000800000460000-0x0000800000803fffff]
0.000000[ T0] Initmem setup node 0 [mem 0x0000800000400000-0x0000800000803fffff]

```

```

0.000000[ T0] Linux version 6.7.12-busybox-nommu-dirty (root@Gazofonias) (riscv64-unknown-linux-gnu-gcc (gc891d8dc23e) 13.2.0,
0.000000[ T0] random: crng init done
0.000000[ T0] OF: fdt: Ignoring memory range 0x8000004000000 - 0x800000600000
0.000000[ T0] Machine model: eupilot-qemu
0.000000[ T0] SBI specification v2.0 detected
0.000000[ T0] SBI implementation ID=0x1 Version=0x10004
0.000000[ T0] SBI TIME extension detected
0.000000[ T0] SBI IPI extension detected
0.000000[ T0] SBI RFENCE extension detected
0.000000[ T0] SBI SRST extension detected
0.000000[ T0] earlycon: ns16550a0 at MMIO 0x0000040010000000 (options '')
0.000000[ T0] printk: legacy bootconsole [ns16550a0] enabled
0.000000[ T0] OF: reserved mem: 0x0000800000400000..0x000080000043ffff (256 KiB) nomap non-reusable mmode_resv188000,400000
0.000000[ T0] OF: reserved mem: 0x0000800000440000..0x000080000045ffff (128 KiB) nomap non-reusable mmode_resv088000,440000
0.000000[ T0] Zone ranges:
0.000000[ T0] DMA32 empty
0.000000[ T0] Normal [mem 0x0000800000600000-0x0000800000403fffff]
0.000000[ T0] Movable zone start for each node
0.000000[ T0] Early memory node ranges
0.000000[ T0] node 0: [mem 0x0000800000600000-0x0000800000403fffff]
0.000000[ T0] Initmem setup node 0 [mem 0x0000800000600000-0x0000800000403fffff]

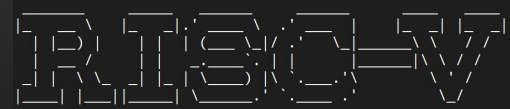
```



```

Linux 6.7.12-busybox-dirty #2 SMP Fri Jun 14 10:41:33 EEST 2024 riscv64 unknown
login[75]: root login on 'ttyS0'
root@eupilot: /root # cat /proc/self/maps
00010000-00096000 r-xp 00000000 00:02 3081 /bin/busybox
00096000-00098000 rw-p 00085000 00:02 3081 /bin/busybox
00098000-00099000 rw-p 00000000 00:00 0
3fa82e1000-3fa82e5000 rw-p 00000000 00:00 0
3fa82e5000-3fa82e7000 r-p 00000000 00:00 0
3fa82e7000-3fa82e8000 r-xp 00000000 00:00 0
3fe1b3a000-3fe1b5b000 rw-p 00000000 00:00 0
root@eupilot: /root # cat /proc/iomem
40010000000-40010000fff : serial
800000400000-80000045ffff : Reserved
800000460000-8000803fffff : System RAM
800000601000-8000010c74e7 : Kernel image
800000601000-80000078e44b : Kernel code
800000c00000-800000dfffff : Kernel rodata
800001000000-80000108dd97 : Kernel data
80000108e000-8000010c74e7 : Kernel bss
root@eupilot: /root #

```



```

Linux 6.7.12-busybox-nommu-dirty #2 Fri Jun 14 12:48:02 EEST 2024 riscv64 unknown
Jan 1 00:00:04 login[56]: root login on 'ttyS0'
/root # cat /proc/self/maps
800001188000-80000118f000 rwxp 00000000 00:00 0
80000118f000-800001190000 rw-p 00000000 00:00 0
800001194000-800001198000 rw-p 00000000 00:00 0
8000012a0000-8000012c0000 rw-p 00000000 00:00 0
800001500000-80000159c000 rwxp 00000000 00:00 0
/root # cat /proc/iomem
40010000000-40010000fff : serial
800000600000-8000403fffff : System RAM
800000601000-800000893a6f : Kernel image
800000601000-800000769677 : Kernel code
8000007e0100-800000819c3f : Kernel rodata
800000819dc0-80000087533f : Kernel data
800000876000-800000893a6f : Kernel bss
/root #

```

Add more tests to our collection...

Open source our BareMetal stuff (most of it)

Contribute our NOMMU fixes on upstream Linux

Keep up with upstream projects and adapt (even more tests)

Add new profiles to yarvt for UEFI / ACPI support

Any suggestions ?



Thank you for your attention. Questions and comments ?

*Disclaimer:*

*"Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Health and Digital Executive Agency (HaDEA). Neither the European Union nor the granting authority can be held responsible for them."*

