embedded-linux-labs Documentation

Release 1.1.0

Philippe Mariman

Mar 16, 2020

Contents:

Series of labs giving an overview of Linux systems in embedded projects and supporting customers developing such systems. Through some theory and practical labs, the architecture of an embedded Linux system, how to build such a system, how to take advantage of open source components to implement system features and reduce development costs, and details how to develop and debug your own applications in an embedded environment. After full completion of this training module, the participant will be ready to start a project using embedded Linux, from system building to application development.

CHAPTER 1

Introduction

1.1 Introduction

1.1.1 References

Important: The labs use the Bootlin Embedded Linux training slides as reference material

- Bootlin slides, chapter: *Introduction to Embedded Linux*
- Bootlin slides, chapter: *Embedded Linux Development*

1.1.2 Hardware for Labs

- Development board (Raspberry Pi 3, Banana Pi M2+, . . .)
- Power supply
- USB to UART TTL cable
- Housing for board

1.1.3 Terminology

Host System Workstation where development and cross compiling is done

Target System The embedded target

1.1.4 Host Tools and Libraries

- **fdisk**: Partitioning tool

- **gcc**: GNU C compiler (package `build-essential`)
- **git**: Software version control software
- **kpartx**: Tool to create device maps from partition tables
- **libncurses5-dev**: Needed to run kconfig for Buildroot
- **minicom**: Terminal emulator
- **mkfs**: Tool to create a filesystem
- **qemu**: Emulation tool
- **ssh**: Secure Shell
- **vim**: Command line text editor

1.1.5 Conventions & Host Set Up

- Debian (or any Debian based distribution) works best
- No GUI environment needed (!)
- Default shell: **bash**
- Directory set up:

Directory	Shell variable	Description
`/home/user/dev/`	`${DEV_PATH}`	main development directory
`/home/user/dev/dl`	`${DL_PATH}`	downloads directory
`/home/user/dev/staging`	`${STAGING_PATH}`	target rootfs staging directory
`/home/user/dev/linux`	`${LINUX_PATH}`	kernel source directory
`/home/user/dev/buildroot`	`${BR_PATH}`	Buildroot source directory
`/home/user/dev/mnt`	`${MNT_PATH}`	mountpoints directory

1.1.6 Programming Rules

- All code produced shall follow the Linux Coding Style [link]
- All code provided for review shall compile (at least)
- All code delivered must be uploaded to git repo.

1.1.7 Warnings

Important:

- When removing the SD card from a target, always first power off the target. Otherwise the SD card might get damaged.
- Disable automounting volumes on the host system. Otherwise formatting the SD card does not work when filesystems of the SD card are mounted.
- If the target has on board flash (eMMC), make sure it is erased. The only bootable source for the labs should be the SD card.

1.1.8 Not Covered (yet)

- Building cross-compilation toolchain with **crosstool-ng**
- NFS mounting the root filesystem
- Patching the kernel sources
- Raw flash filesystems
- Audio and video subsystems
- Real time Linux
- Remote application debugging

1.2 Analyse and Explore a Linux System

1.2.1 References

- `man` pages

1.2.2 Goals

- Analyse a Linux system in detail
- Understand important concepts regarding Linux

1.2.3 Steps

1. Check which `tty` the serial port is connected to:

```
user@host: ps -ef | grep tty
```

2. Check the partition table, their usage and which type of file systems are used:

```
user@host: fdisk -l /dev/sda
```

or:

```
user@host: lsblk /dev/sda
```

3. Check which file systems are mounted:

```
user@host: mount
```

or:

```
user@host: lsblk -f /dev/sda
```

4. Determine the details of the CPU:

```
user@host: cat /proc/cpuinfo
```

5. Determine the details of the memory:

```
user@host: cat /proc/meminfo
```

6. Determine which kernel modules are loaded:

```
user@host: lsmod
```

7. Determine available network interfaces:

```
user@host: ip l
```

1.2.4 Important Concepts

Serial port A UART or RS232 port available to the system which can provide a console/terminal. No need for screen and keyboard for basic terminal support like on desktop systems.

Partition table Sector on storage medium that describes the physical partioning of the storage memory. Most common schemes are *Master Boot Record (MBR)* and *GUID Partition Table (GPT)*.

File system Method that controls how data (files, directories, metadata) is stored and retrieved. It manages the storage memory. Most common file systems are *fat* and *ext4*.

Bootloader Usually collection of software pieces that bootstrap a system. Its most important objective is to load the OS kernel into main memory.

Kernel Core of the OS which controls and manages everything (hardware, users, processes, protocols, ...). It provides an interface to user processes to access resources.

Kernel module Dynamically loadable object file that can extend the functionality of the running kernel. Modules can implement any feature from device drivers to filesystems to protocols.

Device tree A data structure that describes the hardware details of a platform. Loaded and used by the kernel to manage those components (CPU, RAM, IO buses, peripherals, ...).

Root file system The file system where the root directory is located. Contains user space programs, kernel modules to be loaded dynamically, mount points for other file systems, ...

Init system The kernel will start only 1 user process at the end of its own initialization. This process is responsible to bootstrap the rest of user space.

1.3 Distribution Based Root Filesystem

1.3.1 References

- https://www.debian.org/intro/why_debian
- https://www.armbian.com

1.3.2 Goals

- Extract and install an SD card image
- Analyse a Distribution based embedded Linux system

1.3.3 Concepts

TBD

1.3.4 Steps

1. Download the file **rootfs-xxx-debian.img.gz** from the server (where *xxx* stands for the platform):

```
user@host: cd ${DL_PATH}
user@host: wget ${serverip}/downloads/rootfs-xxx-debian.img.gz
```

2. Extract the image on your host:

```
user@host: gunzip rootfs-xxx-debian.img.gz
```

3. Byte copy the extracted image to an SD card (for example: `/dev/mmcblk0` or `/dev/sda`):
   - **WARNING**: clearly check the device id of the SD card!
   - **WARNING**: clearly check that the SD card partitions are not automatically mounted!

```
user@host: dd if=rootfs-xxx-debian.img of=/dev/mmcblk0 bs=1M
```

4. Insert the SD card in the target and boot the board:
   - Watch the serial output while booting, using `minicom` at a baud rate of 115200 kbps

5. Explore the system (lookup login name and password)

```
login:  user_name
passwd: user_password
```

6. On the target: If the boot partition is not mounted on `/boot` or `/boot` does not exist at all:
   - **WARNING**: clearly check the device id of the SD card!

```
root@target: mkdir /boot
root@target: mount /dev/mmcblk0p1 /boot
```

7. Get the IP address and log in over SSH

1.3.5 Questions

- What is the default shell? (hint: shell variable `$SHELL`)
- What binaries are installed? (hint: look in `/bin`, `/sbin`, `/usr/bin`, `/usr/sbin`, `/lib`, `/usr/lib`)
- What is the size of binary and library directories? (hint: use `du`) (hint: look in `/bin`, `/sbin`, `/usr/bin`, `/usr/sbin`, `/lib`, `/usr/lib`)
- What is the total size of each filesystem? (hint: use `df`)
- What is the used size of each filesystem? (hint: use `df`)
- What are the filesystem usage percentages? (hint: use `df`)
- Which kernel is running? (hint: use `uname`)
- Is Python installed? Why would this be used on an embedded system?

- Is Perl installed? Why would this be used on an embedded system?
- What are the steps to update the system?

1.4 Simple Root Filesystem

1.4.1 References

- Bootlin slides, chapter: *Linux Root Filesystem*
- Bootlin slides, chapter: *Busybox*

1.4.2 Goals

- Extract and install an SD card image
- Analyse a simple embedded Linux system
- Understand the Busybox command multiplexer

1.4.3 Steps

1. Download a **rootfs-xxx-simple.img.gz** from the server (where *xxx* stands for the platform):

```
user@host: cd ${DL_PATH}
user@host: wget ${serverip}/downloads/rootfs-xxx-simple.img.gz
```

2. Extract the image on your host:

```
user@host: gunzip rootfs-xxx-simple.img.gz
```

3. Byte copy the extracted image to an SD card (for example: `/dev/mmcblk0` or `/dev/sda`):
   - **WARNING**: clearly check the device id of the SD card!
   - **WARNING**: clearly check that the SD card partitions are not automatically mounted!

```
user@host: dd if=rootfs-simple.img of=/dev/mmcblk0 bs=1M
```

4. Insert the SD card in the target and boot the board
   - Watch the serial output while booting, using `minicom` at a baud rate of 115200 kbps

5. Explore the system:

```
login:  root
passwd: root
```

6. On the target: If the boot partition is not mounted on `/boot` or `/boot` does not exist at all:
   - **WARNING**: clearly check the device id of the SD card!

```
root@target: mkdir /boot
root@target: mount /dev/mmcblk0p1 /boot
```

7. Get the IP address and log in over SSH

1.4.4 Questions

- What is the default shell?
- What binaries are installed?
- What is the size of binary and library directories?
- What is the partition layout?
- What filesystems are mounted?
- What is the total size of each filesystem?
- What is the used size of each filesystem?
- What are the filesystem usage percentages?
- Which kernel is running?
- Where are the kernel image and device tree blob located?
- What is special about standard utilities (`ls`, `cat`, ...)?
- What are the differences with a Debian based embedded Linux systems?

CHAPTER 2

Embedded Linux

2.1 Configure and Build the Linux Kernel

2.1.1 References

- Bootlin slides, chapter *Linux kernel introduction*
- Mainline kernel: https://www.kernel.org/

2.1.2 Goals

- Understand the Linux kernel build and configuration process
- Build a kernel for the native architecture of the host (using the native distribution C compiler)
- Test the kernel on an emulated x86 system

2.1.3 Concepts

- The default architecture is **x86_64**.
- All supported architectures have subdirectory in `arch/` subdirectory of the kernel source.
- Main architectures for embedded Linux devices: `arm`, `mips`, `x86`, `powerpc`

Important: The following steps are taken to configure, build and use a kernel:

```
user@host: make <some-defconfig>
user@host: make
user@host: make install
```

2.1.4 Steps

1. Get the kernel source:

```
user@host: cd ${DL_PATH}
user@host: wget https://cdn.kernel.org/pub/linux/kernel/v5.x/linux-5.0.6.tar.xz
```

2. Extract the source tarball on your host:

```
user@host: mkdir ${LINUX_PATH}
user@host: tar -C ${LINUX_PATH} -xvf linux-5.0.6.tar.xz
user@host: cd ${LINUX_PATH}/linux-5.0.6/
```

3. Set a default configuration for **x86_64**:

```
user@host: make x86_64_defconfig
```

4. Optionally: Start kernel configuration menu:

```
user@host: make menuconfig
```

5. Start kernel build process, (note: the number of jobs for make should equal 2*cpus + 1):

```
user@host: make -j 9
```

6. Test if the kernel boots in an emulated environment:

```
user@host: qemu-system-x86_64 -M pc -no-reboot -kernel arch/x86/boot/bzImage \
            -append "panic=1 console=tty1"
```

2.1.5 Hints

- The following packages might need to be installed in order to build:

```
user@host: sudo apt install libssl-dev flex bison
```

- Remove the `panic=1` parameter to not reboot the virtual machine

2.1.6 Questions

- What does the error generated at the end of the kernel initialization mean?
- Under which path in the kernel configuration is the CPU architecture defined?

2.2 Linux Kernel Boot Process & Command Line Parameters

2.2.1 References

- Bootlin slides, chapter *Linux kernel introduction*
- Mainline kernel: https://www.kernel.org/

2.2.2 Goals

- Understand the Linux kernel boot process
- Test the kernel + initramfs on an emulated `x86_64` system

2.2.3 Kernel Command Line Parameters

- Passed by the bootloader to the kernel.
- Kernel parses the arguments.
- https://www.kernel.org/doc/Documentation/admin-guide/kernel-parameters.txt
- Most important generic options:

Option	Description	Example
`console=`	kernel console output	`console=tty1`
`elevator=`	IO scheduler selection	`elevator=noop`
`init=`	init process to start by the kernel	`init=/usr/local/bin/myinit`
`rdinit=`	init process to start by the kernel inside initramfs	`rdinit=/bin/sh`
`quiet`	disable most log messages	
`root=`	root filesystem partition	`root=/dev/mmcblk0p2`
`rootdelay=`	seconds to wait before mounting root	`rootdelay=2`
`rootflags=`	root filesystem mount options	`rootflags=rw,noatime`
`rootwait`	wait indefinitly for root partition	`root=/dev/sda2 rootwait`

2.2.4 Booting and Init

- The bootloader loads the kernel in RAM and starts execution.
- The kernel initializes itself and the drivers.
- Depending on kernel commandline parameters, it tries to mount the root filesystem.
- Depending on kernel commandline parameters, it starts the user space **init** program.
- The init program is the only process the kernel starts and is referenced as **PID 1**.
- The init program starts the rest of userspace and never exits.

2.2.5 Initramfs

- Kernel can boot in **ramdisk** (legacy) or **initramfs**.
- Intermediary step before mounting the root filesystem and booting into it.
- Initramfs can be used to prepare next booting stage (root partition, . . .).
- Can be appended to the kernel image, or can be loaded seperately by the bootloader.
- Packaged in **cpio** archive, optionally compressed.

- The initramfs is extracted into a **tmpfs** filesystem in RAM.
- https://www.kernel.org/doc/Documentation/filesystems/ramfs-rootfs-initramfs.txt

2.2.6 Virtual filesystems proc and sysfs

- The `proc` virtual filesystem is provided by the kernel.
- It provides:
  - Process statistics and process run time information
  - User access to adjust run time parameters (process management, ...)
- It has to be mounted in user space (`mount -t proc nodev /proc`).
- Used by many common applications (`ps`, `top`, ...).
- https://www.kernel.org/doc/Documentation/filesystems/proc.txt
- `man 5 proc`
- Process specific information located in directory per process `/proc/<pid>`.
- Examples:

proc entry	Description
`/proc/cmdline`	kernel command line at boot
`/proc/cpuinfo`	list of available cpu's in the system
`/proc/filesystems`	list of available filesystems by the kernel
`/proc/partitions`	list of partitions in the system
`/proc/self`	symlink to `/proc/<pid>` of process which opens file
`/proc/self/cmdline`	command line which started process
`/proc/self/comm`	process command
`/proc/self/cwd`	symlink to current working directory
`/proc/self/exe`	symlink to binary which was used for `exec`
`/proc/self/environ`	process environment variables
`/proc/self/fd`	overview of all opened file descriptors of process
`/proc/self/limits`	process attributes (stack size, core dump size, ...)
`/proc/self/mounts`	overview of mounted filesystems

- The `sysfs` virtual filesystem is provided by the kernel.
- It provides an hierarchical directory structured view of the buses, devices and drivers the kernel manages on the system.
- It has to be mounted in user space (`mount -t sysfs nodev /sys`).
- Used by applications to communicate with hardware (drivers) (`udev`, `ip link`, ...).
- https://www.kernel.org/doc/Documentation/filesystems/sysfs.txt
- Examples:

sys entry	Description
/sys/class/net/enp0s25/address	MAC address of interface enp0s25
/sys/class/net/enp0s25/carrier	carrier signal of interface enp0s25

2.2.7 Kernel Logging

- The dmesg command dumps the kernel log ring buffer.
- Kernel log messages are also sent to syslogd or klogd, these log daemons write the logs to /var/log/messages.
- The kernel log buffer size can be set in the kernel configuration (CONFIG_LOG_BUF_SHIFT).
- Example: inserting a USB flash drive, a way to get the dynamic mapping of the inserted device:

```
...
[618.381500] usb 1-1.2: new high-speed USB device number 4 using ehci-pci
[618.491472] usb 1-1.2: New USB device found, idVendor=0781, idProduct=5583
[618.491476] usb 1-1.2: New USB device strings: Mfr=1, Product=2, SerialNum
[618.491478] usb 1-1.2: Product: Ultra Fit
[618.491480] usb 1-1.2: Manufacturer: SanDisk
[618.491481] usb 1-1.2: SerialNumber: 4C530001190526106240
[618.554722] usb-storage 1-1.2:1.0: USB Mass Storage device detected
[618.554829] scsi host6: usb-storage 1-1.2:1.0
[618.554943] usbcore: registered new interface driver usb-storage
[618.566518] usbcore: registered new interface driver uas
[619.559076] scsi 6:0:0:0: Direct-Access SanDisk  Ultra Fit 1.00 PQ: 0 ANS
[619.559658] sd 6:0:0:0: Attached scsi generic sg1 type 0
[619.560156] sd 6:0:0:0: [sdb] 60062500 512-byte logical blocks: 30.8 GB
[619.561591] sd 6:0:0:0: [sdb] Write Protect is off
[619.561595] sd 6:0:0:0: [sdb] Mode Sense: 43 00 00 00
[619.563817] sd 6:0:0:0: [sdb] Write cache: disabled, read cache: enabled,
[619.576674]  sdb: sdb1
[619.580158] sd 6:0:0:0: [sdb] Attached SCSI removable disk
[619.916117] EXT4-fs (sdb1): mounted filesystem with ordered data mode. Op
```

2.2.8 Steps

1. Build a kernel for x86$_$64 and test it in **qemu**:

```
user@host: qemu-system-x86_64 -M pc -no-reboot \
            -kernel ${LINUX_PATH}/linux-5.0.6/arch/x86/boot/bzImage \
            -append "panic=1 console=tty1"
```

2. Create a simple custom init program (*init-hello-world.c*) to test rdinit= within qemu:

```
#include <stdio.h>
#include <unistd.h>

void main(void)
{
    printf("hello world\n");
    sleep(99999999);
}
```

3. Compile the test program with statically linked C library and strip it:

```
user@host: mkdir -p /tmp/ramfs/sbin/
user@host: gcc --static -o /tmp/ramfs/sbin/myinit /tmp/init-hello-world.c
user@host: strip /tmp/ramfs/sbin/myinit
```

4. Create a root file system **cpio archive** which include the init program:

```
user@host: cd /tmp/ramfs
user@host: find . | cpio -o -H newc | gzip > /tmp/root.cpio.gz
```

5. Test the custom init and root file system archive:

```
user@host: qemu-system-x86_64 -M pc -no-reboot \
               -kernel ${LINUX_PATH}/linux-5.0.6/arch/x86/boot/bzImage \
               -append "panic=1 console=tty1 rdinit=/sbin/myinit" \
               -initrd /tmp/root.cpio.gz
```

2.2.9 Hints

- The X11 qemu window can be bypassed, for example on a server or inside a container:
  - The `stdio` of the qemu machine is forwarded to the running terminal
  - Use the following options upon invocation: `-nographic -append "...console=ttyS0"`

2.2.10 Assignments

- Test some use cases and generate some **kernel panics** to analyse.
- Reduce the size of the kernel by removing features, drivers, ... But make sure it still boots.

2.2.11 Questions

- Check the kernel init sequence in the source: function `kernel_init()` in the file `${LINUX_PATH}/init/main.c`
- What is the **sequence of init locations** the kernel searches by default?
- Why must the custom init program be statically linked? (try with dynamic linking also)
- How come the `printf()` function in the custom init program prints to the correct console?
- Let the custom init program return, analyse the kernel panic, why did it panic?
- Why can only a `x86_64` CPU be used? Why not an ARM CPU?

2.3 Busybox

2.3.1 References

- Bootlin slides, chapter *Busybox*

2.3.2 Goals

- Build a simple root file system with a minimal Busybox configuration

2.3.3 Steps

1. Download the Busybox sources:

```
user@host: cd ${DL_PATH}
user@host: wget https://busybox.net/downloads/busybox-1.30.1.tar.bz2
```

2. Extract the Busybox sources:

```
user@host: tar -xvf busybox-1.30.1.tar.bz2
user@host: cd busybox-1.30.1/
```

3. Set the empty config and enter the configuration:

```
user@host: make allnoconfig
user@host: make menuconfig
```

4. Enable the option in the configuration to build statically, symbol `CONFIG_STATIC`:

```
--> Busybox Settings
        --> Build Options
                --> Build Busybox as a static binary (no shared libs)
```

5. Select `ash`, `vi` and all `coreutils`
6. Select the following settings:

```
CONFIG_SHOW_USAGE=y
CONFIG_FEATURE_VERBOSE_USAGE=y
CONFIG_FEATURE_COMPRESS_USAGE=y
CONFIG_BUSYBOX=y
CONFIG_FEATURE_INSTALLER=y
CONFIG_PLATFORM_LINUX=y
CONFIG_INSTALL_APPLET_SYMLINKS=y
```

7. Build and inspect:

```
user@host: make -j 9
user@host: file busybox
```

8. Install Busybox tree:

```
user@host: make CONFIG_PREFIX=../bb_install install
```

2.3.4 Assignments

- Create a *cpio* archive containing the Busybox install tree
- Run the *cpio* archive as initramfs
- Show a directory listing of the root file system in run time

2.3.5 Questions

- Create a file */root/hello* containing "hello world", reboot, what happens to the file? Why?
- What directories are missing in the root file system?

2.3.6 Busybox init System

- Busybox provides a simple `init` system equivalent to `System V init`
- It does not offer a lot of features, like runlevels, but it is a very simple implementation
- `/sbin/init` The init binary (or soft link)
- `/etc/inittab` Jobs description file for the **init** system

```
user@host: cat bb_example/etc/inittab
# /etc/inittab
::sysinit:/etc/init.d/rcS
ttyS0::respawn:/sbin/getty -L ttyS0 0 vt100
```

- `/etc/init.d/rcS` Init script to kickstart all other startup scripts in this directory.

```
user@host: cat bb_example/etc/init.d/rcS
#!/bin/sh

mount -a
```

- `etc/fstab` Description of administered system mount points; all mounted during init.

```
user@host: cat bb_example/etc/fstab
# /etc/fstab
#device          mount-point  type      options           dump  fsck order
/dev/mmcblk0p1   /boot        ext4      rw,noauto         0     1
proc             /proc        proc      defaults          0     0
sysfs            /sys         sysfs     defaults          0     0
devtmpfs         /dev         devtmpfs  mode=0755,nosuid  0     0
tmpfs            /tmp         tmpfs     defaults          0     0
tmpfs            /run         tmpfs     defaults          0     0
```

2.3.7 Assignments

- Enable the standard configuration for Busybox (`make defconfig`)
- Create the necessary files to start a shell at the console
- Create or adapt a simple init script that runs a `hello-world` binary

2.3.8 Questions

- What starts the tty on the serial console?
- Which other init systems exist? Which ones are suitable for embedded systems and why?

2.3.9 Adding User Files

```
user@host: cat bb_example/etc/passwd
root:x:0:0::/root:/bin/sh
```

```
user@host: cat bb_example/etc/shadow
root::0::::::
```

2.4 Cross-compiling the Linux Kernel

2.4.1 References

- Bootlin slides, chapter *Linux kernel introduction*
- Mainline kernel: https://www.kernel.org/

2.4.2 Goals

- Cross-compile a kernel for an emulated target
- Use the Debian packaged cross-compiler
- Make a kernel configuration

2.4.3 Kernel make arguments for cross-compilation

- Possible parameters passed to `make` for cross compilation:

Option	Description	Example
`ARCH=`	indicate the architecture	`ARCH=arm`
`CROSS_COMPILE=`	cross compiler prefix	`CROSS_COMPILE=arm-linux-`
`INSTALL_MOD_PATH=`	path to install modules	`INSTALL_MOD_PATH=../staging`

- Pre-defined configuration files available for a lot of embedded boards.
- Architecture and vendor specific configurations enabled.

2.4.4 Steps

1. Install packaged ARM cross-compiler toolchain from distribution:

```
user@host: sudo apt install gcc-arm-linux-gnueabihf
```

2. Get the kernel source:

```
user@host: cd ${DL_PATH}
user@host: wget https://cdn.kernel.org/pub/linux/kernel/v5.x/linux-5.0.6.tar.xz
```

3. Extract the source tarball on your host:

```
user@host: mkdir ${LINUX_PATH}
user@host: tar -C ${LINUX_PATH} -xvf linux-5.0.6.tar.xz
user@host: cd ${LINUX_PATH}/linux-5.0.6/
```

4. Set a default configuration for **vexpress**, a motherboard containing an ARM Cortex-A9:

```
user@host: ARCH=arm make vexpress_defconfig
```

5. Optionally: Start kernel configuration menu:

```
user@host: ARCH=arm make menuconfig
```

6. Start kernel build process, (note: the number of jobs for make should equal 2*cpus + 1):

```
user@host: ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- make -j 9
```

7. Test if the kernel boots in an emulated environment:

```
user@host: qemu-system-arm -M vexpress-a9 -nographic -no-reboot \
                -kernel arch/arm/boot/zImage \
                -dtb arch/arm/boot/dts/vexpress-v2p-ca9.dtb \
                -append "panic=5 console=ttyAMA0"
```

8. The kernel should panic

2.4.5 Assignments

- Cross compile *init-hello-world* with **arm-linux-gnueabihf-gcc** and test `init=` in qemu

2.4.6 Questions

- Show the differences between the default and your kernel configuration.
- What is the *dtb* file being passed to the qemu?

2.4.7 Staging Installation Steps (Preparing the root file system)

1. Cross-compile the kernel modules:

```
user@host: ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- make -j 9 modules
```

2. Copy the kernel to staging directory:

```
user@host: mkdir ${STAGING_PATH}/boot
user@host: cp arch/arm/boot/zImage ${STAGING_PATH}/boot/zImage
```

3. Copy the device tree to staging directory:

```
user@host: cp arch/arm/boot/dts/vexpress-v2p-ca9.dtb \
                    ${STAGING_PATH}/boot/vexpress-v2p-ca9.dtb
```

4. Install kernel modules to staging directory:

```
user@host: ARCH=arm INSTALL_MOD_PATH=${STAGING_PATH} make modules_install
```

2.4.8 Assignments

- Clean the Busybox build
- Enable the following applets in the Busybox menu: `modprobe`, `lsmod`
- Cross compile Busybox
- Install Busybox to the the staging directory
- Create a compressed *cpio* archive and boot in `qemu`
- Load a kernel module using `modprobe` and check with `lsmod`

2.4.9 Optional Assignments

- Cross compile *toybox*, create a new compressed *cpio* archive and boot in *qemu*. Compare the differences between *toybox* and *busybox*.
- Configure the kernel in a way to make the resulting binary as small as possible (non used funtionality can be removed from the config, e.g. ipv6, wireless networking, exotic protocols, file systems, . . .)

2.5 Linux Kernel Modules

2.5.1 References

- Bootlin slides, chapter *Linux kernel introduction*
- Mainline kernel: https://www.kernel.org/
- Kernel documentation on out of tree builds: https://www.kernel.org/doc/Documentation/kbuild/modules.txt
- http://derekmolloy.ie/writing-a-linux-kernel-module-part-1-introduction/

2.5.2 Goals

- Make a simple *out-of-tree kernel module*
- Build the kernel module for a target (e.g. `qemu-system-arm`)
- Test the module and module commands on a target (e.g. `qemu-system-arm`)

2.5.3 Steps

1. Create a `hello.c` file containing the following code inside a new directory `hello-module/`:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

static int __init hello_init(void)
```

(continues on next page)

(continued from previous page)

```
{
        printk(KERN_INFO "########################## hello module\n");
        return 0;
}

static void __exit hello_exit(void)
{
        printk(KERN_INFO "########################## goodbye module\n");
}

module_init(hello_init);
module_exit(hello_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("foo");
MODULE_DESCRIPTION("a simple hello kernel module");
MODULE_VERSION("0.01");
```

2. Create a `Makefile` file containing the following code:

```
obj-m += hello.o

KBUILD=$(LINUX_PATH)

all:
        make -C $(KBUILD) M=$(PWD) modules
install:
        make -C $(KBUILD) M=$(PWD) INSTALL_MOD_PATH=$(STAGING_PATH) modules_install
clean:
        make -C $(KBUILD) M=$(PWD) clean
```

Important: Makefile commands do not include architecture and cross compiler info

3. Build the module:

```
user@host: make && make install
```

2.5.4 Assignments

- Create a root file system containing the module (use `${STAGING_PATH}`)
- Test the module on a target, e.g. `qemu-system-arm`
- List all shell commands that manipulate kernel modules
- Make the module load automatically at boot

2.5.5 Questions

- What are the purpose of kernel modules?
- What well known external kernel modules exist?
- Where are out-of-tree modules installed?

- What are the risks of using/depending on externally maintained kernel modules?

2.6 Buildroot Initial Config

2.6.1 References

- Main Buildroot documentation: https://buildroot.uclibc.org/downloads/manual/manual.html

2.6.2 Goals

- Explore the Buildroot configuration system
- Make an initial configuration for Buildroot
- Build a first **Board Support Package (BSP)** with Buildroot
- Analyse the generated output

2.6.3 Steps

1. Download a recent Buildroot version: **2019.02.1**:

```
user@host: cd ${DL_PATH}
user@host: wget https://buildroot.uclibc.org/downloads/buildroot-2019.02.1.tar.gz
```

2. Extract the tarball and `cd` into the unpacked directory:

```
user@host: tar -xvf buildroot-2019.02.1.tar.gz
user@host: mv buildroot-2019.02.1/ ${BR_PATH}
user@host: cd ${BR_PATH}
```

3. Initialize the configuration with the default one provided (where *xxx* stands for the platform):

```
user@host: make xxx_defconfig
```

Important: For example: `raspberrypi3_defconfig`, `beaglebone_defconfig`

4. Enter the configuration menu:

```
user@host: make menuconfig
```

5. Update the configuration as following:
   - set the download directory to ${DL_PATH}
   - include WCHAR support
   - include C++ support
   - set root passwd "root"
   - start a getty on `ttyAMA0` or `ttyS0` (depending on the platform)
   - let Buildroot generate a "tar" filesystem image

6. Start the build:

```
user@host: make
```

2.6.4 Assignments

- Build a SD card image for a hardware platform
- Configure the system to automatically handle DHCP on the network interface
- Test the image on the hardware

2.6.5 Questions

- Compare the `Target options` in the menuconfig (using the defconfigs) for:
  - **raspberrypi3** vs **beaglebone**
  - **beaglebone** vs **qemu_x86_64**
- What is the default configured `libc` for the cross toolchain? And what others are available?
- Which version of `gcc` is used for the cross toolchain? What version is shipped on the host machine?

2.6.6 Buildroot Output Directories

- Buildroot will generate its build artifacts under the `output/` directory:
  - `build/`: contains all packages *build* subdirectories
  - `host/`: contains the generated host tools (compiler, autotools, ...)
  - `images/`: contains the generated target images (kernel, device tree blob, bootloader, filesystem images, tarballs, SD card images)
  - `target/`: mimics the root filesystem excluding `/dev`

2.6.7 Hints

- Some supported `make` targets:

```
# print help
user@host: make help

# print existing board configurations
user@host: make list-defconfigs

# load a default configuration file from configs/ directory
user@host: make <target_defconfig>

# start the configuration menu
user@host: make menuconfig

# start the Linux kernel configuration menu
user@host: make linux-menuconfig
```

(continues on next page)

(continued from previous page)

```
# start the Busybox configuration menu
user@host: make busybox-menuconfig

# run the download stage to download all software sources
user@host: make source

# start a build
user@host: make

# export the current configuration
user@host: make savedefconfig
```

- Testing Buildroot results in `qemu`:

```
# run a kernel with the generated ext4 partition image
user@host: qemu-system-x86_64 -kernel qemu_client_br/output/images/bzImage -m 128 \
           -hda qemu_client_br/output/images/rootfs.ext4 -append "root=/dev/sda"

# run a kernel with the generated ext4 partition image, with a serial console
user@host: qemu-system-x86_64 -kernel qemu_client_br/output/images/bzImage -m 128 \
           -hda qemu_client_br/output/images/rootfs.ext4 \
           -append "console=ttyS0,115200 root=/dev/sda1" --nographic

# run a kernel with the generated initramfs image, with a serial console
user@host: qemu-system-x86_64 -kernel qemu_client_br/output/imagesbzImage -m 128 \
           -initrd qemu_client_br/output/imagesrootfs.cpio.gz \
           -append "console=ttyS0,115200 root=/dev/sda1" -hda /dev/sdb --nographic
```

2.7 Extending Buildroot System Image

2.7.1 References

- Bootlin slides, chapter **Embedded Linux system development**
- Main Buildroot documentation: https://buildroot.uclibc.org/downloads/manual/manual.html

2.7.2 Goals

- Get familiar with user space tools and libraries for embedded Linux systems

2.7.3 Assignment

- Change the default hostname of the system image.
- Add a SSH server to the configuration.
- Add a web server to the configuration.
- Add the `netcat` utility from Busybox.
- Add the `ss` utility. Figure out which package to add.
- Add `libzmq` and `libcurl` to the configuration.

- Add a complex package (NodeRed, Python, . . .)

2.7.4 Steps

1. Do the necessary configuration of the assignment and build:

```
user@host: cd ${BR_PATH}
user@host: make menuconfig
user@host: make
```

2. Insert the SD card in the host machine and clear the `root` partition:

```
root@host: mount /dev/mmcblk0p2 ${MNT_PATH}
root@host: rm -rf ${MNT_PATH}/*
```

3. Unpack the latest generated tar image from Buildroot on the `root` partition:

```
root@host: tar -C ${MNT_PATH} -xvf ${BR_PATH}/output/images/rootfs.tar
```

4. Unmount the SD card and test on the target:

```
root@host: umount ${MNT_PATH}
```

- **Note:** as an alternative to extracting a tar archive to the second file system, a full file system image can be byte copied into place (if the partition is big enough):
- **WARNING:** clearly check that the SD card partitions are not automatically mounted!

```
user@host: cd ${BR_PATH}
root@host: dd if=${BR_PATH}/output/images/rootfs.ext4 of=/dev/mmcblk0p2 bs=1M
```

2.7.5 Hints

- The default size of the `ext4` partition of the SD card image might be too small for these changes. Increase the size of the `ext4` image in the *menuconfig*. Also increase the size of the partition and the filesystem on the SD card.

2.7.6 Questions

- Which package did you enable to include a SSH server?
- Which package did you enable to include a web server?
- What are the differences between the `netcat` utilities from Busybox and on your host system?
- Which package did you enable to include `ss`? What is the impact of this for the target root filesystem?

2.8 Cross-Compiling Applications

2.8.1 References

- Bootlin slides, chapter *Cross-Compiling Toolchains*

- Main Buildroot documentation: https://buildroot.uclibc.org/downloads/manual/manual.html

2.8.2 Goals

- Cross-compile an application for the target
- Using the Buildroot cross-compiler and sysroot

2.8.3 Steps

1. Compile a simple C program with the host toolchain, using a Makefile.
2. Adapt the Makefile and compile a simple C program with the toolchain from Buildroot:

```
user@host: ls -l ${BR_PATH}/output/host/usr/bin
```

3. Copy the binary to the target and test it (use the simple SD card image from previous lab).

2.8.4 Questions

- Which binaries are present in the toolchain directory?
- Which version of gcc is used?
- Which libc is gcc using?
- Are there other interesting parameters used in the gcc configuration?
- Check the location of the **sysroot**.
- Show that the generated binary is built for the target architecture using `readelf`:

```
user@host: readelf -a <binary>
user@host: readelf -h <binary>
```

2.9 Creating Boot Media from Scratch

2.9.1 References

- Bootlin slides, chapter *Block filesystems*
- Bootlin slides, chapter *Flash filesystems* (not covered)

2.9.2 Goals

- Filesystems Overview
- Usage of `kpartx` on boot images
- Create an SD card set up
- Bootloader installation and configuration
- Kernel, device tree and modules installation

- Root filesystem installation

2.9.3 Storage Media

- **Block Devices:** Random access on a per-block basis
  - Hard drives, RAM disks (tmpfs)
  - eMMC, USB flash storage, SD card, SSD: these have integrated controller to emulate block device, also handles *wear-leveling* and bad blocks
- **Raw Flash Devices:** Driven by controller on SoC. Support for reading, erasing and writing.

2.9.4 Partition Table

- **Partitioning:** Dividing the storage media in multiple areas for different usage.
- **Partition Table:** Description of partions on the storage media.
- **MBR:** Legacy table format, first 512 bytes of storage media.
- **GPT:** New table format.

2.9.5 Filesystems Overview

- **Block filesystems:** These filesystems can be directly used on the storage media (hard disk or flash plus controller) to control individual blocks within a partition.
- **Flash filesystems:** These filesystems are for specific use for raw flash based devices. They operate on top of *MTD* layer, which manipultes the flash storage device. The current de-facto standard flash filesystem is *UBIFS*.

2.9.6 Steps

1. Create empty image of certain size:

```
user@host: dd if=/dev/zero of=sdcard.img bs=1M count=650
```

2. Create MBR partition table on empty image:

```
user@host: fdisk sdcard.img

Device      Boot  Start     End Sectors  Size Id Type
sdcard.img1        2048  104447  102400   50M  c W95 FAT32 (LBA)
sdcard.img2      104448 1292287 1187840  580M 83 Linux
```

3. Create device maps from partition table:

```
user@host: sudo kpartx -a sdcard.img
```

4. Create FAT32 filesystem on first partition (boot):

```
user@host: sudo mkfs.vfat -n boot /dev/mapper/loop0p1
```

5. Mount first partition and copy boot/ files from staging:

```
user@host: sudo mount /dev/mapper/loop0p1 ${MNT_PATH}
user@host: sudo cp ${STAGING_PATH}/boot/* ${MNT_PATH}
user@host: sudo umount ${MNT_PATH}
```

6. Create ext4 filesystem on second partition (root):

```
user@host: sudo mkfs.ext4 -L root /dev/mapper/loop0p2
```

7. Mount second partition and copy `root/` files from staging and rootfs image:

```
user@host: sudo mount /dev/mapper/loop0p2 ${MNT_PATH}
user@host: sudo tar -C ${MNT_PATH} -xf ${BR_PATH}/output/images/rootfs.tar
user@host: sudo cp -R ${STAGING_PATH}/root/ ${MNT_PATH}
user@host: sudo umount ${MNT_PATH}
```

8. Unmap image:

```
user@host: sudo kpartx -d sdcard.img
```

9. Some platforms require the SoC to load the SPL U-Boot directly from the MMC device:

```
user@host: sudo dd if=u-boot-with-spl.bin of=sdcard.img bs=1K seek=x
```

10. Write to SD card:

```
user@host: sudo dd if=sdcard.img of=/dev/mmcblk0 bs=1M
```

- Or test in qemu:

```
user@host: qemu-system-x86_64 -drive format=raw,file=sdcard.img
```

2.10 Debootstrap a Debian System

2.10.1 Concepts

- **Debootstrap:** Tool to install Debian base system into a subdirectory
- Only needs access to a Debian repository, like https://www.debian.org/mirror/list
- Cross architecture: cross-debootstrapping
- Debian one of the most stable systems available for desktop and server
- Not the most optimal solution for embedded systems
  - Further customisation for embedded systems possible
  - For example: included man pages, locales, . . .

2.10.2 References

- `man debootstrap`

2.10.3 Goals

- Create a Debian root file system
- Integrate custom kernel and modules
- Make a bootable storage medium and test on hardware platform

2.10.4 Steps

1. Prepare host tools:

```
root@host: apt install debootstrap qemu qemu-user-static binfmt-support fakeroot
```

2. Create a minimal install root file system (`--variant=minbase`)

```
root@host: debootstrap --no-check-gpg --foreign --arch=armhf  \
             --variant=minbase jessie rootfs/ \
             http://archive.raspbian.org/raspbian
```

3. Second stage debootstrapping:
   - Necessary for non native architecture, like `armhf`
   - Use qemu-arm-static for cross execution inside `rootfs/`

```
root@host: cp /usr/bin/qemu-arm-static rootfs/usr/bin/
root@host: LANG=C chroot rootfs/ /debootstrap/debootstrap --second-stage
root@host: LANG=C chroot rootfs/ apt-get clean
root@host: LANG=C chroot rootfs/ apt-get autoclean
root@host: rm rootfs/usr/bin/qemu-arm-static
```

4. Directory `rootfs/` can be copied to root file system partition of SD card

2.10.5 Assignments

- Configure the necessary configuration files
- Re-run the build procedure, but include some packages

2.10.6 Questions

- What are the benefits and drawbacks of this procedure?
- What are the possible use cases of using Debian in products?

2.10.7 Hints

- https://github.com/phmariman/linux-tinkering/tree/master/debian-arm-build
- Overlay directory:

```
linux-tinkering/debian-arm-build/deb-rpi-overlay
etc/
 - apt/
     - sources.list
 - fstab
 - hostname
 - modules
 - systemd/
    - system/
        - getty@ttyAMA0.service -> /lib/systemd/system/getty@.service
```

- Sample `/etc/fstab`:

```
root@host: cat rootfs/etc/fstab
proc            /proc      proc defaults         0 0
/dev/mmcblk0p1 /boot      vfat defaults         0 2
/dev/mmcblk0p2 /          ext4 defaults,noatime 0 2
/dev/mmcblk0p3 /opt/data  ext4 defaults,noatime 0 2
```

2.11 U-Boot Bootloader for Embedded Targets

2.11.1 References

- Bootlin slides, chapter *Bootloaders*
- *TBD* ...

2.11.2 Goals

- Understand the concept of multi-stage booting
- *TBD* ...

2.11.3 Steps

- Clone mainline U-Boot repository
- *TBD* ...

2.11.4 Assignments

- Compile and run on target
- Create a custom *boot.cmd* file
- *TBD* ...

2.11.5 Questions

- *TBD* ...

CHAPTER 3

Linux System Programming

3.1 Simple Log Module

3.1.1 Concepts

- Logs don't lie
- Log files are written to `/var/log/` or `/var/log/<application>`, for example:
  - `/var/log/messages`: generic system activity logs
  - `/var/log/cron.log`: Crond logs (cron job)
  - `/var/log/auth.log`: Authentication log
  - `/var/log/httpd/`: Apache access and error logs directory
  - ...
- Log files and directories are **rotated** to limit the space used by older logs (see `logrotate`)
- Log messages can be filtered away by implementing a treshold on log (severity) level

3.1.2 Assignment

- By default, log to the connected terminal
- Make functions to open, close, and print to a log file
- Support the *syslog* logging levels
- Write the logs with a timestamp of format **[2017/10/18 14:40:35.666]**
- Provide support for setting a global log treshold
- Create a header file for reuse in other programs
- Use `snprintf` and `vsnsprintf`

3.1.3 Module API

```
#ifndef LOG_HDR
#define LOG_HDR

// mapping of syslog log levels
#define LOG_LEVEL_EMERGENCY     0
#define LOG_LEVEL_ALERT         1
#define LOG_LEVEL_CRITICAL      2
#define LOG_LEVEL_ERROR         3
#define LOG_LEVEL_WARNING       4
#define LOG_LEVEL_NOTICE        5
#define LOG_LEVEL_INFO          6
#define LOG_LEVEL_DEBUG         7

int log_init(void);
int log_set_level(int lvl);

int log_console_println(int lvl, const char *str);
int log_console_printf(int lvl, const char *fmt, ...);

int log_file_open(const char *path, int truncate);
int log_file_close(int fd);
int log_file_println(int fd, int lvl, const char *str);
int log_file_printf(int fd, int lvl, const char *fmt, ...);

#endif
```

3.1.4 Example Output

```
[2017/10/18 14:40:35.666][INFO     ] HTTP GET /index.html 192.168.157.21
[2017/10/18 14:40:36.666][INFO     ] HTTP GET /index.html 192.168.157.133
[2017/10/18 14:40:37.666][ERROR    ] HTTP PUT /upload/badfile.txt 192.168.157.154
[2017/10/18 14:40:38.666][WARNING  ] HTTP GET /index.php 192.168.157.154 - error 404
[2017/10/18 14:40:39.666][INFO     ] HTTP GET /index.html 192.168.157.100
[2017/10/18 14:40:39.666][EMERGENCY] error - out-of-memory, shutting down daemon
```

3.1.5 Questions

- Why is `snprintf()` superior to `strcpy()` and `sprintf()`?

3.1.6 Extension (Optional)

- Colorize the outpur on the console regarding the log level (red = error for example)
- The terminal will interpret ANSI control characters as text formatting settings
- It is possible to check if a `fd` is a terminal: `isatty()`
- ANSI terminal control characters:

```
#define ANSI_RESET              "\x1b[0m"
#define ANSI_BOLD               "\x1b[1m"
```

(continues on next page)

(continued from previous page)

```
#define ANSI_UNDERLINE          "\x1b[4m"
#define ANSI_COLOR_TXT_BLACK    "\x1b[30m"
#define ANSI_COLOR_TXT_RED      "\x1b[31m"
#define ANSI_COLOR_TXT_GREEN    "\x1b[32m"
#define ANSI_COLOR_TXT_YELLOW   "\x1b[33m"
#define ANSI_COLOR_TXT_BLUE     "\x1b[34m"
#define ANSI_COLOR_TXT_MAGENTA  "\x1b[35m"
#define ANSI_COLOR_TXT_CYAN     "\x1b[36m"
#define ANSI_COLOR_TXT_WHITE    "\x1b[37m"
#define ANSI_COLOR_BKG_BLACK    "\x1b[40m"
#define ANSI_COLOR_BKG_RED      "\x1b[41m"
#define ANSI_COLOR_BKG_GREEN    "\x1b[42m"
#define ANSI_COLOR_BKG_YELLOW   "\x1b[43m"
#define ANSI_COLOR_BKG_BLUE     "\x1b[44m"
#define ANSI_COLOR_BKG_MAGENTA  "\x1b[45m"
#define ANSI_COLOR_BKG_CYAN     "\x1b[46m"
#define ANSI_COLOR_BKG_WHITE    "\x1b[47m"
```

- For example: print 'hello world' formatted bold + red:

```
dprintf(STDOUT_FILENO, ANSI_BOLD ANSI_COLOR_TXT_RED);
dprintf(STDOUT_FILENO, "hello world\n");
dprintf(STDOUT_FILENO, ANSI_RESET);
```

3.2 Timerwheel Module

3.2.1 Concepts

- A **timerwheel** is a data structure to organize multiple user timers based on a single core timer tick
- Used in both OS kernels and user space frameworks
- It can support a large number of user timers

3.2.2 Assignment

- Create a timer module, providing the ability to call a user callback upon timer expiration
- Use a hashed timerwheel and a linked list for every hash node
- Use a single POSIX timer a timer tick
- Make a generic implementation with user callback and data pointer
- Create a header file for reuse in other programs

3.2.3 Module API

```
#ifndef TIMERWHEEL_HDR
#define TIMERWHEEL_HDR

#define TIMER_STOP              0x00
#define TIMER_SINGLE_SHOT       0x01
```

(continues on next page)

(continued from previous page)

```
#define TIMER_INTERVAL          0x02
#define TIMER_AUTO_DEL          0x04

struct timerwheel_ctx;
struct timerwheel_node;

typedef void (*timer_cb)(struct timerwheel_node *node, void *data);

int timerwheel_init(struct timerwheel_ctx **wheel, int tick_period, int wheel_len);
int timerwheel_get_fd(struct timerwheel_ctx *wheel);
int timerwheel_start(struct timerwheel_ctx *wheel);
int timerwheel_tick(struct timerwheel_ctx *wheel);

int timerwheel_node_create(struct timerwheel_ctx *wheel,
                           struct timerwheel_node **handle);
int timerwheel_node_init(struct timerwheel_node *handle, timer_cb funcp, void *data);
int timerwheel_node_reschedule(struct timerwheel_node *handle, int period, int mode);
int timerwheel_node_remove(struct timerwheel_node *handle);

#endif
```

3.2.4 Hints

- Use the following structures as internal main context and node context:

```
// main wheel control structure
struct timerwheel_ctx {
        int fd;                         // timer file descriptor
        int len;                        // length of the wheel
        int tick_period;                // timer tick period in ms
        int current_slot;               // current timer slot
        struct list_node slots[];       // placeholder for variable array
};

// individual timer node structure
struct timerwheel_node {
        int expire;                     // number of ticks
        int mode;                       // singleshot, interval, auto-delete
        int period;                     // node in ms
        void *data;                     // user context to pass to callback
        timer_cb callback;              // user callback to call at expiration
        struct timerwheel_ctx *wheel;   // pointer to parent context
        struct list_node el;
};
```

3.2.5 Questions

- Where is this data structure used in Linux?
- What are the benefits of using such a data structure for timers?
- What other data structures exist for implementing timers? With or without timer tick?

3.3 Event Loop

3.3.1 Concepts

- Event loops
- Asynchronous event handling
- File descriptor based IO multiplexing

3.3.2 Assignment

- Create an **event loop** module implementation
- Use the `epoll` system call
- Make a generic implementation with user callback and data pointer
- Create a header file for reuse in other programs

3.3.3 Module API

```
#ifndef EVLOOP_HDR
#define EVLOOP_HDR

#define EVLOOP_POLL_READ      0x01
#define EVLOOP_POLL_WRITE     0x02
#define EVLOOP_POLL_HUP       0x04
#define EVLOOP_POLL_POLLPRI   0x08

struct evloop_ctx;
struct evloop_event;

typedef void(*evloop_fd_cb)(int fd, short revents, void *data);

// creates a new event loop context
int evloop_create(struct evloop_ctx **ctx);
// starts the event loop context, running a continuous loop inside
int evloop_run(struct evloop_ctx *ctx);
// pause the event loop
int evloop_halt(struct evloop_ctx *ctx);
// clean up and remove the event loop context
int evloop_destroy(struct evloop_ctx *ctx);

// create a fd event and add it to the evloop context ctx
int evloop_event_add_fd(struct evloop_ctx *ctx, struct evloop_event **ev, int fd,
                        int events, evloop_cb cb, void *data);
// clean up and remove the event
int evloop_event_remove(struct evloop_event *ev);

#endif
```

3.3.4 Example

```
#include "evloop.h"
#include "nsock.h"

void socket_cb(int fd, short revents, void *data)
{
        int ret = 0;
        char buf[1024] = "";
        char reply[1024] = "";

        ret = nsock_read(fd, buf, 1024);
        // ...

        // process and build reply

        ret = nsock_write(fd, reply, strlen(reply));
        // ...

        return;
}

int main(int argc, char **argv)
{
        int ret = 0;
        int sfd = -1;
        struct evloop_ctx *evlctx = NULL;
        struct evloop_event *sev = NULL;

        sfd = nsock_open("192.168.1.100", 5555);
        // ...

        ret = evloop_create(&evlctx);
        // ...

        ret = evloop_event_add(evlctx, &sev, sfd, EVLOOP_POLL_READ, socket_cb, NULL);
        // ...

        return evloop_run(evlctx);
}
```

3.3.5 Questions

- What open source libraries exist that provide event loop features?
- How to add support for timers in the API?
- How to add support for signals in the API?

CHAPTER 4

IoT

4.1 Digital Input Counter

4.1.1 Concepts

- Digital inputs and conversions
- Linux kernel GPIO interface
- Reporting data in JSON format
- TLV (Type-Length-Value) protocol

4.1.2 Assignment

- Main context: create an application that counts the pulses on a digital input pin
- Configure a digital pin as input and apply test pulses to this pin
- Periodically write the count values to a report file
- Set up a TCP server socket on port for user connections to get counter report values
- All report values shall be JSON encoded
- Provide command line arguments to set digital pin, TCP port number, loglevel, ...

4.1.3 Hints

- Set up the input pin for reading inside a `epoll()` loop using the `POLLPRI` event flag
- Suggested JSON report format:

```
{
    "timestamp" : "2017/10/18 14:40:35",
    "pin-id" : "/sys/class/gpio/gpio20/value",
    "count" : 511
}
```

- An other digital (gpio) pin can be configured as output and used in a test script as pulse input
- Suggested help message for program:

```
counter: counts the pulses on a digital input
    -h                  Show help
    -l "level"          Log level to be used (console and file logging)
    -i "id"             Identifier for digital input
    -t "period"         Interval to make a counter report in seconds
    -r "path"           Full path to the report file, default current directory
    -p "port"           TCP port number to use, default 5555
    -u "max-nr"         Max number of connected users, default 10
```

- Capture signal `SIGTERM` and `SIGINT` to gracefully shutdown the process
- Use a small TLV-like protocol to report the counter values over TCP socket

4.1.4 Legacy Linux GPIO interface

- To use a GPIO pin, it must first be exported:

```
# echo XY > /sys/class/gpio/export      # with XY the desired pin number
```

- The pin number can be calculated from the pin bank name and number:

```
(position_bank_letter_in_alphabet - 1) * 32 + pin_number
```

- For example:

```
PA17: (1 - 1) * 32 + 17 =   0 + 17 = 17
PH18: (8 - 1) * 32 + 18 = 224 + 18 = 242
```

- The IO direction must be set on the pin:
  - **WARNING:** the default direction is *in*, but set it explicitly!

```
# echo in  > /sys/class/gpio/gpio23/direction
# echo out > /sys/class/gpio/gpio24/direction
```

- The value of the GPIO pin can be read (input) or be written (output):

```
# cat /sys/class/gpio/gpio23/value
# echo 1 > /sys/class/gpio/gpio24/value
```

4.1.5 New Linux GPIO interface

- In kernel 4.8, a new GPIO API interface was released
- Check the kernel source: `tools/gpio/*`
- https://kernel-recipes.org/en/2018/talks/new-gpio-interface-for-user-space/

4.1.6 Questions

- A digital input signal can bounce on transition (e.g. mechanical relay), how to debounce such a signal in software?
- What are possible digital input isolation techniques?
- Why use a TLV-like protocol over a TCP connection?

4.1.7 Assignments (optional)

- Provide an `evloop` API extension for signals
- Provide an `evloop` API extension for timers
- Provide an `evloop` API extension for simple sockets
- Use the new kernel GPIO interface

4.2 HTTP Server Module

4.2.1 References

- https://en.wikipedia.org/wiki/Request\T1\textendash{}response
- https://github.com/nodejs/http-parser
- ...

4.2.2 Concept: M2M and Messaging Patterns

- **M2M**: machine-to-machine communication
- Machine can be anything: sensors, lights, refrigerators, cars, manufacturing robots, irrigation systems, ...
- Need for an infrastructure that connects components and services, ideally in a loosely coupled manner in order to maximize scalability
- **Messaging pattern:** a network-oriented architectural pattern which describes how two different parts of a message passing system communicate with each other
- For example: HTTP GET, ... TODO
- To implement a messaging pattern and perform machine-to-machine communication, an application protocol is used
- The application protocol is usually implemented in user space on top of an in kernel transport mechanism (TCP, UDP, Unix Domain Socket, serial line, wireless link, ...)
- Application protocol definition and encapsulation (example UDP):

4.2.3 Concept: Request/reply messaging pattern

- Aka REQREP, Request/response
- Messaging pattern that allows ...

- Form of asynchronous client-to-service or service-to-service communication that enables event-driven architectures
- Allows decoupling of applications into smaller, independent building blocks
- This increases performance, reliability and scalability
- In most cases, a client needs a separate connection needs to be established to every publisher
- ... TODO

4.2.4 Assignment

- Create a **HTTP server** module
- Use raw POSIX sockets
- Use an open source HTTP parser
- Create a header file for reuse in other programs

4.2.5 Module API

TODO

4.2.6 Hints

- Use an event loop (or poll mechanism) for handling the socket file descriptors

4.2.7 Questions

- What are the most popular REQREP (or messaging in general) protocols?

4.3 NATS Client Module

4.3.1 References

- https://en.wikipedia.org/wiki/Publish\T1\textendash{}subscribe_pattern
- https://en.wikipedia.org/wiki/Message_broker
- https://aws.amazon.com/pub-sub-messaging/
- https://nats.io/blog/openiot/
- https://nats-io.github.io/docs/

4.3.2 Concept: Publish/subsribe messaging pattern

- Aka PUBSUB or Topic Broadcasting
- Messaging pattern that allows **publishers** to multicast (one-to-many) messages to zero or more **subscribers**
- Subscribers can subscribe to **specific topics**, allowing them to receive only messages that are relevant to them

- Form of asynchronous service-to-service communication that enables event-driven architectures
- Allows decoupling of applications into smaller, independent building blocks
- This increases performance, reliability and scalability
- A first topolgy is where a subscriber connects directly to a publisher
- Subscribers can be connected to multiple publishers
- But a separate connection needs to be established to every publisher
- The disadvantage is that the subscribers need to know all possible publishers

- A different topolgy is to use an intermediary central service between publishers and subscribers
- Called a **message broker** or event bus
- Publishers post messages to the broker and subscribers register subscriptions to the broker
- The broker uses a **store-and-forward mechanism** to route the messages with a certain topic
- A disadvantage is that there is an extra hop in the messaging network
- An other disadvantage is that the central broker can fail, which disrupts all messaging traffic

- The application protocol for PUBSUB patterns will include message types to:
  - Subscribe (and unsubscribe) to a topic from the peer or broker
  - Publish a message on a certain topic to all peers or a broker
  - Receive messages on the subscribed topics, the topic is usually included when receiving a message
- Application protocol example and encapsulation over TCP: receive the message *Hello World* under the topic *FOO.BAR*
- **Note:** the NATS protocol uses the symbols `\r\n` as escape sequences in the TCP stream to separate protocol units, these escape sequences have to be sent over the socket

4.3.3 Assignment

- Create a **NATS client protocol** module
- Provide support for the publish-subscribe part of protocol
- Use raw POSIX sockets

- Create a header file for reuse in other programs
- Provide a (re)connection procedure for the TCP sockets in a state machine

4.3.4 Module API

```
#ifndef NSOCK_HDR
#define NSOCK_HDR

struct nsock_ctx;

struct nsock_msg {
        char *topic;
        void *data;
        int data_len;
};

typedef void (*nsock_cb)(nsock_ctx *ctx, struct nsock_msg *msg, void *data);

int nsock_open(const char *address, int port, struct nsock_ctx **ctx);
int nsock_get_fd(struct nsock_ctx *ctx);
int nsock_process(struct nsock_ctx *ctx);
int nsock_subscribe(struct nsock_ctx *ctx, const char *topic,
                    struct nsock_cb cb, void *data);
int nsock_publish_msg(struct nsock_ctx *ctx, struct nsock_msg *msg);
int nsock_read_msg(struct nsock_ctx *ctx, struct nsock_msg *msg);

#endif
```

4.3.5 Hints

- Use an event loop (or poll mechanism) for handling the socket file descriptors

4.3.6 Questions

- What are the most popular PUBSUB (or messaging in general) protocols?
- What other PUBSUB brokers exist? And wich protocol do they use?

CHAPTER 5

Appendices

5.1 Extra Material

5.1.1 References

- Bootlin slides, chapter *References*
- Bootlin Buildroot Training: http://free-electrons.com/doc/training/buildroot/buildroot-slides.pdf
- Main Buildroot documentation: https://buildroot.uclibc.org/downloads/manual/manual.html
- Main official kernel documentation: https://www.kernel.org/doc/

5.1.2 Books

- **Building Embedded Linux Systems**, Karim Yaghmour et al, 2009, O'Reilly Media. Older book, but good overview and covers older technologies (MTD) in detail.
- **Mastering Embedded Linux Programming**, Chris Simmonds, 2015, Packt Publishing. Good overview of the basics, without too much unnecessary details.

5.1.3 Conference Videos

- A tour of the ARM architecture and its Linux support
- Anatomy of Cross-Compilation Toolchains
- Filesystem Considerations for Embedded Devices
- Understanding a Real-Time System
- Tutorial: Building the Simplest Possible Linux System
- Boxes in Boxes: virtualization and containerization the context of embedded routers

- Device Tree: Past, Present, and Future
- Embedded Linux Booting Process
- Understanding a Real-Time System (more than just a kernel) - Steven Rostedt

5.1.4 Articles

- Initramfs articles R. Landley:
  - https://landley.net/writing/rootfs-intro.html
  - https://landley.net/writing/rootfs-howto.html
  - https://landley.net/writing/rootfs-programming.html

5.2 Docker Basics

5.2.1 Docker Lingo

- Images
- Containers
- Image layers
- Ephemeral
- Volumes
- Network
- Port forwarding
- Docker files

5.2.2 Simple Demo

1. Download Ubuntu image:

```
user@host: docker pull ubuntu:latest
```

2. List images:

```
user@host: docker image ls
```

3. Simple temporary container:

```
user@host: docker run -ti --rm --name test1 ubuntu:latest /bin/sh
```

4. List instances:

```
user@host: docker ps -a
```

5. Simple daemonized container:

```
user@host: docker run -d -ti --name test2 ubuntu:latest /bin/sh
```

6. Attach terminal:

```
user@host: docker attach test2
```

7. Stop container:

```
user@host: docker stop test2
```

8. Start container:

```
user@host: docker start test2
```

5.2.3 Web Server Demo

1. nginx sample config as http directory listing:

```
daemon off;
user nginx;
worker_processes 1;

error_log /var/log/nginx/error.log warn;
pid /var/run/nginx.pid;

events {
    worker_connections  1024;
}

http {
    include /etc/nginx/mime.types;
    default_type application/octet-stream;
    sendfile on;
    keepalive_timeout  65;

    server {
        listen 80;
        location / {
                root /usr/share/nginx/data;
                autoindex on;
        }
    }
}
```

2. Dockerfile: ubuntu + nginx + default config file

```
# base container image
FROM ubuntu:18.04

# install packages
RUN export DEBIAN_FRONTEND=noninteractive && \
    apt update --fix-missing && apt upgrade -y && \
    apt install -y sudo locales nginx

# generate locale
```

(continues on next page)

(continued from previous page)

```
RUN locale-gen en_US.UTF-8 && locale-gen --no-purge --lang en_US.UTF-8

# create a nginx user
RUN useradd -M --home /nonexistent --shell /usr/sbin/nologin nginx

# set default configuration
COPY nginx.conf /etc/nginx/nginx.conf

# default command at startup is nginx daemon
ENTRYPOINT [ "nginx" ]
```

3. Build container image from Dockerfile:

```
user@host: docker build -t nginx-http-share .
```

4. Start web server service:

```
user@host: docker run --rm -ti -p 80:80 \
           -v </full/path/to/share/>:/usr/share/nginx/data nginx-http-share
```

5. The web server is accessible on localhost port 80

5.2.4 References

- https://docs.docker.com/reference/
- https://www.youtube.com/watch?v=UV3cw4QLJLs (watch at 1.5x speed)
- http://takacsmark.com/getting-started-with-docker-in-your-project-step-by-step-tutorial/
- https://www.youtube.com/watch?v=6Er8MAvTWlI
- https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers/
- https://12factor.net/processes https://docs.docker.com/engine/tutorials/dockervolumes/
- https://deis.com/blog/2016/docker-storage-introduction/
- https://learning-continuous-deployment.github.io/docker/container/volumes/2015/05/22/persistent-data-with-docker/
- https://github.com/wsargent/docker-cheat-sheet
- https://blog.phusion.nl/2015/01/20/docker-and-the-pid-1-zombie-reaping-problem/

5.3 Docker for Buildroot Development

5.3.1 Dockerfile

```
# base container image
FROM ubuntu:18.04

# install packages for development
RUN export DEBIAN_FRONTEND=noninteractive && \
    apt-get update --fix-missing && apt upgrade -y && \
```

(continues on next page)

(continued from previous page)

```
    apt install -y sudo build-essential libncurses5-dev file git \
    bc rsync unzip python cpio wget vim procps silversearcher-ag \
    ctags man strace netcat tmux gperf bison flex texinfo locales \
    help2man gawk libarchive-zip-perl gdb tree cmake python3

# generate locale
RUN locale-gen en_US.UTF-8 && locale-gen --no-purge --lang en_US.UTF-8

# create a dev user
RUN useradd -m dev && adduser dev sudo && (echo 'dev:dev' | chpasswd)

# set TERM variable to fix ncurses flickering
ENV TERM=xterm-color

# set start user and workdir
USER dev
WORKDIR /home/dev

# default command at startup is bash shell
CMD /bin/bash
```

5.3.2 Container Commands

1. Create container image:

```
user@host: docker build -t brdev .
```

2. Launch a persistant container for Buildroot development (`brdev`):

```
user@host: docker run -ti -v ${HOME}/dockershare/:/home/dev/share/ \
                  --name brdev brdev
```

5.4 SBC - Beagle Bone Green

5.4.1 Specs

- TI AM335x 1GHz ARM Cortex-A8 SoC
- NEON floating-point accelerator
- 512 MB DDR3 RAM
- 4GB eMMC on-board flash
- Micro SD card slot
- 2x 46-pin IO headers
- 1x RJ45 Ethernet connector
- 1x USB2 port
- 1x micro USB port
- 2x Grove connectors (I2C and UART)

5.4.2 References

- Product page (1): https://beagleboard.org/green
- Product page (2): http://wiki.seeedstudio.com/BeagleBone_Green/
- Debian image (Beagle Bone Black): https://beagleboard.org/latest-images

5.4.3 Serial Line

- Separate debug (serial) header

Pin	Function
1	GND
2	CTS#
3	VCC
4	TXD
5	RXD
6	RTS#

- The default Linux UART serial line parameters:
  - Baudrate 115200
  - Databits 8
  - No parity bit
  - Stopbit 1

5.4.4 Interesting Links

- http://derekmolloy.ie/beaglebone/
- http://www.bootembedded.com/beagle-bone-black/building-embedded-linux-system-using-mainline-kernel-for-beaglebone-blac
- https://jumpnowtek.com/beaglebone/Beaglebone-Black-U-Boot-Notes.html
- https://wiki.beyondlogic.org/index.php?title=BeagleBoneBlack_Upgrading_uBoot
- https://diydrones.com/profiles/blogs/booting-up-a-beaglebone-black
- https://www.twam.info/hardware/beaglebone-black/u-boot-on-beaglebone-black
- https://www.digikey.com/eewiki/display/linuxonarm/BeagleBone+Black
- https://gist.github.com/vsergeev/2391575
- https://wiki.beyondlogic.org/index.php/BeagleBoneBlack_Building_Kernel
- http://mkaczanowski.com/embedded-development-with-qemu-beagleboard-yocto-angstrom-buildroot-where-to-begin/
- https://www.nexlab.net/product/mosso/
- https://www.youtube.com/watch?v=SaIpz00lE84
- https://elinux.org/BeagleBone_Black_Extracting_eMMC_contents

5.5 SBC - Raspberry Pi 3

5.5.1 Specs

- Broadcom BCM2837 1.2 GHz 64bit SoC
- Quad-core ARM Cortex A53 (ARMv8) cluster
- 1GB RAM LPDDR2 (900 MHz)
- Broadcom VideoCore IV GPU
- BCM43438 wireless LAN and Bluetooth Low Energy (BLE) on board
- 40-pin extended GPIO
- 4 USB 2 ports
- Full size HDMI
- Micro SD port

5.5.2 Pin Header

Function	Pin	Pin	Function
3.3V	1	2	5V
GPIO02 (SDA1)	3	4	5V
GPIO03 (SCL1)	5	6	Ground
GPIO04 (GPIO_GCLK)	7	8	(TXD0) GPIO14
Ground	9	10	(RXD0) GPIO15
GPIO17 (GPIO_GEN0)	11	12	(GPIO_GEN1) GPIO18
GPIO27 (GPIO_GEN2)	13	14	Ground
GPIO22 (GPIO_GEN3)	15	16	(GPIO_GEN4) GPIO23
3.3V	17	18	(GPIO_GEN5) GPIO24
GPIO10 (SPI_MOSI)	19	20	Ground
GPIO09 (SPI_MISO)	21	22	(GPIO_GEN5) GPIO25
GPIO11 (SPI_CLK)	23	24	(SPI_CE0_N) GPIO08
Ground	25	26	(SPI_CE1_N) GPIO07
ID_SD (I2C ID EEPROM)	27	28	(I2C ID EEPROM) ID_SC
GPIO05	29	30	Ground
GPIO06	31	32	GPIO12
GPIO13	33	34	Ground
GPIO19	35	36	GPIO16
GPIO26	37	38	GPIO20
Ground	39	40	GPIO21

5.5.3 Serial Line

- TXD pin 8
- RXD pin 10
- Ground pin 6
- The default Linux UART serial line parameters:

  - Baudrate 115200
  - Databits 8
  - No parity bit
  - Stopbit 1

5.5.4 Broadcom Raspberry Pi Bootloader

- Proprietary bootloader from Broadcom (BCM) https://github.com/raspberrypi/firmware
- This repository contains:
  - Pre-compiled binaries Raspberry Pi kernel and modules
  - Userspace libraries
  - Bootloader/GPU firmware
- The bootloader resides in the `/boot` directory
- Bootloader sequence consists of 3 stages:
  - 1st stage: executed from code in ROM, loads 2nd stage in L2 cache memory from SD card
  - 2nd stage: executed from L2 cache, initializes RAM, loads 3rd stage (= GPU firmware) in RAM from SD card
  - 3rd stage: executed from RAM, loads kernel image and device tree blob from SD card

Bootloader stages

- 1st stage bootloader
  - Located in ROM on the SoC, not reprogrammable
  - Executed in small RISC core on the GPU
  - Mounts the 1st partition of the SD card
  - Partition must be **FAT32** or **FAT16** formatted
  - SD card is mandatory, but advantage is that board cannot be bricked
  - Then loads 2nd stage bootloader **bootcode.bin** in L2 cache memory
- 2nd stage bootloader (**bootcode.bin**)
  - Loaded from SD card's 1st partition
  - Loaded into L2 cache from GPU
  - Initializes SDRAM
  - Retrieves GPU firmware **start.elf** from 1st partition of the SD card
  - Programs the firmware and starts execution
- 3rd stage bootloader (**start.elf**)
  - Includes GPU firmware (remains in RAM after booting kernel)
  - Loaded from SD card's 1st partition
  - Parses configuration (**config.txt**)

  - Uses **fixup.dat** to configure SDRAM partition between GPU and CPU
  - Load **kernel.img** (default) and device tree blob into memory
  - Copy **cmdline.txt** content into memory
  - Starts up ARM CPU by releasing reset and kernel starts booting
- Conclusion: relevant files:
  - bootcode.bin
  - fixup.dat
  - start.elf
  - config.txt
  - cmdline.txt

Configuration

- https://www.raspberrypi.org/documentation/configuration/config-txt/
- http://elinux.org/RPiconfig
- Example configuration:

```
user@host: cat config.txt
kernel=kernel.img-4.9.x-armv7a
initramfs initrd.img-4.9.x-armv7a
enable_uart=1
dtparam=i2c_arm=on
disable_splash
```

- Example kernel command line:

```
user@host: cat cmdline.txt
dwc_otg.lpm_enable=0 console=ttyAMA0,115200 kgdboc=ttyAMA0,115200
console=tty1 root=/dev/mmcblk0p2 rootfstype=ext4 elevator=noop rootwait
rw smsc95xx.macaddr=b8:27:eb:57:c8:42
```

Comparison to U-Boot Bootloader

- U-Boot has more extended configuration (scripted)
- U-Boot has more advanced features (NFS boot, . . .)

Links

- Product page: https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/
- Raspbian: https://www.raspberrypi.org/downloads/raspbian/
- eLinux: http://elinux.org/RPi_Hub

5.6 Hardware Selection

5.6.1 Overview Single Board Computers (SBC)

Questions per platform

- mainline kernel support
- mainline uboot support
- serial console interface support
- video/audio optional
- CAN nice to have
- rpi header support
- multiple boot modes

Raspberry Pi 3 B+

- https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/
- SoC: Broadcom BCM2710
- pro:
  - already in use in current labs
  - community
- con:
  - no mainline kernel
  - proprietary boot loader
  - old(er) technologies (GPU is main chip)
  - hobby attitude
- price: 35 euro

Beagle Bone Green

- https://beagleboard.org/green
- http://wiki.seeedstudio.com/BeagleBone_Green/
- SoC: TI AM335x
- pro:
  - based on Beagle Bone Black
  - community support
  - mainline kernel
  - mainline uboot
  - on board eMMC

- con:
  - price vs performance
- price: 40 euro

ROC-RK3328-CC

- http://en.t-firefly.com/product/rocrk3328cc
- Soc: Rockchip RK3328
- pro:
  - mainline kernel (from 4.17)
  - mainline uboot
  - rpi compatible (full)
  - new(er) technologies
- con:
  - TBD
- price: 35 euro

AML-S905X-CC (Le Potato)

- https://libre.computer/products/boards/aml-s905x-cc/
- Soc: Amlogic S905X SoC
- pro:
  - mainline kernel
  - mainline uboot
  - rpi compatible (full)
  - community (http://linux-meson.com/doku.php)
- con:
  - TBD
- price: 32 euro

ALL-H3-CC (Tritium)

- https://libre.computer/products/boards/all-h3-cc/
- SoC: Allwinner Hx
- not released yet (!)
- price: 25 euro

BPI-M2+

- http://www.banana-pi.org/m2plus.html
- https://linux-sunxi.org/Sinovoip_Banana_Pi_M2%2B
- SoC: Allwinner H3
- pro:
  - mainline kernel
  - mainline uboot
  - EDU version without eMMC and wireless
  - rpi compatible (40 pin header)
  - USB OTG port
  - FEL mode (multiple boot modes)
- con:
  - power supply through barrel jack
- price: 28 euro, 25 euro (EDU)

PINE A64-LTS

- https://www.pine64.org/?product=pine-a64-lts
- http://linux-sunxi.org/Pine64
- SoC: Allwinner R18 (A64)
- also available as SOM + baseboard (!)
  - https://www.pine64.org/?product=sopine-a64-baseboard-combo
- pro:
  - mainline kernel
  - mainline uboot
  - rpi compatible (40 pin header)
  - no wireless by default
  - multiple IO (eMMC, GPIO, . . .)
  - FEL mode (multiple boot modes)
- con:
  - extensive kernel support (?)
- price: 32 euro

Toradex Colibri iMX7

- https://www.toradex.com/computer-on-modules/colibri-arm-family/nxp-freescale-imx7
- Soc: iMX7

- pro:
  - no video/audio interfaces
  - CAN available
  - multiple carrier boards
  - professional industrial SoC
- con:
  - price
  - community support
- base boards:
  - https://www.toradex.com/products/carrier-board/viola-carrier-board (21 euro)
  - https://www.toradex.com/products/carrier-boards/aster-carrier-board (52 euro)
- price: +/- 70 euro (without base board)

Allwinner Extra Links

- https://linux-sunxi.org/Linux_mainlining_effort#Work_In_Progress
- https://linux-sunxi.org/FEL
- https://linux-sunxi.org/FEL/USBBoot

5.6.2 Network Labs Hardware

- https://omnia.turris.cz/en/
- https://www.pcengines.ch/apu2.htm
- https://protectli.com/6-port/
- https://asrock.com/ipc/overview.asp?Model=NAS-9601
- https://asrock.com/ipc/overview.asp?Model=NAS-9602
- http://linuxgizmos.com/compact-six-port-net-appliance-features-dual-bypass-pairs/
- http://linuxgizmos.com/atom-c3000-based-net-appliance-targets-5g-vcpe-gear/
- http://linuxgizmos.com/atom-c3000-based-net-appliance-offers-eight-lan-ports/

CHAPTER 6

Indices and tables

- genindex
- modindex
- search