# Embedded Linux System Development

QEMU ARM variant

# Practical Labs

bootlin

https://bootlin.com

July 12, 2024

## About this document

Updates to this document can be found on https://bootlin.com/doc/training/embedded-linux-qemu.

This document was generated from LaTeX sources found on https://github.com/bootlin/training-materials.

More details about our training sessions can be found on https://bootlin.com/training.

## Copying this document



Corrections, suggestions, contributions and translations are welcome!

# Training setup

*Download files and directories used in practical labs*

## Install lab data

For the different labs in this course, your instructor has prepared a set of data (kernel images, kernel configurations, root filesystems and more). Download and extract its tarball from a terminal:

```
$ cd
$ wget https://bootlin.com/doc/training/embedded-linux-qemu/embedded-linux-qemu-labs.tar.xz
$ tar xvf embedded-linux-qemu-labs.tar.xz
```

Lab data are now available in an `embedded-linux-qemu-labs` directory in your home directory. This directory contains directories and files used in the various practical labs. It will also be used as working space, in particular to keep generated files separate when needed.

## Update your distribution

To avoid any issue installing packages during the practical labs, you should apply the latest updates to the packages in your distro:

```
$ sudo apt update
$ sudo apt dist-upgrade
```

You are now ready to start the real practical labs!

## Install extra packages

Feel free to install other packages you may need for your development environment. In particular, we recommend to install your favorite text editor and configure it to your taste. The favorite text editors of embedded Linux developers are of course *Vim* and *Emacs*, but there are also plenty of other possibilities, such as Visual Studio Code[1], *GEdit*, *Qt Creator*, *CodeBlocks*, *Geany*, etc.

It is worth mentioning that by default, Ubuntu comes with a very limited version of the `vi` editor. So if you would like to use `vi`, we recommend to use the more featureful version by installing the `vim` package.

## More guidelines

Can be useful throughout any of the labs

- Read instructions and tips carefully. Lots of people make mistakes or waste time because they missed an explanation or a guideline.
- Always read error messages carefully, in particular the first one which is issued. Some people stumble on very simple errors just because they specified a wrong file path and didn't pay enough attention to the corresponding error message.
- Never stay stuck with a strange problem more than 5 minutes. Show your problem to your colleagues or to the instructor.
- You should only use the `root` user for operations that require super-user privileges, such as: mounting a file system, loading a kernel module, changing file ownership, configuring the network. Most regular tasks (such as downloading, extracting sources, compiling...) can be done as a regular user.

[1] This tool from Microsoft is Open Source! To try it on Ubuntu: `sudo snap install code --classic`

- If you ran commands from a root shell by mistake, your regular user may no longer be able to handle the corresponding generated files. In this case, use the `chown -R` command to give the new files back to your regular user.
  Example: `$ sudo chown -R myuser.myuser linux/`

# Building a cross-compiling toolchain

*Objective: Learn how to compile your own cross-compiling toolchain for the musl C library*

After this lab, you will be able to:

- Configure the *crosstool-ng* tool
- Execute *crosstool-ng* and build up your own cross-compiling toolchain

## Setup

Go to the `$HOME/embedded-linux-qemu-labs/toolchain` directory.

For this lab, you need a system or VM with a least 4 GB of RAM.

## Install needed packages

Install the packages needed for this lab:

```
$ sudo apt install build-essential git autoconf bison flex texinfo help2man gawk libtool-bin \
    libncurses5-dev unzip
```

## Getting Crosstool-ng

Let's download the sources of Crosstool-ng, through its git source repository, and switch to a commit that we have tested:

```
$ git clone https://github.com/crosstool-ng/crosstool-ng
$ cd crosstool-ng/
$ git checkout crosstool-ng-1.26.0
```

## Building and installing Crosstool-ng

As we are not building Crosstool-ng from a release archive but from a git repository, we first need to generate a `configure` script and more generally all the generated files that are shipped in the source archive for a release:

```
$ ./bootstrap
```

We can then either install Crosstool-ng globally on the system, or keep it locally in its download directory. We'll choose the latter solution. As documented at https://crosstool-ng.github.io/docs/install/#hackers-way, do:

```
$ ./configure --enable-local
$ make
```

Then you can get Crosstool-ng help by running

```
$ ./ct-ng help
```

## Configure the toolchain to produce

A single installation of Crosstool-ng allows to produce as many toolchains as you want, for different architectures, with different C libraries and different versions of the various components.

Crosstool-ng comes with a set of ready-made configuration files for various typical setups: Crosstool-ng calls them *samples*. They can be listed by using `./ct-ng list-samples`.

We will load the Cortex A9 sample . Load it with the `./ct-ng` command.

Then, to refine the configuration, let's run the `menuconfig` interface:

```
$ ./ct-ng menuconfig
```

In `Path and misc options`:

- If not set yet, enable `Try features marked as EXPERIMENTAL`

In `Toolchain options`:

- Set `Tuple's vendor string` (`TARGET_VENDOR`) to `training`.
- Set `Tuple's alias` (`TARGET_ALIAS`) to `arm-linux`. This way, we will be able to use the compiler as `arm-linux-gcc` instead of `arm-training-linux-musleabihf-gcc`, which is much longer to type.

In `Operating System`:

- Set `Version of linux` to the closest, but older version to 6.1. It's important that the kernel headers used in the toolchain are not more recent than the kernel that will run on the board (v6.1).

In `C-library`:

- If not set yet, set `C library` to `musl` (`LIBC_MUSL`)
- Keep the default version that is proposed

In `C compiler`:

- Set `Version of gcc` to `13.2.0`.
- Make sure that `C++` (`CC_LANG_CXX`) is enabled

In `Debug facilities`:

- Remove all options here. Some debugging tools can be provided in the toolchain, but they can also be built by filesystem building tools.

Explore the different other available options by traveling through the menus and looking at the help for some of the options. Don't hesitate to ask your trainer for details on the available options. However, remember that we tested the labs with the configuration described above. You might waste time with unexpected issues if you customize the toolchain configuration.

## Produce the toolchain

Nothing is simpler:

```
$ ./ct-ng build
```

The toolchain will be installed by default in `$HOME/x-tools/`. That's something you could have changed in Crosstool-ng's configuration.

And wait!

## Testing the toolchain

You can now test your toolchain by adding `$HOME/x-tools/arm-training-linux-musleabihf/bin/` to your `PATH` environment variable and compiling the simple `hello.c` program in your main lab directory with `arm-linux-gcc`:

```
$ arm-linux-gcc -o hello hello.c
```

You can use the `file` command on your binary to make sure it has correctly been compiled for the ARM architecture.

Did you know that you can still execute this binary from your x86 host? To do this, install the QEMU user emulator, which just emulates target instruction sets, not an entire system with devices:

```
$ sudo apt install qemu-user
```

Now, try to run QEMU ARM user emulator:

```
$ qemu-arm hello
qemu-arm: Could not open '/lib/ld-musl-armhf.so.1': No such file or directory
```

What's happening is that `qemu-arm` is missing the shared library loader (compiled for ARM) that this binary relies on. Let's find it in our newly compiled toolchain:

```
$ find ~/x-tools -name ld-musl-armhf.so.1
```

```
/home/tux/x-tools/arm-training-linux-musleabihf/arm-training-linux-musleabihf/sysroot/lib/
ld-musl-armhf.so.1
```

We can now use the `-L` option of `qemu-arm` to let it know where shared libraries are:

```
$ qemu-arm -L ~/x-tools/arm-training-linux-musleabihf/arm-training-linux-musleabihf/sysroot \
    hello
```

```
Hello world!
```

## Cleaning up

*Do this only if you have limited storage space. In case you made a mistake in the toolchain configuration, you may need to run Crosstool-ng again, keeping generated files would save a significant amount of time.*

To save about 9 GB of storage space, do a `./ct-ng clean` in the Crosstool-NG source directory. This will remove the source code of the different toolchain components, as well as all the generated files that are now useless since the toolchain has been installed in `$HOME/x-tools`.

# Bootloader - U-Boot

*Objectives: Compile and install the U-Boot bootloader, use basic U-Boot commands, set up TFTP communication with the development workstation.*

## Setup

Go to the `$HOME/embedded-linux-qemu-labs/bootloader` directory.

Install the `qemu-system-arm` package. In this lab and in the following ones, we will use a QEMU emulated ARM Vexpress Cortex A9 board.

## Configuring and building U-Boot

Download U-Boot:

```
$ git clone https://gitlab.denx.de/u-boot/u-boot
$ cd u-boot
$ git checkout v2023.01
```

Now configure U-Boot to support the ARM Vexpress Cortex A9 board (`vexpress_ca9x4_defconfig`).

Get an understanding of U-Boot's configuration and compilation steps by reading the `README` file, and specifically the *Building the Software* section.

Basically, you need to specify the cross-compiler prefix (the part before `gcc` in the cross-compiler executable name):

```
$ export CROSS_COMPILE=arm-linux-
```

Now that you have a valid initial configuration, run `make menuconfig` to further edit your bootloader features:

- In the `Environment` submenu, we will configure U-Boot so that it stores its environment inside a file called `uboot.env` in a FAT filesystem on an MMC/SD card, as our emulated machine won't have flash storage:
  - Unset `Environment in flash memory` (`CONFIG_ENV_IS_IN_FLASH`)
  - Set `Environment is in a FAT filesystem` (`CONFIG_ENV_IS_IN_FAT`)
  - Set `Name of the block device for the environment` (`CONFIG_ENV_FAT_INTERFACE`): `mmc`
  - `Device and partition for where to store the environment in FAT` (`CONFIG_ENV_FAT_DEVICE_AND_PART`): `0:1`
    The above two settings correspond to the arguments of the `fatload` command.
- Also add support for the `editenv` (`CONFIG_CMD_EDITENV`) and `bootd` (which can be abbreviated as `boot`, `CONFIG_CMD_BOOTD`) that are not present in the default configuration for our board.

Install the following packages which should be needed to compile U-Boot for your board:

```
$ sudo apt install libssl-dev device-tree-compiler swig \
      python3-distutils python3-dev python3-setuptools
```

Finally, run

```
make
```

which will build U-Boot [2].

This generates several binaries, including `u-boot` and `u-boot.bin`.

## Testing U-Boot

Still in U-Boot sources, test that U-Boot works:

```
$ qemu-system-arm -M vexpress-a9 -m 128M -nographic -kernel u-boot
```

- `-M`: emulated machine
- `-m`: amount of memory in the emulated machine
- `-kernel`: allows to load the binary directly in the emulated machine and run the machine with it. This way, you don't need a first stage bootloader. Of course, you don't have this with real hardware.

Press a key before the end of the timeout, to access the U-Boot prompt.

You can then type the `help` command, and explore the few commands available.

Note: to exit QEMU, type `[Ctrl][a]` followed by `[h]` to see available commands. One of them is `[Ctrl][a]` followed by `[x]`, which allows to exit the emulator.

## SD card setup

Go back to the main `bootloader` directory.

We now need to add an SD card image to the QEMU virtual machine, in particular to get a way to store U-Boot's environment.

In later labs, we will also use such storage for other purposes (to store the kernel and device tree, root filesystem and other filesystems).

The commands that we are going to use will be further explained during the *Block filesystems* lectures.

First, using the `dd` command, create a 1 GB file filled with zeros, called `sd.img`:

```
$ dd if=/dev/zero of=sd.img bs=1M count=1024
```

This will be used by QEMU as an SD card disk image

Now, let's use the `cfdisk` command to create the partitions that we are going to use:

```
$ cfdisk sd.img
```

If `cfdisk` asks you to `Select a label type`, choose `dos`, as we don't really need a `gpt` partition table for our labs.

In the `cfdisk` interface, create three primary partitions, starting from the beginning, with the following properties:

- One partition, 64MB big, with the `FAT16` partition type. Mark this partition as bootable.
- One partition, 8 MB big[3], that will be used for the root filesystem. Due to the geometry of the device, the partition might be larger than 8 MB, but it does not matter. Keep the `Linux` type for the partition.
- One partition, that fills the rest of the SD card image, that will be used for the data filesystem. Here also, keep the `Linux` type for the partition.

[2] You can speed up the compiling by using the `-jX` option with `make`, where X is the number of parallel jobs used for compiling. Twice the number of CPU cores is a good value.

[3] For the needs of our system, the partition could even be much smaller, and 1 MB would be enough. However, with the 8 GB SD cards that we use in our labs, 8 MB will be the smallest partition that `cfdisk` will allow you to create.

Press `Write` when you are done.

We will now use the *loop* driver[4] to emulate block devices from this image and its partitions:

```
$ sudo losetup -f --show --partscan sd.img
```

- `-f`: finds a free loop device
- `--show`: shows the loop device that it used
- `--partscan`: scans the loop device for partitions and creates additional `/dev/loop<x>p<y>` block devices.

Also run `sudo dmesg` to confirm that 3 partitions were detected for the loop device selected by `losetup`:

```
[62778.965018] loop13: detected capacity change from 0 to 2097152
[62778.966862]  loop13: p1 p2 p3
```

Last but not least, format the first partition as FAT16 with a `boot` label:

```
$ sudo mkfs.vfat -F 16 -n boot /dev/loop<x>p1
```

The other partitions will be formated later.

Now, you can release the loop device:

```
$ sudo losetup -d /dev/loop<x>
```

## Testing U-Boot's environment

Start QEMU again, but this time with the emulated SD card (you can type the command in a single line):

```
$ qemu-system-arm -M vexpress-a9 -m 128M -nographic \
                -kernel u-boot/u-boot \
                -sd sd.img
```

Now, in the U-Boot prompt, make sure that you can set and store an environment variable:

```
$ setenv foo bar
$ saveenv
```

Type `reset` which reboots the board, and then check that the `foo` variable is still set:

```
$ printenv foo
```

## Setup networking between QEMU and the host

To load a kernel in the next lab, we will setup networking between the QEMU emulated machine and the host.

To do so, create a `qemu-myifup` script that will bring up a network interface between QEMU and the host. Here are its contents:

```
#!/bin/sh
/sbin/ip a add 192.168.0.1/24 dev $1
/sbin/ip link set $1 up
```

If necessary, modify the above network address range so that it doesn't collide with the one of the main company network.

Of course, make this script executable:

[4] Once again, this will be properly be explained during our *Block filesystems* lectures.

```
$ chmod +x qemu-myifup
```

As you can see, the host side will have the `192.168.0.1` IP address. We will use `192.168.0.100` for the target side. Of course, use a different IP address range if this conflicts with your local network.

Then, you will need root privileges to run QEMU this time, because of the need to bring up the network interface:

```
$ sudo qemu-system-arm -M vexpress-a9 -m 128M -nographic \
       -kernel u-boot/u-boot \
       -sd sd.img \
       -net tap,script=./qemu-myifup -net nic
```

Note the new net options:

- `-net tap`: creates a software network interface on the host side
- `-net nic`: adds a network device to the emulated machine

On the host machine, using the `ip a` command, check that there is now a `tap0` network interface with the expected IP address.

On the U-Boot command line, you will have to configure the environment variables for networking:

```
=> setenv ipaddr 192.168.0.100
=> setenv serverip 192.168.0.1
```

To make these settings permanent, save the environment:

```
=> saveenv
```

You can now test the connection to the host:

```
=> ping 192.168.0.1
```

It should finish by:

```
host 192.168.0.1 is alive
```

## Setting up the TFTP server

Let's install a TFTP server on your development workstation:

```
sudo apt install tftpd-hpa
```

Back in U-Boot, run `bdinfo`, which will allow you to find out that RAM starts at `0x60000000`. Therefore, we will use the `0x61000000` address to test *tftp*.

To test the TFTP connection, put a small text file in the directory exported through TFTP on your development workstation. Then, from U-Boot, do:

```
$ tftp 0x61000000 textfile.txt
```

The `tftp` command should have downloaded the `textfile.txt` file from your development workstation into the board's memory at location `0x61000000`.

You can verify that the download was successful by dumping the contents of the memory:

```
=> md 0x61000000
```

## Rescue binary

If you have trouble generating binaries that work properly, or later make a mistake that causes you to lose your bootloader binary, you will find a working version under `data/` in the current lab directory.

# Fetching Linux kernel sources

*Objective: learn how to fetch the Linux kernel sources from git, from both the master and stable branches.*

After this lab, you will be able to:

- Get the kernel sources from git, using the official Linux source tree.
- Fetch the sources for the stable Linux releases, by declaring a remote tree and getting stable branches from it.

## Setup

Create the `$HOME/embedded-linux-qemu-labs/kernel` directory and go into it.

Since the Linux kernel git repository is huge, our goal here is to start downloading it right now, before starting the lectures about the Linux kernel.

## Cloning the mainline Linux tree

To begin working with the Linux kernel sources, we need to clone its reference git tree, the one managed by Linus Torvalds.

However, this requires downloading more than 2.8 GB of data. If you are running this command from home, or if you have very fast access to the Internet at work (and if you are not 256 participants in the training room), you can do it directly by connecting to https://git.kernel.org:

```
git clone https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux
cd linux
```

If Internet access is not fast enough and if multiple people have to share it, your instructor will give you a USB flash drive with a `tar.gz` archive of a recently cloned Linux source tree.

You will just have to extract this archive in the current directory, and then pull the most recent changes over the network:

```
tar xf linux-git.tar.gz
cd linux
git checkout master
git pull
```

Of course, if you directly ran `git clone`, you won't have to run `git pull`, as `git clone` already retrieved the latest changes. You may need to run `git pull` in the future though, if you want to update a newer Linux version.

## Accessing stable releases

The Linux kernel repository from Linus Torvalds contains all the main releases of Linux, but not the stable versions: they are maintained by a separate team, and hosted in a separate repository.

We will add this separate repository as another *remote* to be able to use the stable releases:

```
git remote add stable https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux
git fetch stable
```

As this still represents many git objects to download (2.4 GiB when 6.9 was the latest version), if you are using an already downloaded git tree, your instructor will probably have fetched the *stable* branch ahead of time for you too. You can check by running:

```
git branch -a
```

We will choose a particular stable version in the next labs.

Now, let's continue the lectures. This will leave time for the commands that you typed to complete their execution (if needed).

# Kernel - Cross-compiling

*Objective: Learn how to cross-compile a kernel for an ARM target platform.*

After this lab, you will be able to:

- Checkout a stable version of the Linux kernel
- Set up a cross-compiling environment
- Cross compile the kernel for the QEMU ARM Versatile Express for Cortex-A9
- Use U-Boot to download the kernel
- Check that the kernel you compiled starts the system

## Setup

Stay in the `$HOME/embedded-linux-qemu-labs/kernel` directory.

## Choose a particular stable version of Linux

We will use `linux-6.1.x`, which this lab was tested with.

First, let's get the list of branches we have available:

```
cd linux
git branch -a
```

As we will do our labs with the Linux 6.1, the remote branch we are interested in is `remotes/stable/linux-6.1.y`.

First, execute the following command to check which version you currently have:

```
make kernelversion
```

You can also open the `Makefile` and look at the beginning of it to check this information.

Now, let's create a local branch starting from that remote branch:

```
git checkout stable/linux-6.1.y
```

Check the version again using the `make kernelversion` command to make sure you now have a 6.1.x version.

## Cross-compiling environment setup

To cross-compile Linux, you need to have a cross-compiling toolchain. We will use the cross-compiling toolchain that we previously produced, so we just need to make it available in the PATH:

```
$ export PATH=$HOME/x-tools/arm-training-linux-musleabihf/bin:$PATH
```

Also, don't forget to either:

- Define the value of the `ARCH` and `CROSS_COMPILE` variables in your environment (using `export`)
- **Or** specify them on the command line at every invocation of `make`, i.e.: `make ARCH=... CROSS_COMPILE=... <target>`

## Linux kernel configuration

By running `make help`, look for the proper Makefile target to configure the kernel for your processor.

In course case, use the configuration for the ARM Vexpress boards (`vexpress_defconfig`).

So, apply this configuration, and then run `make menuconfig`.

- Disable `CONFIG_GCC_PLUGINS` if it is set. This will skip building special *gcc* plugins, which would require extra dependencies for the build. Also start `make menuconfig` to add `CONFIG_DEVTMPFS_MOUNT` to your configuration.

## Cross compiling

You're now ready to cross-compile your kernel. Simply run:

```
$ make
```

and wait a while for the kernel to compile. Don't forget to use `make -j<n>` if you have multiple cores on your machine!

Look at the kernel build output to see which file contains the kernel image.

Also look in the Device Tree Source directory to see which `.dtb` files got compiled. Find which `.dtb` file corresponds to your board.

## Load and boot the kernel using U-Boot

As we are going to boot the Linux kernel from U-Boot, we need to set the `bootargs` environment corresponding to the Linux kernel command line:

```
=> setenv bootargs console=ttyAMA0
=> saveenv
```

We will use TFTP to load the kernel image on the board:

- On your workstation, copy the `zImage` and DTB (`vexpress-v2p-ca9.dtb`) to the directory exposed by the TFTP server.
- On the target (in the U-Boot prompt), load `zImage` from TFTP into RAM:

  ```
  => tftp 0x61000000 zImage
  ```

- Now, also load the DTB file into RAM:

  ```
  => tftp 0x62000000 vexpress-v2p-ca9.dtb
  ```

- Boot the kernel with its device tree:

  ```
  => bootz 0x61000000 - 0x62000000
  ```

You should see Linux boot and finally panicking. This is expected: we haven't provided a working root filesystem for our device yet.

You can now automate all this every time the board is booted or reset. Reset the board, and customize `bootcmd`:

```
=> setenv bootcmd 'tftp 0x61000000 zImage; tftp 0x62000000 vexpress-v2p-ca9.dtb; bootz
    0x61000000 - 0x62000000'
=> saveenv
```

Restart the board to make sure that booting the kernel is now automated.

# Tiny embedded system with BusyBox

*Objective: making a tiny yet full featured embedded system*

After this lab, you will:

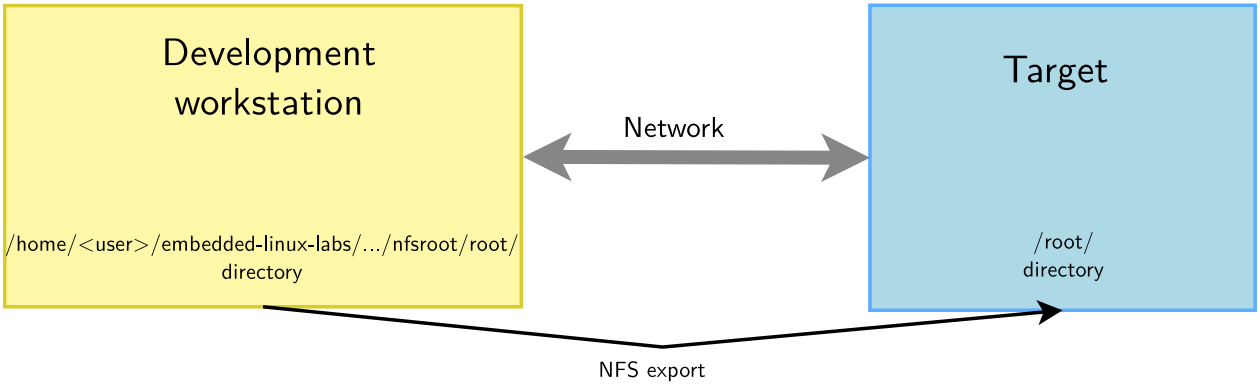
- be able to configure and build a Linux kernel that boots on a directory on your workstation, shared through the network by NFS.
- be able to create and configure a minimalistic root filesystem from scratch (ex nihilo, out of nothing, entirely hand made...) for your target board.
- understand how small and simple an embedded Linux system can be.
- be able to install BusyBox on this filesystem.
- be able to create a simple startup script based on /sbin/init.
- be able to set up a simple web interface for the target.

## Lab implementation

While (s)he develops a root filesystem for a device, a developer needs to make frequent changes to the filesystem contents, like modifying scripts or adding newly compiled programs.

It isn't practical at all to reflash the root filesystem on the target every time a change is made. Fortunately, it is possible to set up networking between the development workstation and the target. Then, workstation files can be accessed by the target through the network, using NFS.

Unless you test a boot sequence, you no longer need to reboot the target to test the impact of script or application updates.




## Setup

Go to the `$HOME/embedded-linux-qemu-labs/tinysystem/` directory.

## Kernel configuration

We will re-use the kernel sources from our previous lab, in `$HOME/embedded-linux-qemu-labs/kernel/`.

In the kernel configuration built in the previous lab, verify that you have all options needed for booting the system using a root filesystem mounted over NFS. Also check that `CONFIG_DEVTMPFS_MOUNT` is enabled (we will explain it later in this lab). If necessary, rebuild your kernel.

## Setting up the NFS server

Create a `nfsroot` directory in the current lab directory. This `nfsroot` directory will be used to store the contents of our new root filesystem.

Install the NFS server by installing the `nfs-kernel-server` package if you don't have it yet. Once installed, edit the `/etc/exports` file as root to add the following line, assuming that the IP address of your board will be `192.168.0.100`:

```
/home/<user>/embedded-linux-qemu-labs/tinysystem/nfsroot 192.168.0.100(rw,no_root_squash,
no_subtree_check)
```

Of course, replace `<user>` by your actual user name.

Make sure that the path and the options are on the same line. Also make sure that there is no space between the IP address and the NFS options, otherwise default options will be used for this IP address, causing your root filesystem to be read-only.

Then, make the NFS server use the new configuration:

```
$ sudo exportfs -r
```

## Booting the system

First, boot the board to the U-Boot prompt. Before booting the kernel, we need to tell it that the root filesystem should be mounted over NFS, by setting some kernel parameters.

So add settings to the `bootargs` environment variable, **in just 1 line**:

```
=> setenv bootargs ${bootargs} root=/dev/nfs ip=192.168.0.100
   nfsroot=192.168.0.1:/home/<user>/embedded-linux-qemu-labs/tinysystem/nfsroot,nfsvers=3,tcp rw
```

Once again, replace `<user>` by your actual user name.

Of course, you need to adapt the IP addresses to your exact network setup. Save the environment variables (with `saveenv`).

Now, boot your system. The kernel should be able to mount the root filesystem over NFS:

```
VFS: Mounted root (nfs filesystem) on device 0:16.
```

If the kernel fails to mount the NFS filesystem, look carefully at the error messages in the console. If this doesn't give any clue, you can also have a look at the NFS server logs in `/var/log/syslog`.

However, at this stage, the kernel should stop because of the below issue:

```
[    7.476715] devtmpfs: error mounting -2
```

This happens because the kernel is trying to mount the *devtmpfs* filesystem in `/dev/` in the root filesystem. This virtual filesystem contains device files (such as `ttyS0`) for all the devices known to the kernel, and with `CONFIG_DEVTMPFS_MOUNT`, our kernel tries to automatically mount *devtmpfs* on `/dev`.

To address this, just create a `dev` directory under `nfsroot` and reboot.

Now, the kernel should complain for the last time, saying that it can't find an init application:

```
Kernel panic - not syncing: No working init found.  Try passing init= option to
kernel. See Linux Documentation/admin-guide/init.rst for guidance.
```

Obviously, our root filesystem being mostly empty, there isn't such an application yet. In the next paragraph, you will add BusyBox to your root filesystem and finally make it usable.

## Root filesystem with BusyBox

Download the sources of the latest BusyBox 1.36.x release:

```
git clone https://git.busybox.net/busybox
cd busybox/
git checkout 1_36_stable
```

Now, configure BusyBox with the configuration file provided in the `data/` directory (remember that the BusyBox configuration file is `.config` in the BusyBox sources).

Then, you can use `$ make menuconfig` to further customize the BusyBox configuration. At least, keep the setting that builds a static BusyBox. Compiling BusyBox statically in the first place makes it easy to set up the system, because there are no dependencies on libraries. Later on, we will set up shared libraries and recompile BusyBox.

Build BusyBox using the toolchain that you used to build the kernel.

Going back to the BusyBox configuration interface, check the installation directory for BusyBox[5]. Set it to the path to your `nfsroot` directory.

Now run `$ make install` to install BusyBox in this directory.

Try to boot your new system on the board. You should now reach a command line prompt, allowing you to execute the commands of your choice.

## Virtual filesystems

Run the `$ ps` command. You can see that it complains that the `/proc` directory does not exist. The ps command and other process-related commands use the `proc` virtual filesystem to get their information from the kernel.

From the Linux command line in the target, create the `proc`, `sys` and `etc` directories in your root filesystem.

Now mount the `proc` virtual filesystem. Now that `/proc` is available, test again the `ps` command.

Note that you can also now halt your target in a clean way with the `halt` command, thanks to `proc` being mounted[6].

## System configuration and startup

The first user space program that gets executed by the kernel is `/sbin/init` and its configuration file is `/etc/inittab`.

In the BusyBox sources, read details about `/etc/inittab` in the `examples/inittab` file.

Then, create a `/etc/inittab` file and a `/etc/init.d/rcS` startup script declared in `/etc/inittab`. In this startup script, mount the `/proc` and `/sys` filesystems.

Any issue after doing this?

## Starting the shell in a proper terminal

Before the shell prompt, you probably noticed the below warning message:

```
/bin/sh: can't access tty; job control turned off
```

This happens because the shell specified in the `/etc/inittab` file in started by default in `/dev/console`:

```
::askfirst:/bin/sh
```

[5] You will find this setting in `Settings -> Install Options -> Destination path for 'make install'`.

[6] `halt` can find the list of mounted filesystems in `/proc/mounts`, and unmount each of them in a clean way before shutting down.

When nothing is specified before the leading `::`, `/dev/console` is used. However, while this device is fine for a simple shell, it is not elaborate enough to support things such as job control (`[Ctrl][c]` and `[Ctrl][z]`), allowing to interrupt and suspend jobs.

So, to get rid of the warning message, we need `init` to run `/bin/sh` in a real terminal device:

```
ttyAMA0::askfirst:/bin/sh
```

Reboot the system and the message will be gone!

## Switching to shared libraries

Take the `hello.c` program supplied in the lab `data` directory. Cross-compile it for ARM, dynamically-linked with the libraries[7], and run it on the target.

You will first encounter a very misleading `not found` error, which is not because the `hello` executable is not found, but because something else was not found while trying to execute this executable.

You can find it by running `file hello` on the host:

```
hello: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV),
dynamically linked, interpreter /lib/ld-musl-armhf.so.1, not stripped
```

So, what's missing is the `/lib/ld-musl-armhf.so.1` executable, which is the dynamic linker required to execute any program compiled with shared libraries. Using the `find` command, look for this file in the toolchain install directory, and copy it to the `lib/` directory on the target.

Then, running the executable again and see that the loader executes and finds out which shared libraries are missing.

In our case with the Musl C library, the dynamic linker also contains the C library, so the program should execute fine, as no further shared libraries are required.

If you still get the same error message, just try again a few seconds later. Such a delay can be needed because the NFS client can take a little time (at most 30-60 seconds) before seeing the changes made on the NFS server.

Now that the small test program works, we are going to recompile BusyBox without the static compilation option, so that BusyBox takes advantage of the shared libraries that are now present on the target.

Before doing that, measure the size of the `busybox` executable.

Then, build BusyBox with shared libraries, and install it again on the target filesystem. Make sure that the system still boots and see how much smaller the `busybox` executable got.

## Implement a web interface for your device

Replicate `data/www/` to the `/www` directory in your target root filesystem.

Now, run the BusyBox http server from the target command line:

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

It will automatically background itself.

If you use a proxy, configure your host browser so that it doesn't go through the proxy to connect to the target IP address, or simply disable proxy usage. Now, test that your web interface works well by opening `http://192.168.0.100/index.html` on the host.

See how the dynamic pages are implemented. Very simple, isn't it?

Finish by adding the command that starts the web server to your startup script, so that it is always started on your target.

[7] Invoke your cross-compiler in the same way you did during the toolchain lab

# Going further

If you have time before the others complete their labs...

## Initramfs booting

Configure your kernel to include the contents of the `nfsroot` directory as an initramfs.

Before doing this, you will need to create an `init` link in the toplevel directory to `sbin/init`, because the kernel will try to execute `/init`.

You will also need to mount *devtmpfs* from the `rcS` script, it cannot be mounted automatically by the kernel when you're booting from an initramfs.

Note: you won't need to modify your `root=` setting in the kernel command line. It will just be ignored if you have an initramfs.

When this works, go back to booting the system through NFS. This will be much more convenient in the next labs.

# Accessing Hardware Devices

*Objective: learn how to access hardware devices.*

## Goals

Now that we have access to a command line shell thanks to a working root filesystem, we can now explore existing devices.

**Important:** *The manipulations in this lab are limited because we're working with an emulated platform, not with real hardware. It would be best to get your hands on the hardware platforms we support. Our instructions for such platforms really cover practising with internal and external hardware devices.*

## Setup

Go to the `$HOME/embedded-linux-qemu-labs/hardware` directory, which provides useful files for this lab.

However, we will go on booting the system through NFS, using the root filesystem built by the previous lab.

## Exploring /dev

Start by exploring `/dev` on your target system. Here are a few noteworthy device files that you will see:

- *Terminal devices*: devices starting with `tty`. Terminals are user interfaces taking text as input and producing text as output, and are typically used by interactive shells. In particular, you will find `console` which matches the device specified through `console=` in the kernel command line. You will also find the `ttyAMA0` device file.
- *Pseudo-terminal devices*: devices starting with `pty`, used when you connect through SSH for example. Those are virtual devices, but there are so many in `/dev` that we wanted to give a description here.
- MMC device(s) and partitions: devices starting with `mmcblk`. You should here recognize the MMC device(s) on your system and the associated partitions.

Don't hesitate to explore `/dev` on your workstation too and ask any questions to your instructor.

## Exploring /sys

The next thing you can explore is the *Sysfs* filesystem.

A good place to start is `/sys/class`, which exposes devices classified by the kernel frameworks which manage them.

For example, go to `/sys/class/net`, and you will see all the networking interfaces on your system, whether they are internal, external or virtual ones.

Find which subdirectory corresponds to the network connection to your host system, and then check device properties such as:

- `speed`: will show you whether this is a gigabit or hundred megabit interface.
- `address`: will show the device MAC address. No need to get it from a complex command!
- `statistics/rx_bytes` will show you how many bytes were received on this interface.

Don't hesitate to look for further interesting properties by yourself!

You can also check whether `/sys/class/thermal` exists and is not empty on your system. That's the thermal framework, and it allows to access temperature measures from the thermal sensors on your system.

Next, you can now explore all the buses (virtual or physical) available on your system, by checking the contents of `/sys/bus`.

In particular, go to `/sys/bus/mmc/devices` to see all the MMC devices on your system. Go inside the directory for the first device and check several files (for example):

- `serial`: the serial number for your device.
- `preferred_erase_size`: the preferred erase block for your device. It's recommended that partitions start at multiples of this size.
- `name`: the product name for your device. You could display it in a user interface or log file, for example.
- `date`: apparently the manufacturing date for the device.

Don't hesitate to spend more time exploring `/sys` on your system and asking questions to your instructor.

That's all for now!

# Filesystems - Block file systems

*Objective: configure and boot an embedded Linux system relying on block storage*

After this lab, you will be able to:

- Produce file system images.
- Configure the kernel to use these file systems
- Use the tmpfs file system to store temporary files
- Load the kernel and DTB from a FAT partition

## Goals

After doing the *A tiny embedded system* lab, we are going to copy the filesystem contents to the emulated SD card. The storage will be split into several partitions, and your QEMU emulated board will be booted from this SD card, without using NFS anymore.

## Setup

Throughout this lab, we will continue to use the root filesystem we have created in the `$HOME/embedded-linux-qemu-labs/tinysystem/nfsroot` directory, which we will progressively adapt to use block filesystems.

## Filesystem support in the kernel

Recompile your kernel with support for SquashFS and ext4[8].

Update your kernel image on the tftp server. We will only later copy the kernel to our FAT partition.

Boot your board with this new kernel and on the NFS filesystem you used in this previous lab.

Now, check the contents of `/proc/filesystems`. You should see that ext4 and SquashFS are now supported.

## Format the third partition

We are going to format the third partition of the SD card image with the ext4 filesystem, so that it can contain uploaded images.

Setup the loop device again:

```
$ sudo losetup -f --show --partscan sd.img
```

And then format the third partition:

```
$ sudo mkfs.ext4 -L data /dev/loop<x>p3
```

Now, mount this new partition on a directory on your host (you could create the `/mnt/data` directory, for example) and move the contents of the `/www/upload/files` directory (in your target root filesystem) into it. The goal is to use the third partition of the SD card as the storage for the uploaded images.

You can now unmount the partition and free the loop device:

[8] Basic configuration options for these filesystems will be sufficient. No need for things like extended attributes.

```
$ sudo umount /mnt/data
$ sudo losetup -d /dev/loop<x>
```

Now, restart QEMU and from the Linux command line and mount this third partition on `/www/upload/files`.

Once this works, modify the startup scripts in your root filesystem to do it automatically at boot time.

Reboot your target system again and with the `mount` command, check that `/www/upload/files` is now a mount point for the third SD card partition. Also make sure that you can still upload new images, and that these images are listed in the web interface.

## Adding a tmpfs partition for log files

For the moment, the upload script was storing its log file in `/www/upload/files/upload.log`. To avoid seeing this log file in the directory containing uploaded files, let's store it in `/var/log` instead.

Add the `/var/log/` directory to your root filesystem and modify the startup scripts to mount a `tmpfs` filesystem on this directory. You can test your `tmpfs` mount command line on the system before adding it to the startup script, in order to be sure that it works properly.

Modify the `www/cgi-bin/upload.cfg` configuration file to store the log file in `/var/log/upload.log`. You will lose your log file each time you reboot your system, but that's OK in our system. That's what `tmpfs` is for: temporary data that you don't need to keep across system reboots.

Reboot your system and check that it works as expected.

## Making a SquashFS image

We are going to store the root filesystem in a SquashFS filesystem in the second partition of the SD card.

In order to create SquashFS images on your host, you need to install the `squashfs-tools` package. Now create a SquashFS image of your NFS root directory.

Setup the loop device again, and using the `dd` command, copy the file system image to the second partition in the SD card image. Release the loop device.

## Booting on the SquashFS partition

In the U-boot shell, configure the kernel command line to use the second partition of the SD card as the root file system. Also add the `rootwait` boot argument, to wait for the SD card to be properly initialized before trying to mount the root filesystem. Since the SD cards are detected asynchronously by the kernel, the kernel might try to mount the root filesystem too early without `rootwait`.

Check that your system still works. Congratulations if it does!

## Store the kernel image and DTB on the SD card

Setup the loop device again, and mount the FAT partition in the SD card image (for example on `/mnt/boot`). Then copy the kernel image and Device Tree to it.

Unmount the FAT partition and release the loop device.

You now need to adjust the `bootcmd` of U-Boot so that it loads the kernel and DTB from the SD card instead of loading them from the network.

In U-boot, you can load a file from a FAT filesystem using a command like

```
=> fatload mmc 0:1 0x61000000 filename
```

Which will load the file named `filename` from the first partition of the device handled by the first MMC controller to the system memory at the address `0x61000000`.

Type `=> reset` in U-Boot to reboot the board and make sure that your system still boots fine.

# Third party libraries and applications

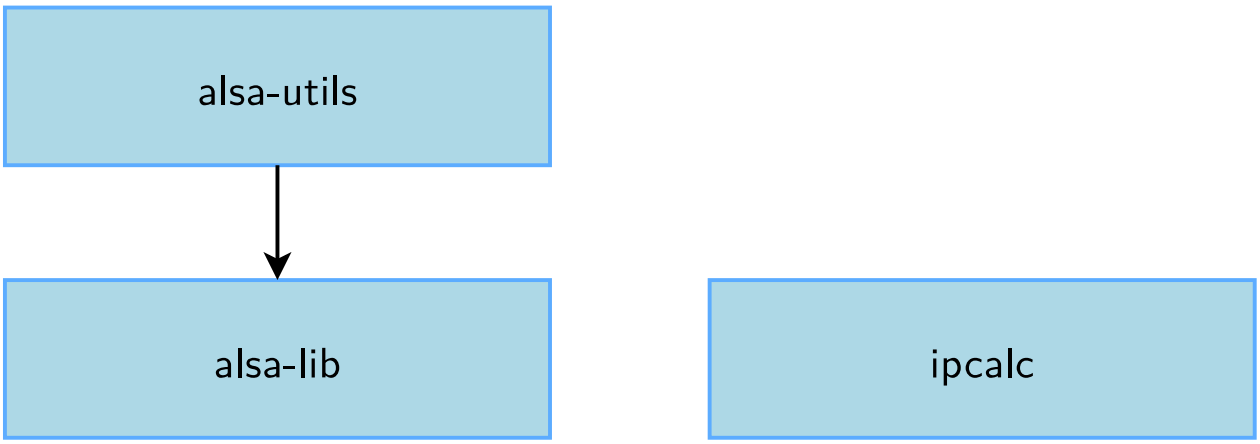
*Objective: Learn how to leverage existing libraries and applications: how to configure, compile and install them*

To illustrate how to use existing libraries and applications, we will extend the small root filesystem built in the *A tiny embedded system* lab to add the *ALSA* libraries and tools to run basic sound support tests. *ALSA* stands for *Advanced Linux Sound Architecture*, and is the Linux audio subsystem.

We'll see that manually re-using existing libraries is quite tedious, so that more automated procedures are necessary to make it easier. However, learning how to perform these operations manually will significantly help you when you face issues with more automated tools.

## Figuring out library dependencies

We're going to integrate the *alsa-utils* and *ipcalc* executables. In our case, the dependency chain for *alsa-utils* is quite simple, it only depends on the *alsa-lib* library. *ipcalc* is standalone and doesn't have any dependency.




Of course, all these libraries rely on the C library, which is not mentioned here, because it is already part of the root filesystem built in the *A tiny embedded system* lab. You might wonder how to figure out this dependency tree by yourself. Basically, there are several ways, that can be combined:

- Read the library documentation, which often mentions the dependencies;
- Read the help message of the `configure script` (by running `./configure --help`).
- By running the `configure` script, compiling and looking at the errors.

To configure, compile and install all the components of our system, we're going to start from the bottom of the tree with *alsa-lib*, then continue with *alsa-utils*. Then, we will also build *ipcalc*.

## Preparation

For our cross-compilation work, we will need two separate spaces:

- A *staging* space in which we will directly install all the packages: non-stripped versions of the libraries, headers, documentation and other files needed for the compilation. This *staging* space can be quite big, but will not be used on our target, only for compiling libraries or applications;
- A *target* space, in which we will only copy the required files from the *staging* space: binaries and

libraries, after stripping, configuration files needed at runtime, etc. This target space will take a lot less space than the *staging* space, and it will contain only the files that are really needed to make the system work on the target.

To sum up, the *staging* space will contain everything that's needed for compilation, while the *target* space will contain only what's needed for execution.

Create the `$HOME/embedded-linux-qemu-labs/thirdparty` directory, and inside, create two directories: `staging` and `target`.

For the target, we need a basic system with BusyBox and initialization scripts. We will re-use the system built in the *A tiny embedded system* lab, so copy this system in the target directory:

```
$ cp -a $HOME/embedded-linux-qemu-labs/tinysystem/nfsroot/* target/
```

Note that for this lab, a lot of typing will be required. To save time typing, we advise you to copy and paste commands from the electronic version of these instructions.

## Testing

Make sure the `target/` directory is exported by your NFS server to your board by modifying `/etc/exports` and restarting your NFS server.

Make your board boot from this new directory through NFS.

## alsa-lib

*alsa-lib* is a library supposed to handle the interaction with the ALSA subsystem. It is available at https://alsa-project.org. Download version 1.2.9, and extract it in `$HOME/embedded-linux-qemu-labs/thirdparty/`.

**Tip**: if the website for any of the source packages that we need to download in the next sections is down, a great mirror that you can use is http://sources.buildroot.net/.

Back to *alsa-lib* sources, look at the `configure` script and see that it has been generated by `autoconf` (the header contains a sentence like *Generated by GNU Autoconf 2.69*). Most of the time, `autoconf` comes with `automake`, that generates Makefiles from `Makefile.am` files. So *alsa-lib* uses a rather common build system. Let's try to configure and build it:

```
$ ./configure
$ make
```

If you look at the generated binaries, you'll see that they are x86 ones because we compiled the sources with gcc, the default compiler. This is obviously not what we want, so let's clean-up the generated objects and tell the `configure` script to use the ARM cross-compiler:

```
$ make clean
$ CC=arm-linux-gcc ./configure
```

Of course, the `arm-linux-gcc` cross-compiler must be in your `PATH` prior to running the configure script. The `CC` environment variable is the classical name for specifying the compiler to use.

Quickly, you should get an error saying:

```
checking whether we are cross compiling... configure: error: in `/home/mike/
embedded-linux-qemu-labs/thirdparty/alsa-lib-1.2.9':
configure: error: cannot run C compiled programs.
If you meant to cross compile, use `--host'.
See `config.log' for more details
```

If you look at the `config.log` file, you can see that the `configure` script compiles a binary with the cross-compiler and then tries to run it on the development workstation. This is a rather usual thing to do for a

`configure` script, and that's why it tests so early that it's actually doable, and bails out if not.

Obviously, it cannot work in our case, and the scripts exits. The job of the `configure` script is to test the configuration of the system. To do so, it tries to compile and run a few sample applications to test if this library is available, if this compiler option is supported, etc. But in our case, running the test examples is definitely not possible.

We need to tell the `configure` script that we are cross-compiling, and this can be done using the `--build` and `--host` options, as described in the help of the `configure` script:

```
System types:
  --build=BUILD configure for building on BUILD [guessed]
  --host=HOST cross-compile to build programs to run on HOST [BUILD]
```

The `--build` option allows to specify on which system the package is built, while the `--host` option allows to specify on which system the package will run. By default, the value of the `--build` option is guessed and the value of `--host` is the same as the value of the `--build` option. The value is guessed using the `./config.guess` script, which on your system should return `x86_64-pc-linux-gnu`. See https://www.gnu.org/software/autoconf/manual/html_node/Specifying-Names.html for more details on these options.

So, let's override the value of the `--host` option:

```
$ ./configure --host=arm-linux
```

Note that `CC` is not required anymore. It is implied by `--host`.

The `configure` script should end properly now, and create a Makefile.

However, there is one subtle issue to handle. We need to tell *alsa-lib* to disable a feature called alsa topology. *alsa-lib* will build fine but we will encounter some problems afterwards, during *alsa-utils* building. So you should configure *alsa-lib* as follows:

```
$ ./configure --host=arm-linux --disable-topology
```

Run the `make` command, which should run just fine.

Look at the result of compiling in `src/.libs`: a set of object files and a set of `libasound.so*` files.

The `libasound.so*` files are a dynamic version of the library. The shared library itself is `libasound.so.2.0.0`, it has been generated by the following command line:

```
$ arm-linux-gcc -shared conf.o confmisc.o input.o output.o async.o error.o dlmisc.o socket.o \
    shmarea.o userfile.o names.o -lm -ldl -lpthread -lrt -Wl,-soname -Wl,libasound.so.2 -o \
    libasound.so.2.0.0
```

And creates the symbolic links `libasound.so` and `libasound.so.2`.

```
$ ln -s libasound.so.2.0.0 libasound.so.2
$ ln -s libasound.so.2.0.0 libasound.so
```

These symlinks are needed for two different reasons:

- `libasound.so` is used at compile time when you want to compile an application that is dynamically linked against the library. To do so, you pass the `-lLIBNAME` option to the compiler, which will look for a file named `lib<LIBNAME>.so`. In our case, the compilation option is `-lasound` and the name of the library file is `libasound.so`. So, the `libasound.so` symlink is needed at compile time;
- `libasound.so.2` is needed because it is the *SONAME* of the library. *SONAME* stands for *Shared Object Name*. It is the name of the library as it will be stored in applications linked against this library. It means that at runtime, the dynamic loader will look for exactly this name when looking for the shared library. So this symbolic link is needed at runtime.

To know what's the *SONAME* of a library, you can use:

```
$ arm-linux-readelf -d libasound.so.2.0.0
```

and look at the `(SONAME)` line. You'll also see that this library needs the C library, because of the `(NEEDED)` line on `libc.so.0`.

The mechanism of `SONAME` allows to change the library without recompiling the applications linked with this library. Let's say that a security problem is found in the *alsa-lib* release that provides *libasound 2.0.0*, and fixed in the next *alsa-lib* release, which will now provide *libasound 2.0.1*.

You can just recompile the library, install it on your target system, change the `libasound.so.2` link so that it points to `libasound.so.2.0.1` and restart your applications. And it will work, because your applications don't look specifically for `libasound.so.2.0.0` but for the *SONAME* `libasound.so.2`.

However, it also means that as a library developer, if you break the ABI of the library, you must change the *SONAME*: change from `libasound.so.2` to `libasound.so.3`.

Finally, the last step is to tell the `configure` script where the library is going to be installed. Most `configure` scripts consider that the installation prefix is `/usr/local/` (so that the library is installed in `/usr/local/lib`, the headers in `/usr/local/include`, etc.). But in our system, we simply want the libraries to be installed in the `/usr` prefix, so let's tell the `configure` script about this:

```
$ ./configure --host=arm-linux --disable-topology --prefix=/usr
$ make
```

For this library, this option may not change anything to the resulting binaries, but for safety, it is always recommended to make sure that the prefix matches where your library will be running on the target system.

Do not confuse the *prefix* (where the application or library will be running on the target system) from the location where the application or library will be installed on your host while building the root filesystem.

For example, *libasound* will be installed in `$HOME/embedded-linux-qemu-labs/thirdparty/target/usr/lib/` because this is the directory where we are building the root filesystem, but once our target system will be running, it will see *libasound* in `/usr/lib`.

The prefix corresponds to the path in the target system and **never** on the host. So, one should **never** pass a prefix like `$HOME/embedded-linux-qemu-labs/thirdparty/target/usr`, otherwise at runtime, the application or library may look for files inside this directory on the target system, which obviously doesn't exist! By default, most build systems will install the application or library in the given prefix (`/usr` or `/usr/local`), but with most build systems (including *autotools*), the installation prefix can be overridden, and be different from the configuration prefix.

We now only have the installation process left to do.

First, let's make the installation in the *staging* space:

```
$ make DESTDIR=$HOME/embedded-linux-qemu-labs/thirdparty/staging install
```

Now look at what has been installed by *alsa-lib*:

- Some configuration files in `/usr/share/alsa`
- The headers in `/usr/include`
- The shared library and its libtool (`.la`) file in `/usr/lib`
- A pkgconfig file in `/usr/lib/pkgconfig`. We'll come back to these later

Finally, let's install the library in the *target* space:

1. Create the `target/usr/lib` directory, it will contain the stripped version of the library

2. Copy the dynamic version of the library. Only `libasound.so.2` and `libasound.so.2.0.0` are needed, since `libasound.so.2` is the *SONAME* of the library and `libasound.so.2.0.0` is the real binary:
   - `$ cp -a staging/usr/lib/libasound.so.2* target/usr/lib`
3. Measure the size of the `target/usr/lib/libasound.so.2.0.0` library before stripping.
4. Strip the library:
   - `$ arm-linux-strip target/usr/lib/libasound.so.2.0.0`
5. Measure the size of the `target/usr/lib/libasound.so.2.0.0` library library again after stripping. How many unnecessary bytes were saved?

Then, we need to install the *alsa-lib* configuration files:

```
$ mkdir -p target/usr/share
$ cp -a staging/usr/share/alsa target/usr/share
```

Now, we need to adjust one small detail in one of the configuration files. Indeed, `/usr/share/alsa/alsa.conf` assumes a UNIX group called `audio` exists, which is not the case on our very small system. So edit this file, and replace `defaults.pcm.ipc_gid audio` by `defaults.pcm.ipc_gid 0` instead.

And we're done with *alsa-lib*!

## Alsa-utils

Download *alsa-utils* from the ALSA offical webpage. We tested the lab with version 1.2.9. The `gettext` package is needed during the build so we have to install it.

```
sudo apt install gettext
```

Once uncompressed, we quickly discover that the *alsa-utils* build system is based on the *autotools*, so we will work once again with a regular `configure` script.

As we've seen previously, we will have to provide the prefix and host options and the CC variable:

```
$ ./configure --host=arm-linux --prefix=/usr
```

Now, we should quiclky get an error in the execution of the `configure` script:

```
checking for libasound headers version >= 1.2.5 (1.2.5)... not present.
configure: error: Sufficiently new version of libasound not found.
```

Again, we can check in `config.log` what the `configure` script is trying to do:

```
configure:15855: checking for libasound headers version >= 1.2.5 (1.2.5)
configure:15902: arm-linux-gcc -c -g -O2 conftest.c >&5
conftest.c:24:10: fatal error: alsa/asoundlib.h: No such file or directory
```

Of course, since *alsa-utils* uses *alsa-lib*, it includes its header file! So we need to tell the C compiler where the headers can be found: there are not in the default directory `/usr/include/`, but in the `/usr/include` directory of our *staging* space. The help text of the `configure` script says:

```
  CPPFLAGS    (Objective) C/C++ preprocessor flags, e.g. -I<include dir> if
              you have headers in a nonstandard directory <include dir>
```

Let's use it:

```
$ CPPFLAGS=-I$HOME/embedded-linux-qemu-labs/thirdparty/staging/usr/include \
    ./configure --host=arm-linux --prefix=/usr
```

Now, it should stop a bit later, this time with the error:

```
checking for snd_ctl_open in -lasound... no
configure: error: No linkable libasound was found.
```

The `configure` script tries to compile an application against *libasound* (as can be seen from the `-lasound` option): *alsa-utils* uses *alsa-lib*, so the `configure` script wants to make sure this library is already installed. Unfortunately, the `ld` linker doesn't find it. So, let's tell the linker where to look for libraries using the `-L` option followed by the directory where our libraries are (in `staging/usr/lib`). This `-L` option can be passed to the linker by using the `LDFLAGS` at configure time, as told by the help text of the `configure` script:

```
  LDFLAGS     linker flags, e.g. -L<lib dir> if you have libraries in a
              nonstandard directory <lib dir>
```

Let's use this `LDFLAGS` variable:

```
$ LDFLAGS=-L$HOME/embedded-linux-qemu-labs/thirdparty/staging/usr/lib \
    CPPFLAGS=-I$HOME/embedded-linux-qemu-labs/thirdparty/staging/usr/include \
    ./configure --host=arm-linux --prefix=/usr
```

Once again, it should fail a bit further down the tests, this time complaining about a missing *curses helper header*. *curses* or *ncurses* is a graphical framework to design UIs in the terminal. This is only used by *alsamixer*, one of the tools provided by *alsa-utils*, that we are not going to use. Hence, we can just disable the build of *alsamixer*.

Of course, if we wanted it, we would have had to build *ncurses* first, just like we built *alsa-lib*.

```
$ LDFLAGS=-L$HOME/embedded-linux-qemu-labs/thirdparty/staging/usr/lib \
    CPPFLAGS=-I$HOME/embedded-linux-qemu-labs/thirdparty/staging/usr/include \
    ./configure --host=arm-linux --prefix=/usr \
    --disable-alsamixer
```

Then, run the compilation with `make`. You should hit a final error:

```
Making all in po
make[2]: Entering directory '/home/tux/embedded-linux-qemu-labs/
thirdparty/alsa-utils-1.2.9/alsaconf/po'
mv: cannot stat 't-ja.gmo': No such file or directory
```

This can be fixed by disabling support for `alsaconf` too:

```
$ LDFLAGS=-L$HOME/embedded-linux-qemu-labs/thirdparty/staging/usr/lib \
    CPPFLAGS=-I$HOME/embedded-linux-qemu-labs/thirdparty/staging/usr/include \
    ./configure --host=arm-linux --prefix=/usr \
    --disable-alsamixer --disable-alsaconf
```

You can now run `make` again. It should work this time.

Let's now begin the installation process. Before really installing in the staging directory, let's install in a dummy directory, to see what's going to be installed (this dummy directory will not be used afterwards, it is only to verify what will be installed before polluting the staging space):

```
$ make DESTDIR=/tmp/alsa-utils/ install
```

The `DESTDIR` variable can be used with all Makefiles based on `automake`. It allows to override the installation directory: instead of being installed in the configuration prefix directory, the files will be installed in `DESTDIR/configuration-prefix`.

Now, let's see what has been installed in `/tmp/alsa-utils/` (run `tree /tmp/alsa-utils`):

```
/tmp/alsa-utils/
|-- lib
```

```
|   `-- udev
|       `-- rules.d
|           `-- 90-alsa-restore.rules
|-- usr
|   |-- bin
|   |   |-- aconnect
|   |   |-- alsabat
|   |   |-- alsaloop
|   |   |-- alsaucm
|   |   |-- amidi
|   |   |-- amixer
|   |   |-- aplay
|   |   |-- aplaymidi
|   |   |-- arecord -> aplay
|   |   |-- arecordmidi
|   |   |-- aseqdump
|   |   |-- aseqnet
|   |   |-- axfer
|   |   |-- iecset
|   |   `-- speaker-test
|   |-- sbin
|   |   |-- alsabat-test.sh
|   |   |-- alsactl
|   |   `-- alsa-info.sh
|   `-- share
|       |-- alsa
|       |   `-- init
|       |       |-- 00main
|       |       |-- ca0106
|       |       |-- default
|       |       |-- hda
|       |       |-- help
|       |       |-- info
|       |       `-- test
|       |-- locale
|       |   |-- de
|       |   |   `-- LC_MESSAGES
|       |   |       `-- alsa-utils.mo
|       |   |-- eu
|       |   |   `-- LC_MESSAGES
|       |   |       `-- alsa-utils.mo
|       |   |-- fr
|       |   |   `-- LC_MESSAGES
|       |   |       `-- alsa-utils.mo
|       |   |-- ja
|       |   |   `-- LC_MESSAGES
|       |   |       `-- alsa-utils.mo
|       |   |-- ka
|       |   |   `-- LC_MESSAGES
|       |   |       `-- alsa-utils.mo
|       |   `-- sk
|       |       `-- LC_MESSAGES
|       |           `-- alsa-utils.mo
|       |-- man
```

```
|       |   |-- man1
|       |   |   |-- aconnect.1
|       |   |   |-- alsabat.1
|       |   |   |-- alsactl.1
|       |   |   |-- alsa-info.sh.1
|       |   |   |-- alsaloop.1
|       |   |   |-- amidi.1
|       |   |   |-- amixer.1
|       |   |   |-- aplay.1
|       |   |   |-- aplaymidi.1
|       |   |   |-- arecord.1 -> aplay.1
|       |   |   |-- arecordmidi.1
|       |   |   |-- aseqdump.1
|       |   |   |-- aseqnet.1
|       |   |   |-- axfer.1
|       |   |   |-- axfer-list.1
|       |   |   |-- axfer-transfer.1
|       |   |   |-- iecset.1
|       |   |   `-- speaker-test.1
|       |   `-- man7
|       `-- sounds
|           `-- alsa
|               |-- Front_Center.wav
|               |-- Front_Left.wav
|               |-- Front_Right.wav
|               |-- Noise.wav
|               |-- Rear_Center.wav
|               |-- Rear_Left.wav
|               |-- Rear_Right.wav
|               |-- Side_Left.wav
|               `-- Side_Right.wav
`-- var
    `-- lib
        `-- alsa

30 directories, 59 files
```

So, we have:

- The *udev* rules in `lib/udev`
- The *alsa-utils* binaries in `/usr/bin` and `/usr/sbin`
- Some sound samples in `/usr/share/sounds`
- The various translations in `/usr/share/locale`
- The manual pages in `/usr/share/man/`, explaining how to use the various tools
- Some configuration samples in `/usr/share/alsa`.

Now, let's make the installation in the *staging* space:

```
$ make DESTDIR=$HOME/embedded-linux-qemu-labs/thirdparty/staging/ install
```

Then, let's manually install only the necessary files in the *target* space. We are only interested in `speaker-test`:

```
$ cd ..
$ cp -a staging/usr/bin/speaker-test target/usr/bin/
$ arm-linux-strip target/usr/bin/speaker-test
```

And we're finally done with *alsa-utils*!

Now test that all is working fine by running the `speaker-test` util on your board, with the headset provided by your instructor plugged in. You may need to add the missing libraries from the toolchain install directory.

Now you can use:

- `speaker-test` with no arguments to generate *pink noise*
- `speaker-test -t sine` to generate a *sine wave*, optionally with `-f <freq>` for a specific frequency

**Known issue:** *according to our tests, you are likely to hear no sound on your Ubuntu 22.04 host when you run* `speaker-test`*. There should be no issue with what you built if* `speaker-test` *executes without an error, and if that's the case, you should consider this a success. Sound was working with the QEMU version on Ubuntu 20.04 and works again on Ubuntu 22.10. So, there is probably an issue somewhere in the QEMU integration in Ubuntu.*

There you are: you built and ran your first program depending on a library different from the C library.

## ipcalc

After practicing with autotools based packages, let's build *ipcalc*, which is using *Meson* as build system. We won't really need this utility in our system, but at least it has no dependencies and therefore offers an easy way to build our first *Meson* based package.

So, first install the `meson` package:

```
$ sudo apt install meson
```

In the main lab directory, then let's check out the sources through `git`:

```
$ git clone https://gitlab.com/ipcalc/ipcalc.git
$ cd ipcalc/
$ git checkout 1.0.3
```

To cross-compile with *Meson*, we need to create a *cross file*. Let's create the `../cross-file.txt` file with the below contents:

```
[binaries]
c = 'arm-linux-gcc'

[host_machine]
system = 'linux'
cpu_family = 'arm'
cpu = 'cortex-a9'
endian = 'little'
```

We also need to create a special directory for building:

```
$ mkdir cross-build
$ cd cross-build
```

We can now have `meson` create the Ninja build files for us:

```
$ meson --cross-file ../../cross-file.txt --prefix /usr ..
```

We are now ready to build *ipcalc*:

```
$ ninja
```

And now install `ipcalc` to the build space:

```
$ DESTDIR=$HOME/embedded-linux-qemu-labs/thirdparty/staging ninja install
```

Check that the `staging/usr/bin/ipcalc` file is indeed an ARM executable.

The last thing to do is to copy it to the target space and strip it:

```
$ cd ../..
$ cp staging/usr/bin/ipcalc target/usr/bin/
$ arm-linux-strip target/usr/bin/ipcalc
```

Note that we could have asked `ninja install` to strip the executable for us when installing it into the staging directory. To do, this, we would have added a `strip` entry in the cross file, and passed `--strip` to *Meson*. However, it's better to keep files unstripped in the staging space, in case we need to debug them.

You can now test that `ipcalc` works on the target:

```
# ipcalc 192.168.0.100
Address: 192.168.0.100
Address space: Private Use
```

## Final touch

To finish this lab completely, and to be consistent with what we've done before, let's strip the C library and its loader too.

First, check the initial size of the binaries:

```
$ ls -l target/lib
```

Then strip the binaries in `/lib`:

```
$ chmod +w target/lib/*.so.*
$ arm-linux-strip target/lib/*.so.*
```

And check the final size:

```
$ ls -l target/lib/
```

# Using a build system, example with Buildroot

*Objectives: discover how a build system is used and how it works, with the example of the Buildroot build system. Build a full Linux system, including the Linux kernel.*

## Goals

Compared to the previous lab, we are going to build a more elaborate system, still containing *alsa-utils* (and of course its *alsa-lib* dependency), but this time using Buildroot, an automated build system.

The automated build system will also allow us to add more packages and play real audio on our system, thanks to the *Music Player Daemon (mpd)* (https://www.musicpd.org/ and its *mpc* client.

As in a real project, we will also build the Linux kernel from Buildroot, and install the kernel modules in the root filesystem.

*Important note: because of the current sound playing issues mentioned before, this lab will be less exhaustive compared to our instructions on real hardware. You should be able to run the commands in the QEMU emulated machine though, proving that the tools were built correctly. So, we will build tools like mpd and mpc, but won't test them because of the absence of sound.*

## Setup

Go to the `$HOME/embedded-linux-qemu-labs/buildroot` directory.

## Get Buildroot and explore the source code

The official Buildroot website is available at https://buildroot.org/. Clone the *Git* repository:

```
git clone https://git.buildroot.net/buildroot
cd buildroot
```

Now checkout the tag corresponding to the latest 2023.02.<n> release (Long Term Support), which we have tested for this lab.

Several subdirectories or files are visible, the most important ones are:

- `boot` contains the Makefiles and configuration items related to the compilation of common bootloaders (GRUB, U-Boot, Barebox, etc.)
- `board` contains board specific configurations and root filesystem overlays.
- `configs` contains a set of predefined configurations, similar to the concept of defconfig in the kernel.
- `docs` contains the documentation for Buildroot.
- `fs` contains the code used to generate the various root filesystem image formats
- `linux` contains the Makefile and configuration items related to the compilation of the Linux kernel
- `Makefile` is the main Makefile that we will use to use Buildroot: everything works through Makefiles in Buildroot;

- `package` is a directory that contains all the Makefiles, patches and configuration items to compile the user space applications and libraries of your embedded Linux system. Have a look at various subdirectories and see what they contain;
- `system` contains the root filesystem skeleton and the *device tables* used when a static `/dev` is used;
- `toolchain` contains the Makefiles, patches and configuration items to generate the cross-compiling toolchain.

## Configure Buildroot

In our case, we would like to:

- Generate an embedded Linux system for ARM;
- Use an already existing external toolchain instead of having Buildroot generating one for us;
- Compile the Linux kernel and deploy its modules in the root filesystem;
- Integrate *BusyBox*, *alsa-utils*, *mpd*, *mpc* and *evtest* in our embedded Linux system;
- Integrate the target filesystem into a tarball

To run the configuration utility of Buildroot, simply run:

```
$ make menuconfig
```

Set the following options. Don't hesitate to press the `Help` button whenever you need more details about a given option:

- `Target options`
  - `Target Architecture`: `ARM (little endian)`
  - `Target Architecture Variant`: `cortex-A9`
  - `Enable NEON SIMD extension support`: Enabled
  - `Enable VFP extension support`: Enabled
  - `Target ABI`: `EABIhf`
  - `Floating point strategy`: `VFPv3-D16`
- `Toolchain`
  - `Toolchain type`: `External toolchain`
  - `Toolchain`: `Custom toolchain`
  - `Toolchain path`: use the toolchain you built: `/home/<user>/x-tools/arm-training-linux-musleabihf` (replace `<user>` by your actual user name)
  - `External toolchain gcc version`: `12.x`
  - `External toolchain kernel headers series`: `6.1.x or later`
  - `External toolchain C library`: `musl (experimental)`
  - We must tell Buildroot about our toolchain configuration, so select `Toolchain has SSP support?` and `Toolchain has C++ support?`. Buildroot will check these parameters anyway.
- `Kernel`
  - Enable `Linux Kernel`
  - Set `Kernel version` to `Latest version (6.1)`
  - Set `Kernel configuration` to `Using an in-tree defconfig file`

  - Set `Defconfig name` to `vexpress`
  - Select `Build a Device Tree Blob (DTB)`
  - Set `In-tree Device Tree Source file names` to `vexpress-v2p-ca9`
- `Target packages`
  - Keep `BusyBox` (default version) and keep the BusyBox configuration proposed by Buildroot;
  - `Audio and video applications`
    * Select `alsa-utils`, and in the submenu:
      · Select `alsamixer`. You will be able to test this application too, and that will also pull the `ncurses` library, which we will also use in the next lab.
      · Select `speaker-test`
    * Select `mpd`, and in the submenu:
      · Keep only `alsa`, `vorbis` and `tcp sockets`
    * Select `mpd-mpc`.
- `Filesystem images`
  - Select `tar the root filesystem`

Exit the menuconfig interface. Your configuration has now been saved to the `.config` file.

## Generate the embedded Linux system

Just run:

```
$ make
```

Buildroot will first create a small environment with the external toolchain, then download, extract, configure, compile and install each component of the embedded system.

All the compilation has taken place in the `output/` subdirectory. Let's explore its contents:

- `build`, is the directory in which each component built by Buildroot is extracted, and where the build actually takes place
- `host`, is the directory where Buildroot installs some components for the host. As Buildroot doesn't want to depend on too many things installed in the developer machines, it installs some tools needed to compile the packages for the target. In our case it installed *pkg-config* (since the version of the host may be ancient) and tools to generate the root filesystem image (*genext2fs*, *makedevs*, *fakeroot*).
- `images`, which contains the final images produced by Buildroot. In our case it contains a tarball of the filesystem, called `rootfs.tar`, plus the compressed kernel and Device Tree binary. Depending on the configuration, there could also a bootloader binary or a full SD card image.
- `staging`, which contains the "build" space of the target system. All the target libraries, with headers and documentation. It also contains the system headers and the C library, which in our case have been copied from the cross-compiling toolchain.
- `target`, is the target root filesystem. All applications and libraries, usually stripped, are installed in this directory. However, it cannot be used directly as the root filesystem, as all the device files are missing: it is not possible to create them without being root, and Buildroot has a policy of not running anything as root.

## Run the generated system

Go back to the `$HOME/embedded-linux-qemu-labs/buildroot/` directory. Create a new `nfsroot` directory that is going to hold our system, exported over NFS. Go into this directory, and untar the rootfs using:

```
$ tar xvf ../buildroot/output/images/rootfs.tar
```

Add our `nfsroot` directory to the list of directories exported by NFS in `/etc/exports`.

Also update the kernel and Device Tree binaries used by your board, from the ones compiled by Buildroot in `output/images/`.

Boot the board, and log in (`root` account, no password).

You should now reach a shell.

Even though we have no sound at the moment, you can run `speaker-test` to check that this application works. You can also test the `alsamixer` command too.

By running the `ps` command, you may also check whether the `mpd` server was started on your system. However, as said earlier, we won't try to test it as we currently have no sound in QEMU on Ubuntu 22.04.

## Analyzing dependencies

It's always useful to understand the dependencies drawn by the packages we build.

First we need to install a *Graphviz*:

```
$ sudo apt install graphviz
```

Now, let's use Buildroot's target to generate a dependency graph:

```
$ make graph-depends
```

We can now study the dependency graph:

```
$ evince output/graphs/graph-depends.pdf
```

In particular, you can see that adding MPD and its client required to compile *Meson* for the host, and in turn, *Python 3* for the host too. This substantially contributed to the build time.

# Application development

*Objective: Compile and run your own ncurses application on the target.*

## Setup

Go to the `$HOME/embedded-linux-qemu-labs/appdev` directory.

## Compile your own application

We will re-use the system built during the *Buildroot lab* and add to it our own application.

In the lab directory the file `app.c` contains a very simple *ncurses* application. It is a simple game where you need to reach a target using the arrow keys of your keyboard. We will compile and integrate this simple application to our Linux system.

Buildroot has generated toolchain wrappers in `output/host/bin`, which make it easier to use the toolchain, since these wrappers pass some mandatory flags (especially the `--sysroot` *gcc* flag, which tells *gcc* where to look for the headers and libraries).

Let's add this directory to our `PATH`:

```
$ export PATH=$HOME/embedded-linux-qemu-labs/buildroot/buildroot/output/host/bin:$PATH
```

Let's try to compile the application:

```
$ arm-linux-gcc -o app app.c
```

It complains about undefined references to some symbols. This is normal, since we didn't tell the compiler to link with the necessary libraries. So let's use `pkg-config` to query the *pkg-config* database about the location of the header files and the list of libraries needed to build an application against *ncurses*[9]:

```
$ arm-linux-gcc -o app app.c $(pkg-config --libs --cflags ncurses)
```

Our application is now compiled! Copy the generated binary to the NFS root filesystem (in the `root/` directory for example), start your system, and run your application!

[9] Again, `output/host/bin` has a special `pkg-config` that automatically knows where to look, so it already knows the right paths to find `.pc` files and their sysroot.

# Remote application debugging

*Objective: Use* `strace` *and* `ltrace` *to diagnose program issues. Use* `gdbserver` *and a cross-debugger to remotely debug an embedded application*

## Setup

Go to the `$HOME/embedded-linux-qemu-labs/debugging` directory. Create an `nfsroot` directory.

## Debugging setup

Because of issues in *gdb* and *ltrace* in the musl version that we are using in our toolchain, we will use a different toolchain in this lab, based on *glibc*.

As *glibc* has more complete features than lighter libraries, it looks like a good idea to do your application debugging work with a *glibc* toolchain first, and then switch to lighter libraries once your application and software stack is production ready.

As done in the *Buildroot* lab, clone once again the Buildroot *Git* repository, and checkout the tag corresponding to the latest 2023.02.<n> release (Long Term Support), which we have tested for this lab.

Then, in the `menuconfig` interface, configure the target architecture as done previously but configure the toolchain and target packages differently:

- In `Toolchain`:
  - `Toolchain type: External toolchain`
  - `Toolchain: Bootlin toolchains`
  - `Toolchain origin: Toolchain to be downloaded and installed`
  - `Bootlin toolchain variant: armv7-eabihf glibc stable 2022.08-1`
  - Select `Copy gdb server to the Target`
- `Target packages`
  - `Debugging, profiling and benchmark`
    - Select `ltrace`
    - Select `strace`

Now, build your root filesystem.

Go back to the `$HOME/embedded-linux-qemu-labs/debugging` directory and extract the `buildroot/output/images/rootfs.tar` archive in the `nfsroot` directory.

Add this directory to the `/etc/exports` file and run `sudo exportfs -r`.

Boot your ARM board over NFS on this new filesystem, using the same kernel as before.

## Using strace

Now, go to the `$HOME/embedded-linux-qemu-labs/debugging` directory.

`strace` allows to trace all the system calls made by a process: opening, reading and writing files, starting other processes, accessing time, etc. When something goes wrong in your application, strace is an invaluable tool to see what it actually does, even when you don't have the source code.

Update the PATH:

```
$ export PATH=$HOME/embedded-linux-qemu-labs/debugging/buildroot/output/host/bin:$PATH
```

With your cross-compiling toolchain compile the `data/vista-emulator.c` program, copy the resulting binary to the `/root` directory of the root filesystem and then strip it.

Back to target system, try to run the `/root/vista-emulator` program. It should hang indefinitely!

Interrupt this program by hitting `[Ctrl] [C]`.

Now, running this program again through the `strace` command and understand why it hangs. You can guess it without reading the source code!

Now add what the program was waiting for, and now see your program proceed to another bug, failing with a segmentation fault.

## Using ltrace

Now run the program through `ltrace`.

Now you should see what the program does: it tries to consume as much system memory as it can!

Also run the program through `ltrace -c`, to see what function call statistics this utility can provide.

It's also interesting to run the program again with `strace`. You will see that memory allocations translate into `mmap()` system calls. That's how you can recognize them when you're using `strace`.

## Using gdbserver

We are now going to use `gdbserver` to understand why the program segfaults.

Compile `vista-emulator.c` again with the `-g` option to include debugging symbols. This time, just keep it on your workstation, as you already have the version without debugging symbols on your target.

Then, on the target side, run `vista-emulator` under `gdbserver`. `gdbserver` will listen on a TCP port for a connection from `gdb`, and will control the execution of `vista-emulator` according to the `gdb` commands:

```
=> gdbserver localhost:2345 vista-emulator
```

On the host side, run `arm-linux-gdb` (also found in your toolchain):

```
$ arm-linux-gdb vista-emulator
```

`gdb` starts and loads the debugging information from the `vista-emulator` binary that has been compiled with `-g`.

Then, we need to tell where to find our libraries, since they are not present in the default `/lib` and `/usr/lib` directories on your workstation. This is done by setting the `gdb sysroot` variable (on one line):

```
(gdb) set sysroot /home/<user>/embedded-linux-qemu-labs/debugging/\
    buildroot/output/staging
```

Of course, replace `<user>` by your actual user name.

And tell `gdb` to connect to the remote system:

```
(gdb) target remote <target-ip-address>:2345
```

Then, use `gdb` as usual to set breakpoints, look at the source code, run the application step by step, etc. Graphical versions of `gdb`, such as `ddd` can also be used in the same way. In our case, we'll just start the program and wait for it to hit the segmentation fault:

```
(gdb) continue
```

You could then ask for a backtrace to see where this happened:

```
(gdb) backtrace
```

This will tell you that the segmentation fault occurred in a function of the C library, called by our program. This should help you in finding the bug in our application.

## Post mortem analysis

Following the details in the slides, configure your shell on the target to get a `core` file dumped when you run `vista-emulator` again.

Once you have such a file, inspect it with `arm-linux-gdb` on the target, set the `sysroot` setting, and then generate a backtrace to see where the program crashed.

This way, you can have information about the crash without running the program through the debugger.

## What to remember

During this lab, we learned that...

- It's easy to study the behavior of programs and diagnose issues without even having the source code, thanks to `strace` and `ltrace`.
- You can leave a small `gdbserver` program (about 300 KB) on your target that allows to debug target applications, using a standard `gdb` debugger on the development host.
- It is fine to strip applications and binaries on the target machine, as long as the programs and libraries with debugging symbols are available on the development host.
- Thanks to `core` dumps, you can know where a program crashed, without having to reproduce the issue by running the program through the debugger.