Embedded Linux from scratch in 50 minutes (on RISC-V)

Michael Opdenacker
michael.opdenacker@bootlin.com

bootlin
embedded Linux and kernel engineering

Michael Opdenacker

- Founder and Embedded Linux engineer at Bootlin:
  - Embedded Linux **expertise**
  - **Development**, consulting and training
  - Focusing **only on Free and Open Source Software**
- About myself:
  - Always happy to learn from every new project, and share what I learn.
  - Initial author of Bootlin's freely available embedded Linux, kernel and boot time reduction training materials (https://bootlin.com/docs/)
  - Documentation maintainer for the Yocto Project

About this presentation

- This presentation is an update to a talk I made in 2021
  https://bootlin.com/pub/conferences/2021/fosdem/opdenacker-embedded-linux-45minutes-riscv/
- This presentation is available under the same Creative-Commons Attribution Share-Alike 3.0 license
- I'm doing this presentation on my own behalf.
  This doesn't represent opinions or statements from Bootlin.

Introduction

What I like in embedded Linux

- Linux is perfect for operating devices with a fixed set of features.
  Unlike on the desktop, Linux is almost in every existing embedded system.
- Embedded Linux makes Linux easy to learn: just a few programs and libraries are sufficient. **You can understand the usefulness of each file in your filesystem**.
- The Linux kernel is standalone: no complex dependencies against external software. The code is in C (or Rust)!
- Linux works with just a few MB of RAM and storage
- There's a new version of Linux every 2-3 months.
- Relatively small development community. You end up meeting lots of familiar faces at technical conferences (like the Embedded Linux Conference).
- Lots of opportunities (and funding available) for becoming a contributor (Linux kernel, bootloader, build systems...).

Goals

Show you the most important aspects of embedded Linux development work

- Building a cross-compiling toolchain
- Creating a disk image
- Booting a using a bootloader
- Loading and starting the Linux kernel
- Building a root filesystem populated with basic utilities
- Configuring the way the system starts
- Setting up networking and controlling the system via a web interface
- Do this on QEMU and on real hardware!

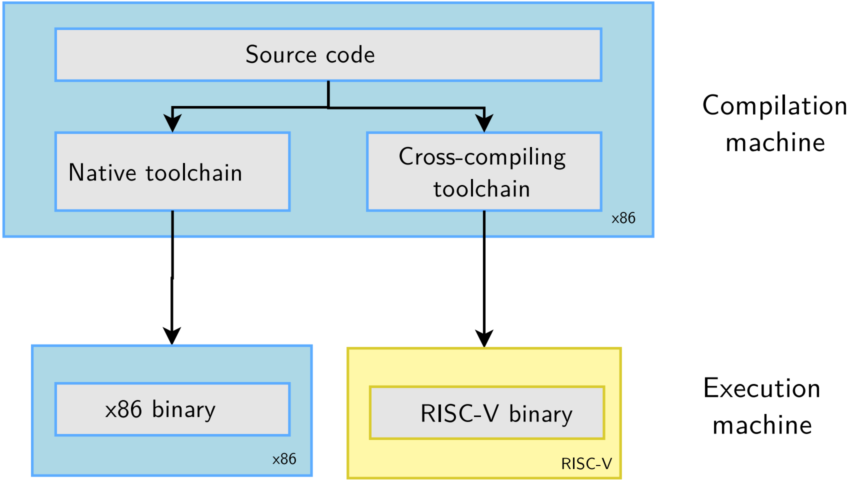
Things to build today

- Cross-compiling toolchain: *Buildroot 2024.02.1 (LTS)*
- Firmware / first stage bootloader: *OpenSBI*
- Bootloader: *U-Boot 2024.04*
- Kernel: *Linux 6.8.x*
- Root filesystem and application: *BusyBox 1.36.1*

That's possible to compile and assemble in less than 50 minutes!

Cross-compiling toolchain

What's a cross-compiling toolchain?



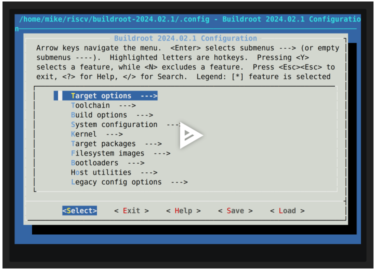
Why generate your own cross-compiling toolchain?

Compared to ready-made toolchains:

- You can choose your compiler version
- You can choose your C library (glibc, uClibc, musl)
- You can tweak many other features!
- You gain reproducibility: if a bug is found, just apply a fix. Don't need to get another toolchain (different bugs)

Generating a RISC-V musl toolchain with Buildroot

- Download Buildroot 2024.02.1 from https://buildroot.org
- Extract the sources (`tar xf`)
- Run `make menuconfig`
- In `Target options` →`Target Architecture`, choose `RISCV`
- In `Toolchain` →`C library`, choose `musl`.
- Save your configuration and run:
  `make sdk`
- At the end, you have a toolchain archive in
  `output/images/riscv64-buildroot-linux-musl_sdk-buildroot.tar.gz`
- Extract the archive in a suitable directory, and in the extracted directory, run: `./relocate-sdk.sh`



https://asciinema.org/a/655846

The RISC-V CPU architecture

RISC-V: a new open-source ISA

- ISA: *Instruction Set Architecture*
- Created by the University of California Berkeley, in a world dominated by proprietary ISAs with heavy royalties (ARM, x86)
- Exists in 32, 64 and 128 bit variants, from microcontrollers to powerful server hardware.
- Anyone can use and extend it to create their own SoCs and CPUs.
- This reduces costs and promotes reuse and collaboration
- Implementations can be proprietary. Many hardware vendors are using RISC-V CPUs in their hardware (examples: Microchip, Western Digital, Nvidia...)
- Free implementations are also available

Shakti Open Source Processor Development Ecosystem (BSD license)

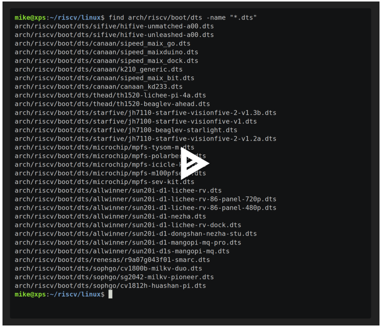
See https://en.wikipedia.org/wiki/RISC-V

RISC-V boards supported by Linux

How to find out with boards are supported by mainline Linux?

- In the Linux kernel sources, run:
  `find arch/riscv/boot/dts -name "*.dts"`
- You can also synthetize RISC-V cores on FPGAs
- You can also get started with the QEMU emulator, which simulates a virtual board with *virtio* hardware

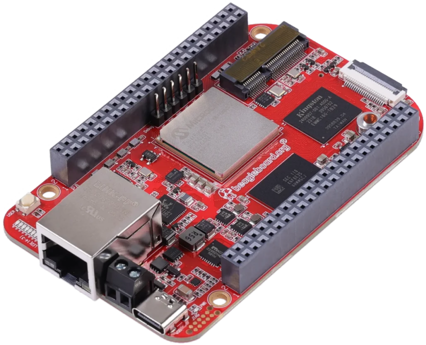
Already try it with JSLinux: https://bellard.org/jslinux/



https://asciinema.org/a/655447

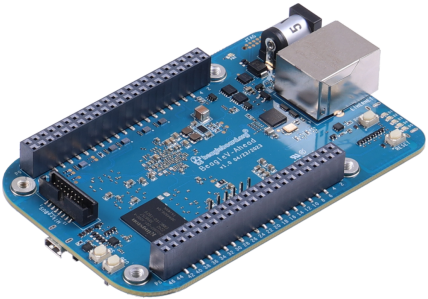
Beagleboard.org RISC-V boards

Open Hardware and community friendly boards



BeagleV-Fire

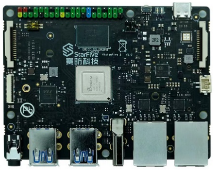
https://www.beagleboard.org/boards/beaglev-fire
Microchip Polarfire MPFS025T SoC FPGA, 150 USD.



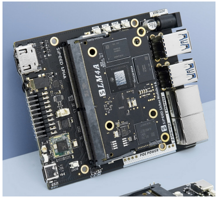
BeagleV-Ahead

https://www.beagleboard.org/boards/beaglev-ahead
Alibaba T-Head TH1520 SoC, 150 USD.

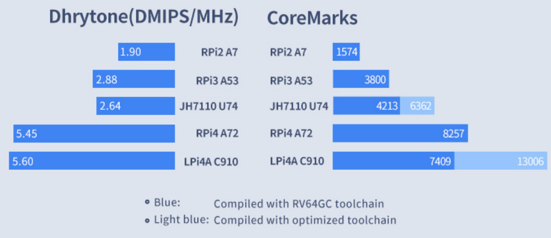
Other community friendly RISC-V boards



VisionFive2 by StarFive, StarFive JH7110 quad-core CPU with IMG BXE4-32 GPU, 40 pin Raspberry PI compatible header, 130 USD (8 GB version). Good upstream support.



LicheePi 4A from Sipeed. Like BeagleV-Ahead, Alibaba T-Head TH1520 SoC. Supported through a community only effort. 180 USD.



https://wiki.sipeed.com/hardware/en/lichee/th1520/lpi4a/1_intro.html

StarFive JH7110: VisionFive2
Alibaba T-Head TH1520: BeagleV-Ahead, LicheePi 4A

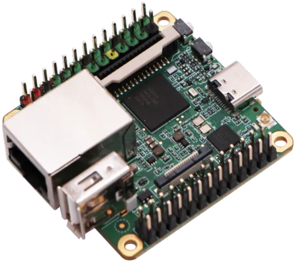
Thanks to Drew Fustini for the selection!

Inexpensive Milk-V boards

https://milkv.io/duo

Milk-V Duo:
Cvitech CV1800B C906@1GHz + C906@700MHz CPU
64 MB RAM, 5 USD

Milk-V Duo 256M:
Sophgo SG2002 C906@1GHz + C906@700MHz, 1xCortex-A53 @ 1GHz
256 MB RAM, 8 USD



https://milkv.io/duo-s
Sophgo SG2000 C906@1GHz + C906@700MHz, 1xCortex-A53 @ 1GHz
512 MB RAM, 10 USD

Products targeting camera applications
Caution: 1 core for Linux, 1 core for RTOS
Preliminary support in upstream kernel
Thanks to Thomas Bonnefille for the recommendation!

Back to the cross-compiling toolchain

Testing the toolchain

- Create a new `riscv64-env.sh` file you can source to set environment variables for your project:

```
export PATH=$HOME/toolchain/riscv64-buildroot-linux-musl_sdk-buildroot/bin:$PATH
```

- Run `source riscv64-env.sh`, take a `hello.c` file and test your new compiler:

```
$ riscv64-linux-gcc -static -o hello hello.c
$ file hello
hello: ELF 64-bit LSB executable, UCB RISC-V, double-float ABI, version 1 (SYSV), statically linked,
not stripped
```

  We are compiling statically so far to avoid having to deal with shared libraries.

- Test your executable with QEMU in user mode:

```
$ qemu-riscv64 hello
Hello world!
```

Hardware emulator

Finding which machines are emulated by QEMU

Tests made with QEMU 6.2.0 (Ubuntu 22.04)

```
sudo apt install qemu-system-misc
$ qemu-system-riscv64 -M ?
Supported machines are:
none                 empty machine
shakti_c             RISC-V Board compatible with Shakti SDK
sifive_e             RISC-V Board compatible with SiFive E SDK
sifive_u             RISC-V Board compatible with SiFive U SDK
spike                RISC-V Spike board (default)
virt                 RISC-V VirtIO board
```

We are going to use the `virt` one, emulating VirtIO peripherals (more efficient than emulating real hardware).

Booting process and privileges

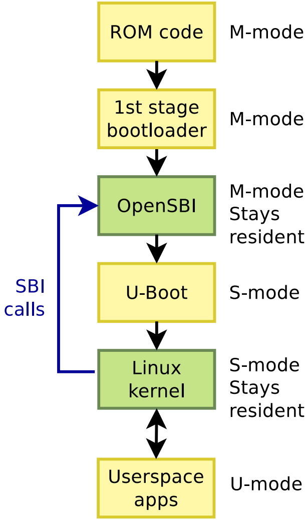
RISC-V privilege modes

RISC-V has three privilege modes:

- **U**ser (U-Mode): applications
- **S**upervisor (S-Mode): OS kernel
- **M**achine (M-Mode): bootloader and firmware

Here are typical combinations:

- **M**: simple embedded systems
- **M**, **U**: embedded systems with memory protection
- **M**, **S**, **U**: UNIX-style operating systems with virtual memory



Firmware

OpenSBI: Open Supervisor Binary Interface

- Required to start an OS (S mode) from the Supervisor/Firmware (M mode)
- Would be the first thing to build.
- However, OpenSBI 0.9 is already integrated in `qemu-system-riscv64` and I got issues replacing it. Let's keep this one. It's like a BIOS.

U-Boot bootloader

Environment for U-Boot cross-compiling

- Clone the U-Boot Git tree (go to https://u-boot.org)

```
git clone https://github.com/u-boot/u-boot
cd u-boot
git tag | grep 2024.04
git checkout v2024.04
```

- Let's add an environment variable to our `riscv64-env.sh` file for cross-compiling:

```
export CROSS_COMPILE=riscv64-linux-
```

- `CROSS_COMPILE` is the cross-compiler prefix, as our cross-compiler is `riscv64-linux-gcc`.

Cross-compiling U-Boot

- Find U-Boot ready-made configurations for RISC-V:

```
ls configs | grep riscv
```

- We will choose the configuration for QEMU and U-Boot running in S Mode:

```
make qemu-riscv64_smode_defconfig
```

- Now let's compile U-Boot (`-j20`: 20 compile jobs in parallel)

```
make -j20
```

- Result: `u-boot.bin` (859376 bytes!).
  We could make it much smaller by removing many options!

Starting U-Boot in QEMU

```
qemu-system-riscv64 -m 2G \
    -nographic \
    -machine virt \
    -smp 8 \
    -kernel u-boot/u-boot.bin
```

- `-m`: amount of RAM in the emulated machine
- `-smp`: number of CPUs in the emulated machine

Exit QEMU with `[Ctrl][a]` followed by `[x]`

```
Boot HART ID              : 1
Boot HART Domain          : root
Boot HART ISA             : rv64imafdcsu
Boot HART Features        : scounteren,mcounteren,time
Boot HART PMP Count       : 16
Boot HART PMP Granularity : 4
Boot HART PMP Address Bits: 54
Boot HART MHPM Count      : 0
Boot HART MHPM Count      : 0
Boot HART MIDELEG         : 0x0000000000000222
Boot HART MEDELEG         : 0x000000000000b109


U-Boot 2024.04 (Apr 22 2024 - 22:36:43 +0200)

CPU:   rv64imafdcsu
Model: riscv-virtio,qemu
DRAM:  2 GiB
Core:  32 devices, 13 uclasses, devicetree: board
Flash: 32 MiB
Loading Environment from nowhere... OK
In:    serial,usbkbd
Out:   serial,vidconsole
Err:   serial,vidconsole
No working controllers found
Net:   No ethernet found.
Working FDT set to fef031d0
Hit any key to stop autoboot:  0

Device 0: unknown device

Device 1: unknown device
scanning bus for devices...

Device 0: unknown device
starting USB...
No working controllers found
No ethernet found.
No ethernet found.
```

Linux kernel

Environment for kernel cross-compiling

- Download the latest Linux 6.8.x sources from https://kernel.org
- Extract the sources: `tar xf linux-6.8.<x>.tar.xz`
- Let's rename the source directory to make our instructions version independent: `mv linux-6.8.<x> linux`
- Go to the Linux source directory: `cd linux`
- Let's add two environment variables for kernel cross-compiling to our `riscv64-env.sh` file:

```
export CROSS_COMPILE=riscv64-linux-
export ARCH=riscv
```

- `ARCH` is the name of the subdirectory in arch/ corresponding to the target architecture.

Kernel configuration

- Lets take the default Linux kernel configuration for RISC-V:

```
$ make help | grep defconfig
  defconfig       - New config with default from ARCH supplied defconfig
  savedefconfig   - Save current config as ./defconfig (minimal config)
  alldefconfig    - New config with all symbols set to default
  olddefconfig    - Same as oldconfig but sets new symbols to their
  nommu_k210_defconfig        - Build for nommu_k210
  nommu_k210_sdcard_defconfig - Build for nommu_k210_sdcard
  nommu_virt_defconfig        - Build for nommu_virt
$ make defconfig
```

- We can now further customize the configuration:

```
make menuconfig
```

Compiling the kernel

```
make -j 20
```

At the end, you have these files:

`vmlinux`: raw kernel in ELF format (not bootable, for debugging)

`arch/riscv/boot/Image`: uncompressed bootable kernel

`arch/riscv/boot/Image.gz`: compressed kernel

Booting the kernel

Booting the Linux kernel directly

We could boot the Linux kernel directly as follows

```
qemu-system-riscv64 -m 2G \
    -nographic \
    -machine virt \
    -smp 8 \
    -kernel linux/arch/riscv/boot/Image \
    -append "console=ttyS0" \
```

However, what we want to demonstrate is the normal booting process:
OpenSBI $\rightarrow$U-Boot $\rightarrow$Linux $\rightarrow$Userspace

Booting the Linux kernel from U-Boot

- We want to show how to set the U-Boot environment to load the Linux kernel and to specify the Linux kernel command line
- For this purpose, we will need some storage space to store the U-Boot environment, load the kernel binary, and also to contain the filesystem that Linux will boot on.
- Therefore, let's create a disk image to give some storage space for QEMU

Disk image creation (1)

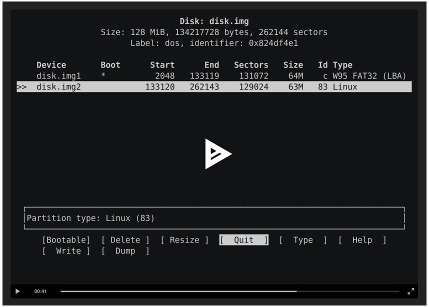
- Let's create a 128 MB disk image:

```
dd if=/dev/zero of=disk.img bs=1M count=128
```

- Let's create two partitions in this image

```
cfdisk disk.img
```

  - A first 64 MB primary partition (type `W95 FAT32 (LBA)`), marked as bootable
  - A second partition with remaining space (default type: `Linux`)
- Fun note: no need to be `root` here!



https://asciinema.org/a/656814

Disk image creation (2)

- Let's access the partitions in this disk image:

```
sudo losetup -f --show --partscan disk.img
/dev/loop31

ls -la /dev/loop31*
brw-rw---- 1 root disk   7,  2 Jan 14 10:50 /dev/loop31
brw-rw---- 1 root disk 259, 11 Jan 14 10:50 /dev/loop31p1
brw-rw---- 1 root disk 259, 12 Jan 14 10:50 /dev/loop31p2
```

- We can now format the partitions:

```
sudo mkfs.vfat -F 32 -n boot /dev/loop31p1
sudo mkfs.ext4 -L rootfs /dev/loop31p2
```

Copying the Linux image to the FAT partition

- Let's create a mount point for the FAT partition:

```
mkdir /mnt/boot
```

- Let's mount it:

```
sudo mount /dev/loop31p1 /mnt/boot
```

- Let's copy the kernel image to it:

```
sudo cp linux/arch/riscv/boot/Image /mnt/boot
```

- And then unmount the filesystem to commit changes:

```
sudo umount /mnt/boot
```

Recompiling U-Boot for environment support

We want U-Boot be able to use an environment stored in the FAT partition we created. This way we can customize U-Boot's behaviour!

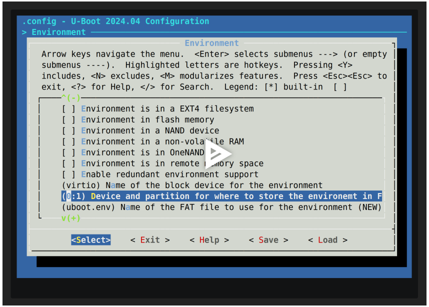
- So, let's reconfigure U-Boot:

```
make menuconfig
```

  - CONFIG_ENV_IS_IN_FAT=y
  - CONFIG_ENV_FAT_INTERFACE="virtio"
  - CONFIG_ENV_FAT_DEVICE_AND_PART="0:1"

- Then recompile U-Boot

```
make -j20
```



https://asciinema.org/a/656816

Run U-Boot with an environment

- Add a disk to the emulated machine:

```
qemu-system-riscv64 -m 2G -nographic -machine virt -smp 8 \
    -kernel u-boot/u-boot.bin \
    -drive file=disk.img,format=raw,id=hd0 \
    -device virtio-blk-device,drive=hd0
```

- In U-Boot, you should now be able to save an environment:

```
=> setenv foo bar
=> saveenv
=> reset
...
=> printenv foo
bar
```

Booting Linux from U-Boot

Requirements for booting Linux

To boot the Linux kernel, U-Boot needs to load

- A Linux kernel image. In our case, let's load it from our virtio disk to RAM (find a suitable RAM address by using the `bdinfo` command in U-Boot):

```
fatload virtio 0:1 84000000 Image
```

- A *Device Tree Binary (DTB)*, letting the kernel know which SoC and devices we have. This allows the same kernel to support many different SoCs and boards.
  - *DTB* files are compiled from *DTS* files in `arch/riscv/boot/dts/`
  - However, there is no such *DTS* file for the RISC-V QEMU virt board.
  - The *DTB* for our board is actually passed by QEMU to OpenSBI and then to U-Boot.
  - In U-Boot, at least in our case, the *DTB* is available in RAM at address `${fdtcontroladdr}`

Linux kernel command line

In U-Boot, we need to set the Linux arguments (*kernel command line*)

```
setenv bootargs root=/dev/vda2 console=ttyS0 earlycon=sbi rw
```

- `root=/dev/vda2`
  Device for Linux to mount as root filesystem
- `console=ttyS0`
  Device (here first serial line) to send Linux booting messages to
- `earlycon=sbi`
  Allows to see messages before the console driver is initialized (*Early Console*).
- `rw`
  Allows to mount the root filesystem in read-write mode.

Booting Linux

- Here's the command to boot the Linux `Image` file:

```
booti <Linux address> - <DTB address>
```

- In our case:

```
booti 0x84000000 - ${fdtcontroladdr}
```

- So, let's define the default series of commands that U-Boot will automatically run:

```
setenv bootcmd 'fatload virtio 0:1 84000000 Image; booti 0x84000000 - ${fdtcontroladdr}'
```

- Save these new settings:

```
saveenv
```

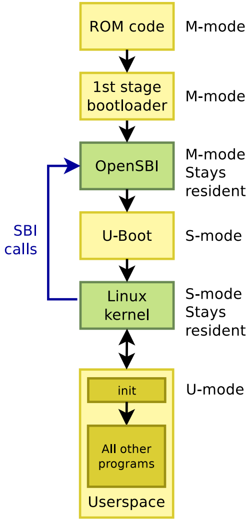
- And boot our system (`boot` runs `bootcmd`):

```
boot
```

Booting Linux... almost there

```
[    0.581124] NET: Registered PF_INET6 protocol family
[    0.588486] Segment Routing with IPv6
[    0.588698] In-situ OAM (IOAM) with IPv6
[    0.589026] sit: IPv6, IPv4 and MPLS over IPv4 tunneling driver
[    0.591357] NET: Registered PF_PACKET protocol family
[    0.595137] 9pnet: Installing 9P2000 support
[    0.595619] Key type dns_resolver registered
[    0.615441] debug_vm_pgtable: [debug_vm_pgtable         ]: Validating architecture page...
[    0.624764] Legacy PMU implementation is available
[    0.625600] clk: Disabling unused clocks
[    0.625790] ALSA device list:
[    0.625868]   No soundcards found.
[    0.672876] EXT4-fs (vda2): recovery complete
[    0.673455] EXT4-fs (vda2): mounted filesystem 2c5da046-5bee-4760-8d16-573bb8d1c176 r/w...
[    0.673876] VFS: Mounted root (ext4 filesystem) on device 254:2.
[    0.675577] devtmpfs: error mounting -2
[    0.698640] Freeing unused kernel image (initmem) memory: 2240K
[    0.700178] Run /sbin/init as init process
[    0.700602] Run /etc/init as init process
[    0.700710] Run /bin/init as init process
[    0.700812] Run /bin/sh as init process
[    0.700985] Kernel panic - not syncing: No working init found.  Try passing init= option...
```

Linux booted, mounted the root filesystem, but failed to find an *init* program to run. Let's add one!



Building the root filesystem

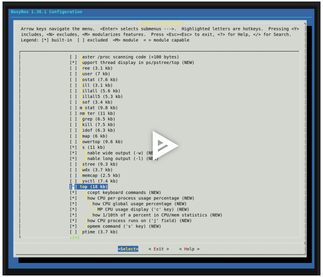
BusyBox - Most Linux commands in one binary

```
[, [[, acpid, add-shell, addgroup, adduser, adjtimex, arch, arp, arping, ash, awk, base64, basename, bc, beep, blkdiscard, blkid,
blockdev, bootchartd, brctl, bunzip2, bzcat, bzip2, cal, cat, chat, chattr, chgrp, chmod, chown, chpasswd, chpst, chroot, chrt,
chvt, cksum, clear, cmp, comm, conspy, cp, cpio, crond, crontab, cryptpw, cttyhack, cut, date, dc, dd, deallocvt, delgroup,
deluser, depmod, devmem, df, dhcprelay, diff, dirname, dmesg, dnsd, dnsdomainname, dos2unix, dpkg, dpkg-deb, du, dumpkmap,
dumpleases, echo, ed, egrep, eject, env, envdir, envuidgid, ether-wake, expand, expr, factor, fakeidentd, fallocate, false,
fatattr, fbset, fbsplash, fdflush, fdformat, fdisk, fgconsole, fgrep, find, findfs, flock, fold, free, freeramdisk, fsck,
fsck.minix, fsfreeze, fstrim, fsync, ftpd, ftpget, ftpput, fuser, getopt, getty, grep, groups, gunzip, gzip, halt, hd, hdparm,
head, hexdump, hexedit, hostid, hostname, httpd, hush, hwclock, i2cdetect, i2cdump, i2cget, i2cset, i2ctransfer, id, ifconfig,
ifdown, ifenslave, ifplugd, ifup, inetd, init, insmod, install, ionice, iostat, ip, ipaddr, ipcalc, ipcrm, ipcs, iplink, ipneigh,
iproute, iprule, iptunnel, kbd_mode, kill, killall, killall5, klogd, last, less, link, linux32, linux64, linuxrc, ln, loadfont,
loadkmap, logger, login, logname, logread, losetup, lpd, lpq, lpr, ls, lsattr, lsmod, lsof, lspci, lsscsi, lsusb, lzcat, lzma,
lzop, makedevs, makemime, man, md5sum, mdev, mesg, microcom, mim, mkdir, mkdosfs, mke2fs, mkfifo, mkfs.ext2, mkfs.minix, mkfs.vfat,
mknod, mkpasswd, mkswap, mktemp, modinfo, modprobe, more, mount, mountpoint, mpstat, mt, mv, nameif, nanddump, nandwrite,
nbd-client, nc, netstat, nice, nl, nmeter, nohup, nologin, nproc, nsenter, nslookup, ntpd, nuke, od, openvt, partprobe, passwd,
paste, patch, pgrep, pidof, ping, ping6, pipe_progress, pivot_root, pkill, pmap, popmaildir, poweroff, powertop, printenv, printf,
ps, pscan, pstree, pwd, pwdx, raidautorun, rdate, rdev, readahead, readlink, readprofile, realpath, reboot, reformime,
remove-shell, renice, reset, resize, resume, rev, rm, rmdir, rmmod, route, rpm, rpm2cpio, rtcwake, run-init, run-parts, runlevel,
runsv, runsvdir, rx, script, scriptreplay, sed, sendmail, seq, setarch, setconsole, setfattr, setfont, setkeycodes, setlogcons,
setpriv, setserial, setsid, setuidgid, sh, sha1sum, sha256sum, sha3sum, sha512sum, showkey, shred, shuf, slattach, sleep, smemcap,
softlimit, sort, split, ssl_client, start-stop-daemon, stat, strings, stty, su, sulogin, sum, sv, svc, svlogd, svok, swapoff,
swapon, switch_root, sync, sysctl, syslogd, tac, tail, tar, taskset, tc, tcpsvd, tee, telnet, telnetd, test, tftp, tftpd, time,
timeout, top, touch, tr, traceroute, traceroute6, true, truncate, ts, tty, ttysize, tunctl, ubiattach, ubidetach, ubimkvol,
ubirename, ubirmvol, ubirsvol, ubiupdatevol, udhcpc, udhcpc6, udhcpd, udpsvd, uevent, umount, uname, unexpand, uniq, unix2dos,
unlink, unlzma, unshare, unxz, unzip, uptime, users, usleep, uudecode, uuencode, vconfig, vi, vlock, volname, w, wall, watch,
watchdog, wc, wget, which, who, whoami, whois, xargs, xxd, xz, xzcat, yes, zcat, zcip
```

Source: run `/bin/busybox` - July 2021 status

BusyBox - Downloading and configuring

- Download BusyBox 1.36.1 sources from https://busybox.net
- Extract the archive with `tar xf`
- Run `make allnoconfig`
  Starts with no applet selected
- Run `make menuconfig`
  - In `Settings` →`Build Options`, enable `Build static binary (no shared libs)`
  - In `Settings` →`Build Options`, set `Cross compiler prefix` to `riscv64-linux-`
  - In `Settings` →`Library Tuning`, enable `Command line editing` and `Tab completion`.
  - Then enable support for the following commands:
    `hush`, `init`, `reboot`, `mount`, `cat`, `chmod`, `echo`, `ls`, `mkdir`, `ps`, `top`, `uptime`, `vi`, `httpd`, `ifconfig`



https://asciinema.org/a/656959

BusyBox - Installing and compiling

- Compiling: `make -j 20`
  Resulting size: only 460,840 bytes! (could be 300,000 with fewer features)
  Funny to see that we're using a 64 bit system to run such small programs!
- Installing in `_install/`: `make install`
- See the created directory structure and the symbolic links to `/bin/busybox`
- Installing to the root filesystem:

```
sudo mkdir /mnt/rootfs
sudo mount /dev/loop31p2 /mnt/rootfs
sudo rsync -aH _install/ /mnt/rootfs/
```

```
├── bin
│   ├── busybox
│   ├── cat -> busybox
│   ├── chmod -> busybox
│   ├── echo -> busybox
│   ├── hush -> busybox
│   ├── ls -> busybox
│   ├── mkdir -> busybox
│   ├── mount -> busybox
│   ├── ps -> busybox
│   ├── sh -> busybox
│   └── vi -> busybox
├── sbin
│   ├── ifconfig -> ../bin/busybox
│   ├── init -> ../bin/busybox
│   └── reboot -> ../bin/busybox
└── usr
    ├── bin
    │   ├── top -> ../../bin/busybox
    │   └── uptime -> ../../bin/busybox
    └── sbin
        └── httpd -> ../../bin/busybox
```

Completing the root filesystem (1)

We also need to create a `dev` directory for device files. The kernel will automatically mount the `devtmpfs` filesystem there (as `CONFIG_DEVTMPFS_MOUNT=y`)

```
sudo mkdir /mnt/rootfs/dev
sudo umount /mnt/rootfs
```

The system should have everything it needs to boot now:

```
[    0.463042] VFS: Mounted root (ext4 filesystem) on device 254:2.
[    0.464862] devtmpfs: mounted
[    0.486872] Freeing unused kernel image (initmem) memory: 2240K
[    0.488446] Run /sbin/init as init process
starting pid 87, tty '': '/etc/init.d/rcS'
can't run '/etc/init.d/rcS': No such file or directory

Please press Enter to activate this console.
starting pid 89, tty '': '-/bin/sh'

BusyBox v1.36.1 (2024-04-29 07:21:47 CEST) built-in shell (ash)
#
```

Completing the root filesystem (2)

Let's try to run the ps command to see the list of processes:

```
# ps
  PID USER       VSZ STAT COMMAND
ps: can't open '/proc': No such file or directory
```

We need to create /proc and /sys so that we can mount the proc and sysfs virtual filesystems on the target, which are needed by many system commands. We can now run the commands **on the target system**:

```
mkdir /proc
mkdir /sys
mount -t proc nodev /proc
mount -t sysfs nodev /sys
```

Completing the root filesystem (3)

Let's automate the mounting of proc and sysfs...

- Let's create an /etc/inittab file to configure Busybox Init:

```
# This is run first script:
::sysinit:/etc/init.d/rcS
# Start an "askfirst" shell on the console:
::askfirst:/bin/sh
```

- Let's create and fill /etc/init.d/rcS to automatically mount the virtual filesystems:

```
#!/bin/sh
mount -t proc nodev /proc
mount -t sysfs nodev /sys
```

Common mistakes

- Don't forget to make the `rcS` script executable. Linux won't allow to execute it otherwise.
- Do not forget `#!/bin/sh` at the beginning of shell scripts! Without the leading `#!` characters, the Linux kernel has no way to know it is a shell script and will try to execute it as a binary file!
- Don't forget to specify the execution of a shell in `/etc/inittab` or at the end of `/etc/init.d/rcS`. Otherwise, execution will just stop without letting you type new commands!

Add support for networking (1)

- Add a network interface to the emulated machine:

```
sudo qemu-system-riscv64 -m 2G -nographic -machine virt -smp 8 \
    -kernel u-boot/u-boot.bin \
    -drive file=disk.img,format=raw,id=hd0 \
    -device virtio-blk-device,drive=hd0 \
    -netdev tap,id=tapnet,ifname=tap2,script=no,downscript=no \
    -device virtio-net-device,netdev=tapnet
```

- Need to be `root` to bring up the `tap2` network interface

Add support for networking (2)

- On the target machine:

```
ifconfig -a
ifconfig eth0 192.168.2.100
```

- On the host machine:

```
ifconfig -a
sudo ifconfig tap2 192.168.2.1
ping 192.168.2.100
```

Simple CGI script

```
#!/bin/sh
echo "Content-type: text/html"
echo
echo "<html>"
echo "<meta http-equiv=\"refresh\" content=\"1\">"
echo "<header></header><body>"
echo "<h1>Uptime information</h1>"
echo "Your embedded device has been running for:<pre><font color=Blue>"
echo `uptime`
echo "</font></pre>"
echo "</body></html>"
```

Store it in `/www/cgi-bin/uptime` and make it executable.

Start a web server

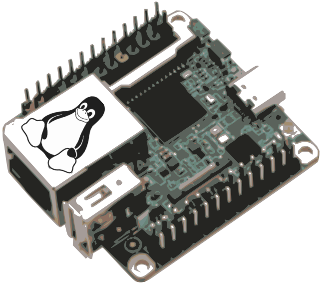
- On the target machine:

```
/usr/sbin/httpd -h /www
```

- On the host machine, open in your browser:
  http://192.168.2.100/cgi-bin/uptime

Demo: booting Linux on Milk-V Duo S board

- We can use the same binary kernel!
- A Linux kernel can be built for many different SoCs at the same time.
- All we need is just a different description of the hardware (DTB)
- However, support for this board and its SoC is pretty basic in the mainline kernel so far:
  - The MMC driver not fully ready yet (patches submitted for the 6.9 kernel)
  - We will therefore boot on a filesystem in RAM (*Initramfs*), included in the kernel binary.



Connecting the Milk-V Duo S board

Let's use the new `tio` command to access the serial line:

- `tio` doesn't die but waits when the line is disconnected
- `tio` can also log to a file
- `tio` is easy to use:
  `tio /dev/ttyUSB0`

SD card for the Milk-V Duo S board

- Format your micro-SD card as previously:

```
sudo cfdisk /dev/mmcblk0
```

- Mount the boot partition
- We are not ready to use a mainline U-Boot yet, so copy the `fip.bin` file from https://gitlab.com/michaelopdenacker/embedded-linux-from-scratch-riscv/-/raw/main/binaries/milk-v/duo-s/fip.bin to the boot partition.
- Copy the same `Image` file to the boot partition:

```
cp arch/riscv/boot/Image /mnt/boot/
```

- Also copy a DTB from a very similar board:

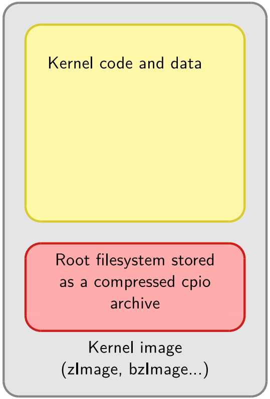
```
cp arch/riscv/boot/dts/sophgo/cv1812h-huashan-pi.dtb /mnt/boot/
```

Boot the Milk-V Duo S board

- Insert the micro-SD card, power the board, and in the U-Boot prompt, type:

```
setenv bootargs console=ttyS0,115200; fatload mmc 0 82000000 Image;
fatload mmc 0 84000000 cv1812h-huashan-pi.dtb; booti 82000000 - 84000000
```

- However, it won't boot because we haven't given it a root filesystem yet.
- So, let's prepare an *Initramfs* to boot on, and include it into the kernel binary.



Initramfs for RISC-V (1)

Something that should run on any RISC-V board!

- Mount your root filesystem image again and copy it to a directory

```
sudo mount /dev/loop31p2 /mnt/rootfs
sudo rsync -aH /mnt/rootfs ~/riscv/rootfs
```

- Linux will try to start `/init` in the *initramfs*

```
cd ~/riscv/rootfs
ln -s sbin/init .
```

- You also need to mount the `devtmpfs` filesystem manually by adding this line to `etc/init.d/rcS`:

```
mount -t devtmpfs nodev /dev
```

Initramfs for RISC-V (2)

- Unlike on ARM, you also need a `/dev/console` file in an *Initramfs* before mounting `/dev/`:

```
sudo mknod dev/console c 5 1
```

- Now, configure Linux to bundle this new directory as *Initramfs*. In `General setup`, set `Initramfs source file(s)` to `../rootfs`.
- Recompile Linux and update the `Image` file on the boot partition.
- Voilà!

```
[    1.106406] mousedev: PS/2 mouse device common for all mice
[    1.114012] sdhci: Secure Digital Host Controller Interface driver
[    1.120482] sdhci: Copyright(c) Pierre Ossman
[    1.125252] Synopsys Designware Multimedia Card Interface Driver
[    1.131783] sdhci-pltfm: SDHCI platform and OF driver helper
[    1.138127] usbcore: registered new interface driver usbhid
[    1.143955] usbhid: USB HID core driver
[    1.149969] NET: Registered PFOINET6 protocol family
[    1.157399] Segment Routing with IPv6
[    1.161377] In-situ OAM (IOAM) with IPv6
[    1.165618] sit: IPv6, IPv4 and MPLS over IPv4 tunneling driver
[    1.173106] NET: Registered PF_PACKET protocol family
[    1.178579] 9pnet: Installing 9P2000 support
[    1.183219] Key type dns_resolver registered
[    1.229810] debug_vm_pgtable: [debug_vm_pgtable         ]: Validat...
[    1.241117] Legacy PMU implementation is available
[    1.246652] clk: Disabling unused clocks
[    1.250832] ALSA device list:
[    1.253991]   No soundcards found.
[    1.259366] dw-apb-uart 4140000.serial: forbid DMA for kernel console
[    1.268035] Freeing unused kernel image (initmem) memory: 2476K
[    1.274290] Run /init as init process

Please press Enter to activate this console.

BusyBox v1.36.1 (2024-04-30 06:50:07 CEST) built-in shell (ash)

~ #
```

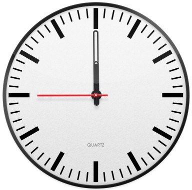
A few things to remember

- Embedded Linux is just made out of simple components.
  It makes it easier to get started with Linux.
- You just need a toolchain, a bootloader, a kernel and a few executables.
- RISC-V is a new, open Instruction Set Architecture, use it and support it!
- With Asciinema, you can copy text from videos!
- You will love `tio` as a replacement to `picocom`.

Going further and thanks

अब आप गुरू हैं

- Drew Fustini's unmatched presentation about Linux on RISC-V: https://tinyurl.com/elc2023-bof
- Bootlin's training materials and conference presentations (Creative Commons CC-BY-SA licence): https://bootlin.com/docs/
- Thanks to Drew Fustini for sharing his personal advice.
- Thanks to YOU for attending this talk!



See credits for all images

Questions? Suggestions? Comments?

Michael Opdenacker
michael.opdenacker@bootlin.com

https://bootlin.com/pub/conferences/2024/risc-v/