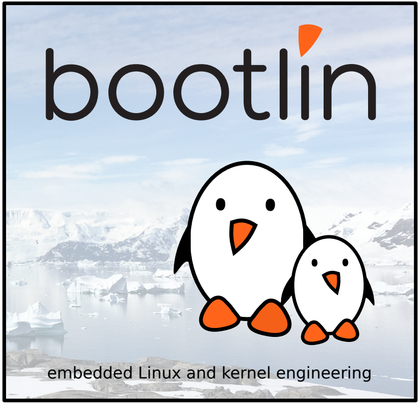
Embedded Linux from scratch in 45 minutes (on RISC-V)

Michael Opdenacker
michael.opdenacker@bootlin.com

Corrections, suggestions, contributions and translations are welcome!

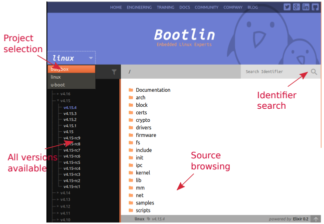



EMBEDDED LINVX FROM SCRATCH IN XLV MINVTES ON RISC-V


EMBEDDED
LIVE
EVENT
MMXX

Michael Opdenacker

- Founder and Embedded Linux engineer at Bootlin:
  - Embedded Linux **expertise**
  - **Development**, consulting and training
  - Focusing **only on Free and Open Source Software**
- Free Software contributor:
  - Current maintainer of the Elixir Cross Referencer, making it easier to study the sources of big C projects like the Linux kernel. See https://elixir.bootlin.com
  - Co-author of Bootlin's freely available embedded Linux and kernel training materials (https://bootlin.com/docs/)
  - Former maintainer of GNU Typist



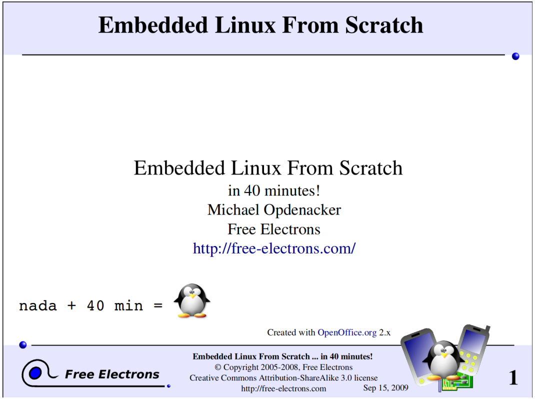
Introduction

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 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!
- 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...).

Reviving an old presentation

- First shown in 2005 at the Libre Software Meeting in Dijon, France.
- Showing a 2.6 Linux kernel booting on a QEMU emulated ARM board.
- One of our most downloaded presentations at that time.



Things that changed since 2005

In the embedded environment

- The Maker movement
- Cheap development boards
  500+ EUR →50-100 EUR
- The rise of Open Hardware
  (Arduino, Beaglebone Black...)
- *RISC-V*: a new open-source hardware instruction set architecture

In the Linux kernel:

- Linux 2.6.x →5.x
- `tar` →`git`
- Linux is now everywhere, no need to convince customers to use it. It's even easier and easier to convince then to fund contributions to the official version.
- *devtmpfs*: automatically creates device files
- ARM and other architectures: devices described by the *Device Tree* instead of C code

And many more!

RISC-V: a new open-source Instruction Set Architecture (ISA)

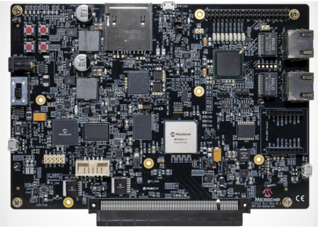
- Created by the University of California Berkeley, in a world dominated by proprietary ISAs with heavy royalties (ARM, x86)
- Exists in 32, 64 and 128 bit variants, from microcontrollers to powerful server hardware.
- Anyone can use and extend it to create their own SoCs and CPUs.
- This reduces costs 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 being created

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

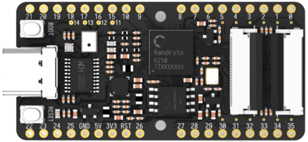
How to use RISC-V with Linux?

Hardware is now getting available

- Icicle kit: with Microchip's PolarFire SoC and an FPGA with 254 K gates. Sold at 499 USD at CrowdSupply: `https://frama.link/dK1oanrd`
- Boards with the Kendryte K210 SoC. Sipeed MAix BiT only costs 13 USD at Seed Studio: `https://frama.link/QhBdPjsm`. Supported by Linux 5.8 but limited, but its MMU is not supported by Linux.
- You can also synthetize RISC-V cores on programmable logic (FPGAs)
- Before more hardware is available next year, you can get started with the QEMU emulator, which simulates a virtual board with *virtio* hardware



PolarFire SoC Icicle kit from Microchip



Seed Studio Sipeed MAix BiT

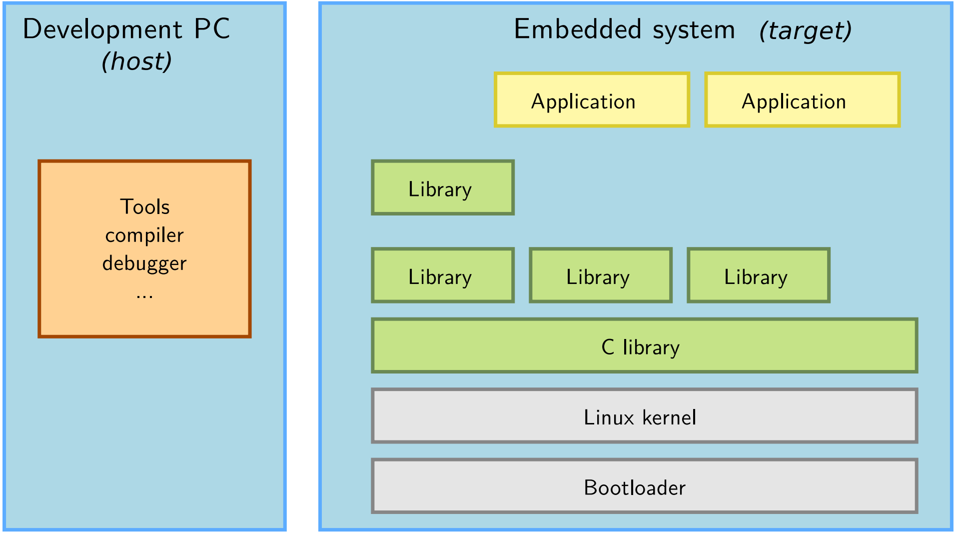
Already try it with JSLinux: `https://bellard.org/jslinux/`

Things to build today

- Cross-compiling toolchain: *Buildroot 2020.08*
- Firmware / Bootloader: *OpenSBI*
- Kernel: *Linux 5.10-rc6*
- Root filesystem and application: *BusyBox 1.32.0*

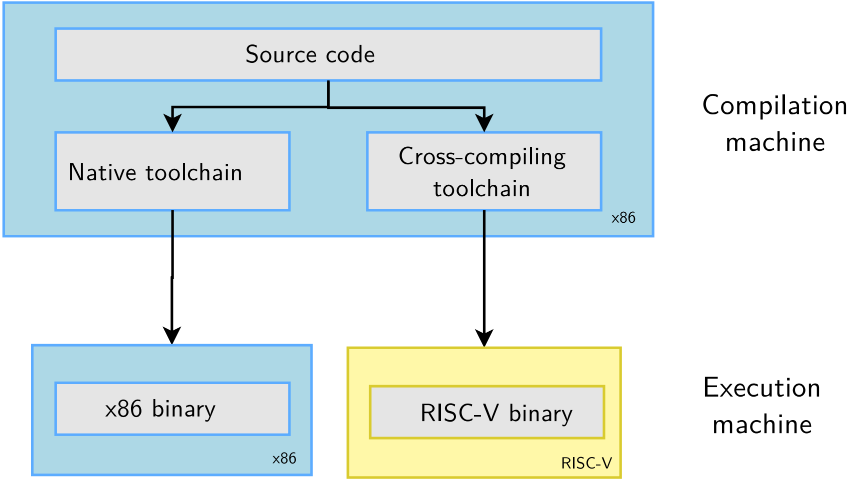
That's easy to compile and assemble in less than 45 minutes!

Embedded Linux - host and target



Cross-compiling toolchain

What's a cross-compiling toolchain?



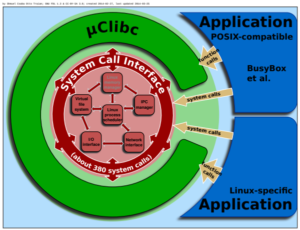
Why generate your own cross-compiling toolchain?

Compared to ready-made toolchains:

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

Choosing the C library

- The C library is an essential component of a Linux system
  - Interface between the applications and the kernel
  - Provides the well-known standard C API to ease application development
- Several C libraries are available:
  - *glibc*: full featured, but rather big (2 MB on ARM)
  - *uClibc*: better adapted to embedded use, smaller and supporting RISC-V 64. Not supported by Buildroot on this platform though.
  - *musl*: great for embedded use too, more recent
- The choice of the C library must be made at cross-compiling toolchain generation time, as the GCC compiler is compiled against a specific C library.

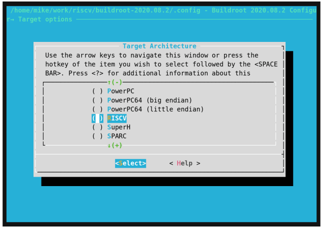


Source: Wikipedia

(http://bit.ly/2zrGve2)

Generating a RISC-V musl toolchain with Buildroot

- Download Buildroot 2020.08 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 an 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/375640

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, 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

Getting QEMU 5.0

I like to work with recent versions, but QEMU 4.x is probably good enough too!

- On Ubuntu 20.10:

```
sudo apt install qemu-system-misc
```

- On Ubuntu 20.04:

```
sudo add-apt-repository ppa:jacob/virtualisation
sudo apt-get update
sudo apt install qemu-system-misc
```

Finding which machines are emulated by QEMU

```
$ qemu-system-riscv64 -M ?
Supported machines are:
none                empty machine
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)
spike_v1.10         RISC-V Spike Board (Privileged ISA v1.10)
spike_v1.9.1        RISC-V Spike Board (Privileged ISA v1.9.1)
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

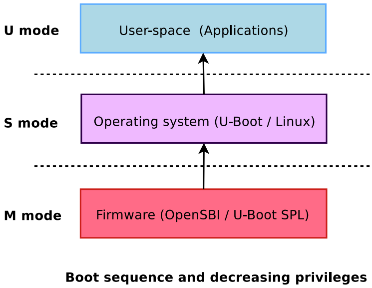
RISC-V privilege modes

RISC-V has three privilege modes:

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

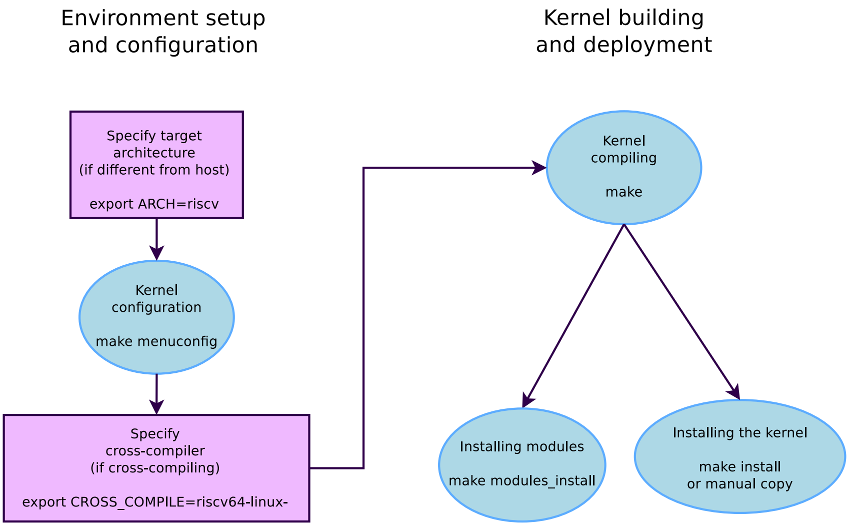
Here are typical combinations:

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



Linux kernel

Kernel building overview



Environment for kernel cross-compiling

- Download Linux 5.10-rc6 sources
- Let's add two environment variables for kernel cross-compiling to our `riscv64-env.sh` file:

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

- `CROSS_COMPILE` is the cross-compiler prefix, as our cross-compiler is `riscv64-linux-gcc`.
- `ARCH` is the name of the subdirectory in `arch/` corresponding to the target architecture.

Kernel configuration

- Lets take the default Linux kernel configuration for RISCV:

```
$ 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_virt_defconfig          - Build for nommu_virt
  rv32_defconfig                - Build for rv32
$ make defconfig
```

- We can now further customize the configuration:

```
make menuconfig
```

Compiling the kernel

```
make
```

To compile faster, run multiple **j**obs in parallel:

```
make -j 8
```

To **re**compile faster (7x according to some benchmarks), run multiple **j**obs in parallel:

```
make -j 8 CC="ccache riscv64-linux-gcc"
```

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

Firmware

OpenSBI: Open Supervisor Binary Interface

- Required to start an OS (S mode) from the Supervisor/Firmware (M mode)

```
git clone https://github.com/riscv/opensbi.git
cd opensbi
make PLATFORM=generic FW_PAYLOAD_PATH=../linux-5.10-rc6/arch/riscv/boot/Image
```

- Run the above command every time you update your kernel
- This generates the `build/platform/generic/firmware/fw_payload.elf` file which is a binary that QEMU can boot. QEMU cannot directly boot the binary kernel, unlike on ARM.

Booting the kernel

Booting the kernel with QEMU

```
qemu-system-riscv64 \
  -nographic \
  -machine virt \
  -m 128M \
  -kernel opensbi/build/platform/generic/firmware/fw_payload.elf \
  -append "console=ttyS0" \
```

- `-m`: amount of RAM in the emulated machine
- `-append`: kernel command line

The kernel starts to boot but eventually panics. We need a *root filesystem*!

```
[    0.491433] ---[ end Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(0,0) ]---
```

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

Building the root filesystem

BusyBox - Most commands in one binary - about 10 years ago

[, [[, acpid, addgroup, adduser, adjtimex, ar, arp, arping, ash, awk, basename, beep, blkid, brctl, bunzip2, bzcat, bzip2, cal, cat, catv, chat, chattr, chgrp, chmod, chown, chpasswd, chpst, chroot, chrt, chvt, cksum, clear, cmp, comm, cp, cpio, crond, crontab, cryptpw, cut, date, dc, dd, deallocvt, delgroup, deluser, depmod, devmem, df, dhcprelay, diff, dirname, dmesg, dnsd, dnsdomainname, dos2unix, dpkg, du, dumpkmap, dumpleases, echo, ed, egrep, eject, env, envdir, envuidgid, expand, expr, fakeidentd, false, fbset, fbsplash, fdflush, fdformat, fdisk, fgrep, find, findfs, flash_lock, flash_unlock, fold, free, freeramdisk, fsck, fsck.minix, fsync, ftpd, ftpget, ftpput, fuser, getopt, getty, grep, gunzip, gzip, hd, hdparm, head, hexdump, hostid, hostname, httpd, hush, hwclock, id, ifconfig, ifdown, ifenslave, ifplugd, ifup, inetd, init, inotifyd, insmod, install, ionice, ip, ipaddr, ipcalc, ipcrm, ipcs, iplink, iproute, iprule, iptunnel, kbd_mode, kill, killall, killall5, klogd, last, length, less, linux32, linux64, linuxrc, ln, loadfont, loadkmap, logger, login, logname, logread, losetup, lpd, lpq, lpr, ls, lsattr, lsmod, lzmacat, lzop, lzopcat, makemime, man, md5sum, mdev, mesg, microcom, mkdir, mkdosfs, mkfifo, mkfs.minix, mkfs.vfat, mknod, mkpasswd, mkswap, mktemp, modprobe, more, mount, mountpoint, mt, mv, nameif, nc, netstat, nice, nmeter, nohup, nslookup, od, openvt, passwd, patch, pgrep, pidof, ping, ping6, pipe_progress, pivot_root, pkill, popmaildir, printenv, printf, ps, pscan, pwd, raidautorun, rdate, rdev, readlink, readprofile, realpath, reformime, renice, reset, resize, rm, rmdir, rmmod, route, rpm, rpm2cpio, rtcwake, run-parts, runlevel, runsv, runsvdir, rx, script, scriptreplay, sed, sendmail, seq, setarch, setconsole, setfont, setkeycodes, setlogcons, setsid, setuidgid, sh, sha1sum, sha256sum, sha512sum, showkey, slattach, sleep, softlimit, sort, split, start-stop-daemon, stat, strings, stty, su, sulogin, sum, sv, svlogd, swapoff, swapon, switch_root, sync, sysctl, syslogd, tac, tail, tar, taskset, tcpsvd, tee, telnet, telnetd, test, tftp, tftpd, time, timeout, top, touch, tr, traceroute, true, tty, ttysize, udhcpc, udhcpd, udpsvd, umount, uname, uncompress, unexpand, uniq, unix2dos, unlzma, unlzop, unzip, uptime, usleep, uudecode, uuencode, vconfig, vi, vlock, volname, watch, watchdog, wc, wget, which, who, whoami, xargs, yes, zcat, zcip

Source: run `/bin/busybox`

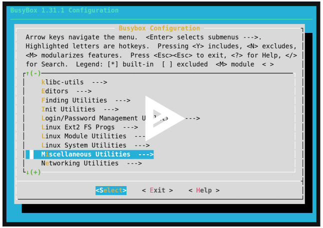
BusyBox - Most commands in one binary - In 2019

[, [[, acpid, add-shell, addgroup, adduser, adjtimex, ar, arch, arp, arping, awk, base64, basename, bbconfig, bc, beep, blkdiscard, blkid, blockdev, bootchartd, brctl, bunzip2, busybox, bzcat, bzip2, cal, cat, chat, chattr, chcon, 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, 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, flash_eraseall, flash_lock, flash_unlock, flashcp, flock, fold, free, freeramdisk, fsck, fsck.minix, fsfreeze, fstrim, fsync, ftpd, ftpget, ftpput, fuser, getenforce, getopt, getsebool, getty, grep, groups, gunzip, gzip, halt, hd, hdparm, head, hexdump, hexedit, hostid, hostname, httpd, hush, hwclock, 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, load_policy, loadfont, loadkmap, logger, login, logname, logread, losetup, lpd, lpq, lpr, ls, lsattr, lsmod, lsof, lspci, lsscsi, lsusb, lzcat, lzma, lzop, lzopcat, makedevs, makemime, man, matchpathcon, md5sum, mdev, mesg, microcom, minips, mkdir, mkdosfs, mke2fs, mkfifo, mkfs.ext2, mkfs.minix, mkfs.reiser, mkfs.vfat, mknod, mkpasswd, mkswap, mktemp, modinfo, modprobe, more, mount, mountpoint, mpstat, mt, mv, nameif, nanddump, nandwrite, nbd-client, nc, netcat, 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, printenv, printf, ps, pscan, pstree, pwd, pwdx, raidautorun, rdate, rdev, readahead, readlink, readprofile, realpath, reboot, reformime, remove-shell, renice, reset, resize, restorecon, resume, rev, rfkill, rm, rmdir, rmmod, route, rpm, rpm2cpio, rtcwake, run-init, run-parts, runcon, runlevel, runsv, runsvdir, rx, script, scriptreplay, sed, selinuxenabled, sendmail, seq, sestatus, setarch, setconsole, setenforce, setfattr, setfiles, setfont, setkeycodes, setlogcons, setpriv, setsebool, 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, tune2fs, ubiattach, ubidetach, ubimkvol, ubirename, ubirmvol, ubirsvol, ubiupdatevol, udhcpc, udhcpd, udpsvd, uevent, umount, uname, uncompress, unexpand, uniq, unit, unix2dos, unlink, unlzma, unlzop, 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

BusyBox - Dowloading

- Create a `rootfs` installation directory
- Download BusyBox 1.32.0 sources from https://busybox.net
- Extract 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` →`Installation Options...`, set `Destination path for 'make install'` to the path of your `rootfs` directory.
  - Then enable support for the following commands: `ash`, `init`, `halt`, `mount`, `cat`, `mkdir`, `echo`, `ls`, `uptime`, `vi`, `ifconfig`, `httpd`



https://asciinema.org/a/281501

BusyBox - Installing and compiling

- Compiling: `make` or `make -j 8` (faster)
  Resulting size: 301,016 bytes only!
  Funny to see that we're using a 64 bit system to run such small programs!
- Installing: `make install`
- See the created directory structure and the symbolic links to `/bin/busybox`

```
rootfs
├── bin
│   ├── ash -> busybox
│   ├── busybox
│   ├── cat -> busybox
│   ├── ls -> busybox
│   ├── mount -> busybox
│   └── sh -> busybox
├── sbin
│   ├── halt -> ../bin/busybox
│   ├── ifconfig -> ../bin/busybox
│   └── init -> ../bin/busybox
└── usr
    └── sbin
        └── httpd -> ../../bin/busybox
```

Creating a root filesystem image

- Creating an empty file with a 1M size:

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

- Formating this file for the ext2 filesystem:

```
mkfs.ext2 rootfs.img
```

Populating the root filesystem

- Create a mount point:

```
sudo mkdir /mnt/rootfs
```

- Mounting the root filesystem image:

```
sudo mount -o loop rootfs.img /mnt/rootfs
```

- Filling the BusyBox file structure:

```
sudo rsync -a rootfs/ /mnt/rootfs/
```

- Flushing the changes into the mounted filesystem image:

```
sync
```

Booting Linux with the root filesystem

- Add a disk to the emulated machine:

```
qemu-system-riscv64 -nographic -machine virt -m 128M \
    -kernel opensbi/build/platform/generic/firmware/fw_payload.elf \
    -append "console=ttyS0 ro root=/dev/vda" \
    -drive file=rootfs.img,format=raw,id=hd0 \
    -device virtio-blk-device,drive=hd0 \
```

- You should see the root filesystem is mounted:

```
[    0.630560] EXT4-fs (vda): mounting ext2 file system using the ext4 subsystem
[    0.659433] EXT4-fs (vda): mounted filesystem without journal. Opts: (null)
[    0.663114] VFS: Mounted root (ext2 filesystem) readonly on device 254:0.
```

Completing and configuring the root filesystem (1)

- Create a dev directory.
  The devtmpfs filesystem will automatically be mounted there (as CONFIG_DEVTMPFS_MOUNT=y)
- Let's try to mount the proc and sysfs filesystems:

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

Completing and configuring the root filesystem (1)

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 -nographic -machine virt -m 128M \
    -kernel opensbi/build/platform/generic/firmware/fw_payload.elf \
    -append "console=ttyS0 ro root=/dev/vda" \
    -drive file=rootfs.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
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

Next steps

Starting Linux from U-Boot in QEMU

- Would allow to show the U-Boot bootloader here too
- Almost ready: loads U-Boot, loads the Linux kernel but fails in early kernel booting
- Investigations documented and ongoing on RISC-V *sw-dev* mailing list: `https://frama.link/TDCk_VBV`

What to remember

- Embedded Linux is easy. It makes it easier to get started with Linux.
- You just need a toolchain, a kernel and a few executables.
- RISC-V is a new, open Instruction Set Architecture, use it and support it!
- In embedded Linux, things don't change that much over time. You just get more features.

Going further

- Drew Fustini's unmatched presentation about Linux on RISC-V:
  https://tinyurl.com/y6j8lfyz
- Our "Embedded Linux system development" training materials (500+ pages, CC-BY-SA licence):
  https://bootlin.com/doc/training/embedded-linux/
- All our training materials and conference presentations:
  https://bootlin.com/docs/
- The Embedded Linux Wiki: presentations, howtos... contribute to it!
  https://elinux.org

Questions? Suggestions? Comments?

Michael Opdenacker

michael.opdenacker@bootlin.com

https://bootlin.com/pub/conferences/2020/lee/