Embedded Linux System Development

STM32MP2 variant

Practical Labs

bootlin
https://bootlin.com

December 03, 2025

About this document

Updates to this document can be found on https://bootlin.com/doc/training/embedded-linux-stm32mp2.

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-stm32mp2/embedded-linux-stm32mp2-labs.tar.xz
$ tar xvf embedded-linux-stm32mp2-labs.tar.xz
```

Lab data are now available in an `embedded-linux-stm32mp2-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-stm32mp2-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 gettext python3
```

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-1.28.0
```

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 `aarch64-unknown-linux-uclibc` 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 `Target options`:

- Set `Emit assembly for CPU` (`ARCH_CPU`) to `cortex-a53`.
- Check that `Endianness` (`ARCH_ENDIAN`) is set to `Little endian`

In `Toolchain options`:

- Set `Tuple's vendor string` (`TARGET_VENDOR`) to `training`.
- Set `Tuple's alias` (`TARGET_ALIAS`) to `aarch64-linux`. This way, we will be able to use the compiler as `aarch64-linux-gcc` instead of `aarch64-training-linux-musl-gcc`, which is much longer to type.

In `Operating System`:

- Set `Version of linux` to the closest, but older version to 6.6. It's important that the kernel headers used in the toolchain are not more recent than the kernel that will run on the board (v6.6).

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 `14.2.0`.
- 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/aarch64-training-linux-musl/bin/` to your `PATH` environment variable and compiling the simple `hello.c` program in your main lab directory with `aarch64-linux-gcc`:

```
$ aarch64-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 AARCH64 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 AARCH64 user emulator:

```
$ qemu-aarch64 hello
qemu-aarch64: Could not open '/lib/ld-musl-aarch64.so.1': No such file or directory
```

What's happening is that `qemu-aarch64` is missing the shared library loader (compiled for AARCH64) that this binary relies on. Let's find it in our newly compiled toolchain:

```
$ find ~/x-tools -name ld-musl-aarch64.so.1
```

```
/home/tux/x-tools/aarch64-training-linux-musl/aarch64-training-linux-musl/sysroot/lib/
ld-musl-aarch64.so.1
```

We can now use the `-L` option of `qemu-aarch64` to let it know where shared libraries are:

```
$ qemu-aarch64 -L ~/x-tools/aarch64-training-linux-musl/aarch64-training-linux-musl/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 - TF-A and U-Boot

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

As the bootloader is the first piece of software executed by a hardware platform, the installation procedure of the bootloader is very specific to the hardware platform. There are usually two cases:

- The processor offers nothing to ease the installation of the bootloader, in which case the JTAG has to be used to initialize flash storage and write the bootloader code to flash. Detailed knowledge of the hardware is of course required to perform these operations.
- The processor offers a monitor, implemented in ROM, and through which access to the memories is made easier.

The STM32MP2 SoC falls into the second category. The monitor integrated in the ROM reads the SD card to search for a valid bootloader (the boot mode is actually configurable via a few input pins). In case no bootloader is found, it will operate in a fallback mode, that will allow to use an external tool to reflash some executable through USB. Therefore, either by using an MMC/SD card or this fallback mode, we can start up an STM32MP2-based board without having anything installed on it.

Setup

Go to the `$HOME/embedded-linux-stm32mp2-labs/bootloader` directory.

Setting up serial communication with the board

Plug the USB-C to USB-C cable on the Discovery board. There are two USB-C ports on the board, choose the one labeled `USB PWR ST-LINK`. This is a debug interface that exposes multiple debugging interfaces, including a UART interface. When plugged in your computer, a serial port should appear, `/dev/ttyACM0`.

You can also see this device appear by looking at the output of `sudo dmesg`.

To communicate with the board through the serial port, install a serial communication program, such as `picocom`:

```
$ sudo apt install picocom
```

If you run `ls -l /dev/ttyACM0`, you can also see that only `root` and users belonging to the `dialout` group have read and write access to the serial console. Therefore, you need to add your user to the `dialout` group:

```
$ sudo adduser $USER dialout
```

Important: for the group change to be effective, you have to reboot your computer (at least on Ubuntu 24.04) and log in again. A workaround is to run `newgrp dialout`, but it is not global. You have to run it in each terminal.

Run `picocom -b 115200 /dev/ttyACM0`, to start serial communication on `/dev/ttyACM0`, with a baudrate of 115200. If you wish to exit `picocom`, press `[Ctrl][a]` followed by `[Ctrl][x]`.

Don't be surprised if you don't get anything on the serial console yet, even if you reset the board. That's because the SoC has nothing to boot on yet. We will prepare a micro SD card to boot on in the next sections.

TF-A and U-Boot relationship

The boot process is done in two main stages. At power-on, the ROM code (also called ROM monitor) executes the first-stage bootloader, known as *fsbl*, from the internal SRAM. This stage is responsible for initializing essential system components, including the DRAM.

Once the system is sufficiently initialized, the fsbl loads and transfers control to the second-stage bootloader, known as *ssbl*, which is responsible for loading and launching the main operating system (Linux).

In our setup, the *fsbl* is provided by TF-A BL2, and the *ssbl* is U-Boot.

TF-A BL2 loads U-Boot from the Firmware Image Package (*FIP*), which also includes other components such as the secure monitor (BL31) and the OP-TEE secure OS. The FIP is generated as part of the TF-A build process, which requires U-Boot and OP-TEE to be built beforehand, so that it can be included in the package.

U-Boot setup

Download U-Boot from ST's Git repository:

```
$ git clone https://github.com/STMicroelectronics/u-boot.git
$ cd u-boot
$ git checkout v2023.10-stm32mp-r1
```

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:

1. Specify the cross-compiler prefix (the part before `gcc` in the cross-compiler executable name):

   ```
   $ export CROSS_COMPILE=aarch64-linux-
   ```

2. Run `$ make <NAME>_defconfig`, where the list of available configurations can be found in the `configs/` directory. We will use the standard one (`stm32mp25`).

3. Now that you have a valid initial configuration, you can now 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 an ext4 filesystem:
     - Disable `Environment is not stored`. We want changes to variables to be persistent across reboots
     - Enable `Environment is in a EXT4 filesystem`. Disable all other options for environment storage (e.g. MMC, SPI, UBI, flash)
     - The value for `Name of the block device for the environment` should be `mmc`
     - The value for `Device and partition for where to store the environment in EXT4` should be `0:3`, which indicates we want to store the environment in the 3th partition of the first MMC device.
     - The value for `Name of the EXT4 file to use for the environment` should be `/uboot.env`, which indicates the filename inside which the U-Boot environment will be stored

   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-dev python3-setuptools uuid-dev libgnutls28-dev
   ```

4. Finally, run

   ```
   make DEVICE_TREE=stm32mp257f-dk
   ```

   which will build U-Boot [2]. The `DEVICE_TREE` variable specifies the specific Device Tree that describes our hardware board.

   Note: u-boot build may fail on your machine if you have a recent version of python. Such issue is already fixed upstream, but not in the version targeted for the training. To get the relevant fix, you can cherry-pick the fix onto your local branch:

   ```
   git cherry-pick a63456b9191fae2fe49f4b121e025792022e3950
   ```

OP-TEE setup

OP-TEE is required on STM32MP2 platforms to provide a secure execution environment during the boot process. It enables isolated execution of sensitive operations such as key management and authentication. Included in the FIP (Firmware Image Package), OP-TEE is essential for booting on STM32MP2 platforms.

Get the ST-provided OP-TEE sources:

```
$ cd ..
$ git clone https://github.com/STMicroelectronics/optee_os.git
$ cd optee_os
$ git checkout 4.0.0-stm32mp-r1
```

OP-TEE need some libraries to be built. You can install them with:

```
sudo apt install -y python3-pyelftools cmake python3-cryptography
```

Then to build OP-TEE, several configuration parameters have to be passed to the Makefile:

- Specify the cross-compiler prefix (the part before gcc in the cross-compiler executable name), either using the environment variable: `$ export CROSS_COMPILE=aarch64-linux-`, or just by adding it to the `make` commande line.
- Specifies the cross-compiler prefix specifically for building the TEE core `$ export CROSS_COMPILE_core=aarch64-linux-`
- Specifies the cross-compiler used for Trusted Applications (TAs) compiled for 64-bit ARM: `$ export CROSS_COMPILE_ta_`
- Specifies which architectures to build user-mode TAs for : `$ export CFG_USER_TA_TARGETS=ta_arm64`
- Disables support for the Trusted User Interface (TUI) framework: `$ export CFG_WITH_TUI=n`
- Ensures that the OP-TEE core is built for 64-bit ARM architecture (AArch64). It is required for platforms like STM32MP2, which are based on ARMv8 and operate in 64-bit mode: `$ export CFG_ARM64_core=y`
- Specify the output file : `$ export O=out`
- Define the specific hardware target for the OP-TEE build `$ export PLATFORM=stm32mp2`

The full command line to build OP-TEE is therefore:

```
make O=out \
  CROSS_COMPILE=aarch64-linux- \
  CROSS_COMPILE_core=aarch64-linux- \
  CROSS_COMPILE_ta_arm64=aarch64-linux- \
  CFG_ARM64_core=y \
```

[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.


```
  CFG_USER_TA_TARGETS=ta_arm64 \
  PLATFORM=stm32mp2 \
  CFG_WITH_TUI=n \
  all
```

Now that OP-TEE is built, we can move on to the next and final step: TF-A.

TF-A setup

Get the ST-provided TF-A sources:

```
$ cd ..
$ git clone https://github.com/STMicroelectronics/arm-trusted-firmware.git
$ cd arm-trusted-firmware/
$ git checkout v2.10-stm32mp-r1
```

We have to patch the TF-A sources to add more recent firmware code for the DDR configuration:

```
$ git clone https://github.com/STMicroelectronics/stm32-ddr-phy-binary.git drivers/st/ddr/phy\
   /firmware/bin
```

Then to build TF-A, several configuration parameters have to be passed to the Makefile:

- Specify the cross-compiler prefix (the part before gcc in the cross-compiler executable name), either using the environment variable: `$ export CROSS_COMPILE=aarch64-linux-`, or just by adding it to the `make` commande line.
- The architecture has to be selected: `ARCH=aarch64`, as well as the major version of ARM architecture, here the Cortex A35 is an ARMv8, so we need to use `ARM_ARCH_MAJOR=8`
- The STM32MP2 platform is selected too with `PLAT=stm32mp2`
- Sets the Secure Payload Dispatcher to OP-TEE: `SPD=opteed`
- For this specific board, the device tree is generated and then needs to be specifed: `DTB_FILE_NAME=stm32mp257f-dk.dtb`
- Specify the location of the OP-TEE BL32 image. Needed for trusted OS support: `BL32=../optee_os/out/core/tee-header_v2.bin`
- Specify the additional OP-TEE image (paged memory region), which is part of the Trusted Execution Environment: `BL32_EXTRA1=../optee_os/out/core/tee-pager_v2.bin`
- Specify the final part of the OP-TEE image (pageable region), which works with the pager image: `BL32_EXTRA2=../optee_os/out/core/tee-pageable_v2.bin`
- Specify the Device Tree that will be used by U-Boot which is actually the Device Tree passed to U-Boot: `BL33_CFG=../u-boot/u-boot.dtb`
- Specify that TF-A will be located on the SD card and therefore needs to have support for SD/MMC: `STM32MP_SDMMC=1`
- Specify the location of the BL33 image, in our case U-Boot: `BL33=../u-boot/u-boot-nodtb.bin`
- Select LPDDR4 memory configuration type: `STM32MP_LPDDR4_TYPE=1`

We can now generate the `all` and `fip` make targets with a single (but long) command line:

```
$ make \
  CROSS_COMPILE=aarch64-linux- \
  PLAT=stm32mp2 ARM_ARCH_MAJOR=8 \
  ARCH=aarch64 \
```



```
STM32MP_LPDDR4_TYPE=1 \
DTB_FILE_NAME=stm32mp257f-dk.dtb \
STM32MP_SDMMC=1 SPD=opteed \
BL32=../optee_os/out/core/tee-header_v2.bin \
BL32_EXTRA1=../optee_os/out/core/tee-pager_v2.bin \
BL32_EXTRA2=../optee_os/out/core/tee-pageable_v2.bin \
BL33=../u-boot/u-boot-nodtb.bin \
BL33_CFG=../u-boot/u-boot.dtb \
all fip
```

At the end of the build, the important output files generated are located in `build/stm32mp2/release/`. We will find there:

- `tf-a-stm32mp257f-dk.stm32`, which is TF-A BL2, serving as our first stage bootloader
- `fip.bin`, which is the FIP image, which itself includes U-Boot and OP-TEE. This image will serve as the second stage bootloader.

Flashing the bootloaders

The ROM monitor in the STM32MP2 boot chain is the first piece of code executed by the Cortex-A35 processor in secure mode. It selects the boot source based on BOOT pins, OTP configuration, and TAMP registers, and optionally performs authentication and decryption of the First Stage Bootloader (FSBL). Once a valid FSBL (typically Trusted Firmware-A BL2) is located, it is loaded into SYSRAM and executed. The FSBL then initializes key system components such as the clock tree and DDR controller, and proceeds to load the Secure Monitor (BL31), the Secure OS (OP-TEE), and the Second Stage Bootloader (SSBL), which is U-Boot. After all required components are securely loaded into DDR memory, control is transferred to U-Boot, which takes over as the SSBL to continue system initialization and load the Linux kernel.

In our case, all bootloader stages will be stored on the SD card, for which a specific partitioning is necessary:

```
Number  Start   End      Size      File system  Name    Flags
1       34s     512s     479s                   fsbl
2       513s    8704s    8192s                  fip
3       8705s   129537s  120833s                bootfs
```

On your workstation, plug in the SD card your instructor gave you. Type the `sudo dmesg` command to see which device is used by your workstation. In case the device is `/dev/mmcblk0`, you will see something like

```
[46939.425299] mmc0: new high speed SDHC card at address 0007
[46939.427947] mmcblk0: mmc0:0007 SD16G 14.5 GiB
```

The device file name may be different (such as `/dev/sdb` if the card reader is connected to a USB bus (either internally or using a USB card reader).

In the following instructions, we will assume that your SD card is seen as `/dev/mmcblk0` by your PC workstation.

Type the `mount` command to check your currently mounted partitions. If SD partitions are mounted, unmount them:

```
$ sudo umount /dev/mmcblk0p*
```

We will erase the existing partition table and partition contents by simply zero-ing the first 128 MiB of the SD card:

```
$ sudo dd if=/dev/zero of=/dev/mmcblk0 bs=1M count=128
```

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


```
$ sudo parted /dev/mmcblk0
```

The ROM monitor handles *GPT* partition tables, let's create one:

```
(parted) mklabel gpt
```

Then, the 4 partitions are created with:

```
(parted) mkpart fsbl 0% 512s
(parted) mkpart fip 513s 8704s
(parted) mkpart bootfs 8705s 129537s
```

You can verify everything looks right with:

```
(parted) print
Model: SD SA08G (sd/mmc)
Disk /dev/mmcblk0: 15278080s
Sector size (logical/physical): 512B/512B
Partition Table: gpt
Disk Flags:

Number  Start    End        Size       File system  Name     Flags
 1      34s      512s       479s                    fsbl
 2      513s     8704s      8192s                   fip
 3      8705s    129537s    120833s                 bootfs
```

Once done, quit:

```
(parted) quit
```

Note: `parted` *is definitely not very user friendly compared to other tools to manipulate partitions (such as* `cfdisk`*), but that's the only tool which supports assigning names to GPT partitions. In your projects, you could use* `gparted`*, which is a more friendly graphical front-end on top of* `parted`*.*

Now, format the boot partition as an ext4 filesystem. This is where U-Boot saves its environment:

```
$ sudo mkfs.ext4 -L boot -O ^metadata_csum /dev/mmcblk0p3
```

The `-O ^metadata_csum` option allows to create the filesystem without enabling metadata checksums, which U-Boot doesn't support yet.

Now write the TF-A binary in the `fsbl` partition:

```
$ sudo dd if=/build/stm32mp2/release/tf-a-stm32mp257f-dk.stm32 of=/dev/mmcblk0p1 bs=1M \
    conv=fdatasync
```

Then flash the *fip* partition with the Firmware Image Package containing U-Boot, the BL31 monitor and OP-TEE:

```
$ sudo dd if=build/stm32mp2/release/fip.bin of=/dev/mmcblk0p2 bs=1M conv=fdatasync
```

Testing the bootloaders

Insert the SD card in the board slot. You can now power-up the board by connecting the USB-C cable to the board, in CN6, `PWR_IN` and to your PC at the other end. Check that it boots your new bootloaders. You can verify this by checking the build dates:

```
NOTICE:  CPU: STM32MP257FAK Rev.Y
NOTICE:  Model: STMicroelectronics STM32MP257F-DK Discovery Board
NOTICE:  Board: MB1605 Var1.0 Rev.C-01
NOTICE:  BL2: v2.10-stm32mp2-r1.0(release):custom(75f8c318)
NOTICE:  BL2: Built : 10:31:22, Apr  4 2025
```



```
NOTICE:  BL2: Booting BL31
NOTICE:  BL31: v2.10-stm32mp2-r1.0(release):custom(75f8c318)
NOTICE:  BL31: Built : 10:31:22, Apr  4 2025
I/TC: Early console on UART#2
I/TC:
I/TC: Embedded DTB found
I/TC: OP-TEE version: 4.0.0-stm32mp-r1-dev (gcc version 13.3.0 (crosstool-NG 1.27.0.20_329bb4d)) #7 Fri Apr
I/TC: WARNING: This OP-TEE configuration might be insecure!
I/TC: WARNING: Please check https://optee.readthedocs.io/en/latest/architecture/porting_guidelines.html
I/TC: Primary CPU initializing
I/TC: WARNING: All debug access are allowed
I/TC: Override the OTP 124: 0 to 0x18db6
I/TC: WARNING: Embeds insecure stm32mp_provisioning driver
I/TC: UART console (non-secure)
I/TC: PMIC STPMIC REFID:2.@ V1.1
I/TC: Platform stm32mp2: flavor 257F_DK - DT stm32mp257f-dk.dts
I/TC: OP-TEE ST profile: secure_and_system_services
[    0.000000] SCP-firmware 2.13.0-intree-optee-os-4.0.0-stm32mp-r1-dev
[    0.000000]
[    0.000000] [FWK] Module initialization complete!
I/TC: Primary CPU switching to normal world boot
I/TC: Reserved shared memory is disabled
I/TC: Dynamic shared memory is enabled
I/TC: Normal World virtualization support is disabled
I/TC: Asynchronous notifications are enabled


U-Boot 2023.10-stm32mp-r1 (Apr 03 2025 - 15:40:02 +0200)

CPU: STM32MP257FAK Rev.Y
Model: STMicroelectronics STM32MP257F-DK Discovery Board
Board: stm32mp2 (st,stm32mp257f-dk)
Board: MB1605 Var1.0 Rev.C-01
DRAM:  4 GiB
optee optee: OP-TEE: revision 4.0 (d9c4df97)
I/TC: Reserved shared memory is disabled
I/TC: Dynamic shared memory is enabled
I/TC: Normal World virtualization support is disabled
I/TC: Asynchronous notifications are enabled
Core:  432 devices, 41 uclasses, devicetree: board
WDT:   Started watchdog with servicing every 1000ms (32s timeout)
NAND:  0 MiB
MMC:   STM32 SD/MMC: 0, STM32 SD/MMC: 1
Loading Environment from MMC... OK
In:    serial
Out:   serial
Err:   serial
Net:   eth0: eth1@482c0000
FWU metadata read failed
No EFI system partition
No EFI system partition
Failed to persist EFI variables
Hit any key to stop autoboot:  0
Boot over mmc0!
```



```
switch to partitions #0, OK
mmc0 is current device
STM32MP>
```

In U-Boot, type the `help` command, and explore the few commands available.

Adding a new command to the U-Boot shell

Check whether the `config` command is available. This command allows to dump the configuration settings U-Boot was compiled from.

If it's not, go back to U-Boot's configuration and enable it.

Re-run the build of U-Boot, and then re-run the build of TF-A so that a new version of the `fip.bin` with the updated U-Boot is generated.

Update the `fip` partition on the SD card with the new `fip.bin` image and test that the command is now available and works as expected.

Playing with the U-Boot environment

Display the U-Boot environment using `printenv`.

Set a new U-Boot variable `foo` to a value of your choice, using `setenv`, and verify it has been set. Reset the board, and check if `foo` is still defined: it should not.

Now repeat this process, but before resetting the board, use `saveenv`. After the reset, check the `foo` variable is still defined.

Now reset the environment to its default settings using `env default -a`, and save these changes using `saveenv`.

Setting up networking

The next step is to configure U-boot and your workstation to let your board download files, such as the kernel image and Device Tree Binary (DTB), using the TFTP protocol through a network connection.

With a network cable, connect the Ethernet port of your board to the one of your computer. If your computer already has a wired connection to the network, your instructor will provide you with a USB Ethernet adapter. A new network interface should appear on your Linux system.

Network configuration on the target

Let's configure networking in U-Boot:

- `ipaddr`: IP address of the board
- `serverip`: IP address of the PC host

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

Of course, make sure that this address belongs to a separate network segment from the one of the main company network.

To make these settings permanent, save the environment:

```
=> saveenv
```

Network configuration on the PC host

To configure your network interface on the workstation side, we need to know the name of the network interface connected to your board.

Find the name of this interface by typing:

```
=> ip a
```

The network interface name is likely to be `enxxx`[3]. If you have a pluggable Ethernet device, it's easy to identify as it's the one that shows up after pluging in the device.

Then, instead of configuring the host IP address from NetworkManager's graphical interface, let's do it through its command line interface, which is so much easier to use:

```
$ nmcli con add type ethernet ifname en... ip4 192.168.0.1/24
```

Setting up the TFTP server

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

`sudo apt install tftpd-hpa`

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

```
=> tftp 0xc2000000 textfile.txt
```

In case the download fails, make sure your host interface is correctly configured and if a firewall is enabled make sure it does not filter out our requests:

`sudo ufw allow from 192.168.0.100`

Otherwise, the `tftp` command should have downloaded the `textfile.txt` file from your development workstation into the board's memory at location `0xc2000000`[4].

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

```
=> md 0xc2000000
```

We will see in the next labs how to use U-Boot to download, flash and boot a kernel.

Rescue binaries

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

[3] Following the *Predictable Network Interface Names* convention: https://www.freedesktop.org/wiki/Software/systemd/PredictableNetworkInterfaceNames/

[4] This location is part of the board DRAM. If you want to check where this value comes from, you can check the SoC datasheet at https://www.st.com/resource/en/reference_manual/dm00327659.pdf. It's a big document (more than 4,000 pages). In this document, look for `Memory organization` and you will find the SoC memory map. You will see that the address range for the memory controller (*DDRC*) starts at the address we are looking for. You can also try with other values in the RAM address range.

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 Stmicroelectronics git.

Setup

Create the `$HOME/embedded-linux-stm32mp2-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 Linux tree

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

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://github.com/torvalds/linux.git
```

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.

We will now add the STMicroelectronics branch as remote, which contains patches required for Linux to run properly on this board.

```
git remote add st https://github.com/STMicroelectronics/linux
git fetch st
```

Now let's take a look at the branches provided by ST:

```
git branch -r | grep st/
```

For the labs we will use a specific branch of the Linux kernel ST, which is the `v6.6-stm32mp-r1` branch.

```
$ git checkout v6.6-stm32mp-r1
```

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 STM32MP257F-DK Discovery kit
- Use U-Boot to download the kernel
- Check that the kernel you compiled starts the system

Setup

Stay in the `$HOME/embedded-linux-stm32mp2-labs/kernel` directory.

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=$HOME/x-tools/aarch64-training-linux-musl/bin:$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.

The standard configuration for this kernel is actually `defconfig`, but this will generate a pretty big kernel with support for many other SoCs. However, we can reduce it to compile faster and get a small kernel.

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.
- In the `Platform selection` menu, remove support for all SoC families except the *STMicroelectronics STM32 SoC Family* one.
- Disable `CONFIG_DRM`, which will skip support for many display controller and GPU drivers.
- Disable `CONFIG_NET_VENDOR_MELLANOX`, to disable the build of Mellanox network drivers that cause a build issue in this particular kernel version.
- Disable `CONFIG_INPUT_TOUCHSCREEN`, as 2 touchscreen drivers have incorrect unconditional dependencies on DRM code, causing build failures in this particular kernel version.
- In Device Driver enable `CONFIG_DEVTMPFS` and `CONFIG_DEBUG_FS`

- Enabled the CONFIG_STMMAC_ETH, CONFIG_STMMAC_PLATFORM and CONFIG_DWMAC_STM32 as built-in options instead of modules, as we need network support in the kernel image itself.

Please note that this will definitely not build the smallest and most optimized kernel for STM32MP2: defconfig enables plenty of features and drivers that will not be useful on our particular board.

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=ttySTM0,115200
=> saveenv
```

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

- On your workstation, copy the Image and DTB (stm32mp257f-dk.dtb) to the directory exposed by the TFTP server.
- On the target (in the U-Boot prompt), load Image from TFTP into RAM:

  ```
  => tftp 0xc2000000 Image
  ```

- Now, also load the DTB file into RAM:

  ```
  => tftp 0xc4f00000 stm32mp257f-dk.dtb
  ```

- Boot the kernel with its device tree:

  ```
  => booti 0xc2000000 - 0xc4f00000
  ```

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 0xc2000000 Image; tftp 0xc4f00000 stm32mp257f-dk.dtb; booti 0xc2000000
    - 0xc4f00000'
=> 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:

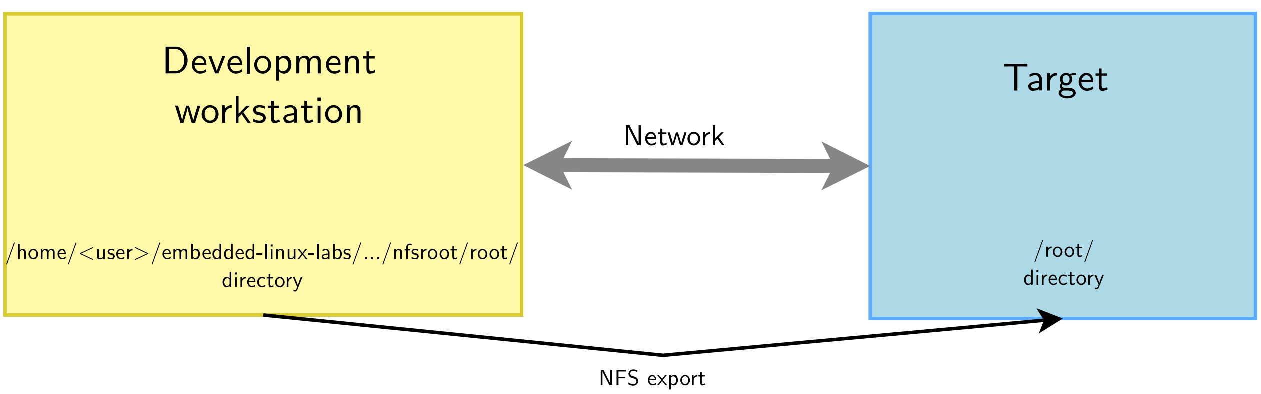
- be able to configure and build a Linux kernel that boots on a directory on your workstation, shared through the network by NFS.
- be able to create and configure a minimalistic root filesystem from scratch (ex nihilo, out of nothing, entirely hand made...) for your target board.
- understand how small 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-stm32mp2-labs/tinysystem/` directory.

Kernel configuration

We will re-use the kernel sources from our previous lab, in `$HOME/embedded-linux-stm32mp2-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-stm32mp2-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-stm32mp2-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 X:Y.
```

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 1.37.x release:

```
git clone https://git.busybox.net/busybox
cd busybox/
git checkout 1_37_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.

If you are running on a distribution that uses GCC >= 14.x, you will face an issue when trying to run `make menuconfig`, caused by a bug in Busybox, unfixed as of Busybox 1.37.0. You can fix this issue by applying an additional patch to the Busybox source:

```
git am $HOME/embedded-linux-stm32mp2-labs/tinysystem/data/\
    0001-menuconfig-GCC-failing-saying-ncurses-is-not-found.patch
```

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?

[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.

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
```

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:

```
ttySTM0::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 AARCH64, 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 64-bit LSB executable, ARM aarch64, version 1 (SYSV),
dynamically linked, interpreter /lib/ld-musl-aarch64.so.1, not stripped
```

So, what's missing is the `/lib/ld-musl-aarch64.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:

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


```
=> /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.

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 and declare new ones.

Goals

Now that we have access to a command line shell thanks to a working root filesystem, we can now explore existing devices and make new ones available. In particular, we will make changes to the Device Tree and compile an out-of-tree Linux kernel module.

Setup

Go to the `$HOME/embedded-linux-stm32mp2-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 `ttySTM0` 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.
- If you have a real board (not QEMU) and a USB stick, you could plug it in and if your kernel was built with USB host and mass storage support, you should see a new `sda` device appear, together with the `sda<n>` devices for its 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.

Driving GPIOs

At this stage, we can only explore GPIOs through the legacy interface in `/sys/class/gpio`, because the *libgpiod* interface commands are provided through a dedicated project which we have to build separately, and *Busybox* does not provide a re-implementation for the *libgpiod* tools. In a later lab, we will build *libgpiod* tools which use the modern `/dev/gpiochipX` interface.

The first thing to do is to enable this legacy interface by enabling `CONFIG_GPIO_SYSFS` in the kernel configuration. Also make sure *Debugfs* is enabled (`CONFIG_DEBUG_FS` and `CONFIG_DEBUG_FS_ALLOW_ALL`).

After rebooting the new kernel, the first thing to do is to mount the *Debugfs* filesystem:

```
# mount -t debugfs debugfs /sys/kernel/debug/
```

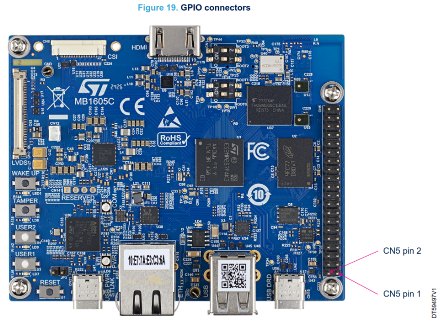
Then, you can check information about available GPIOs banks and which GPIOs are already in use:

```
# cat /sys/kernel/debug/gpio
```

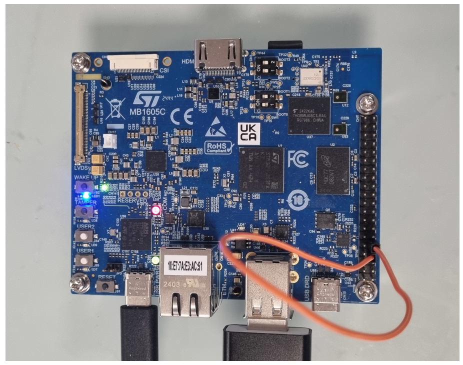
We are now going to use one of the header pins at the top of the board, which is not already used by another device.

Take one of the F-F breadboard wires provided by your instructor and:

Figure 19. GPIO connectors



- Connect one end to pin 7 of connector PF11
- Connect the other end to pin 9 (GND)



If you check the *GPIO expansion connector* table in the board documentation [8], you will see that the MCO1 pin on the board is connected to the PF11 STM32 pin. PF11 is actually a GPIO pin on GPIO bank F, and is configured as a GPIO by default (no need to change pin muxing to use this pin as a GPIO).

If you get back to the contents of `/sys/kernel/debug/gpio`, you'll find that GPIO bank F corresponds to `gpiochip5` and to GPIO numbers 590 to 605. Hence, PF11, the second pin on this bank corresponds to GPIO number 601.

We now have everything we need to drive this GPIO using the legacy interface. First, let's enable it:

```
# cd /sys/class/gpio
# echo 601 > export
```

If indeed the pin is still available, this should create a new `PF11` file in `/sys/class/gpio`.

We can now configure this pin as input:

```
# echo in > PF11/direction
```

And check its value:

```
# cat PF11/value
0
```

The value should be `0` as the pin is connected to a ground level. Now, let's connect our GPIO pin to pin 1 (IOREF 3V3). Let's check the value again:

```
# cat PF11/value
1
```

The value is 1 because our pin is connected to a 3.3V level now.

You could use this GPIO to add a button switch to your board, for example.

Note that you could also configure the pin as output and set its value through the `value` file. This way, you could add an external LED to your board, for example.

Before moving on to the next section, you can also check `/sys/kernel/debug/gpio` again, and see that `gpio-601` is now in use, through the sysfs interface, and is configured as an input pin.

When you're done, you can see your GPIO free:

```
# echo 601 > unexport
```

[8] https://www.st.com/en/evaluation-tools/stm32mp257f-dk.html

Driving LEDs

First, make sure your kernel is compiled with `CONFIG_LEDS_CLASS=y`, `CONFIG_LEDS_GPIO=y` ,`CONFIG_EXPERT=y`, `CONFIG_I2C_STM32F7=y` and `CONFIG_LEDS_TRIGGER_TIMER=y`.

Then, go to `/sys/class/leds` to see all the LEDs that you are allowed to control.

Let's control the LED which is called `heartbeat`.

Go into the directory for this LED, and check its trigger (what routine is used to drive its value):

```
# cat trigger
```

As you can see, there are many triggers to choose from, the current being `heartbeat`, corresponding to the CPU activity.

You can disable all triggers by:

```
# echo none > trigger
```

And then directly control the LED:

```
# echo 1 > brightness
# echo 0 > brightness
```

You could also use the `timer` trigger to light the LED with specified time on and time off:

```
# echo timer > trigger
# echo 10 > delay_on
# echo 200 > delay_off
```

Managing the I2C buses and devices

Enabling an I2C bus

The next thing we want to do is connect an Nunchuk joystick to an I2C bus on our board. The I2C bus is very frequently used to connect all sorts of external devices. That's why we're covering it here. The first task is to find a suitable bus. If you study the same document about the board, you will find that only I2C2 and I2C8 are conveniently available through the headers pin. Let's try to use I2C8!

First, let's see which I2C buses are already enabled:

```
# i2cdetect -l
i2c-0 i2c  STM32F7 I2C(0x0000000040130000)  I2C adapter
```

`i2c-0` is the I2C controller with registers at `0x0000000040130000`, which is `I2C2` in the STM32MP2 nomenclature. Refer to the STM32MP25 memory map in the datasheet for details. Pay attention to the numbering difference: `i2c-0` is the Linux numbering, based on the registration order of enabled I2C busses. Here, because only I2C2 is enabled, it is called `i2c-0`.

Using the datasheet for the SoC [9], we can find what is the base address of the registers for the `I2C8` controller: it is `0x46040000`.

Customizing the Device Tree

To enable I2C2on our system, we need to assign set `status = "okay";` in the corresponding Device Tree node.

[9] https://www.st.com/resource/en/reference_manual/rm0457-stm32mp25xx-advanced-armbased-3264bit-mpus-stmicroelectronics.pdf

Fortunately, I2C2is already defined in the one of the DTS includes used by the Device Tree for our board. In our case, that's in arch/arm64/boot/dts/st/stm32mp251.dtsi. Look by yourself in this file, and you will find its definition, but with `status = "disabled";`. This means that this I2C controller is not enabled yet, and it's up to boards using it to do so.

We could modify the arch/arm64/boot/dts/st/stm32mp257f-dk.dts file for our board, but that's not a very good idea as this file is maintained by the kernel developers. The changes that you make could collide with future changes made by the maintainers for this file.

A more futureproof idea is to create a new Device Tree file which includes the standard one, and adds custom definitions. So, create a new

`arch/arm64/boot/dts/st/stm32mp257f-dk-custom.dts` file containing:

```
/dts-v1/;
#include "stm32mp257f-dk.dts"

&i2c8 {
        status = "okay";
        /delete-property/ pinctrl-names;
};
```

As you can see, it's also possible to include `dts` files, and not only `dtsi` ones.

Why the `/delete-property/` statement? That's because we want to see what happens when a device doesn't have associated pin definitions yet.

A device like an I2C controller node is typically declared in the DTSI files for the SoC, without pin settings as these are board specific. Pin definitions are then usually defined at board level.

In our case, we don't see such definitions, but they are actually found in the arch/arm64/boot/dts/st/stm32mp251.dtsi file, shared between multiple stm32mp25 boards, which is included by the toplevel Device Tree for our board.

Modify the arch/arm64/boot/dts/st/Makefile file to add your custom Device Tree, and then have it compiled (`make dtbs`).

Reboot your board with the update.

Back to the running system, we can now see that there is one more I2C bus. We can also recognize the I2C8 address (`0x46040000`) though it's now associated to the `i2c-0` device name, which already existed previously, but mapped to the I2C2 controller.

```
# i2cdetect -l
i2c-1 i2c  STM32F7 I2C(0x0000000046040000)  I2C adapter
i2c-0 i2c  STM32F7 I2C(0x0000000040130000)  I2C adapter
```

Now, let's use `i2cdetect`'s capability to probe a bus for devices. Let's start by the bus associated to `i2c-0`:

```
i2cdetect -r 0
i2cdetect: WARNING! This program can confuse your I2C bus
Continue? [y/N] y
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- UU -- -- -- UU UU -- UU
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
```

```
70: -- -- -- -- -- -- -- --
```

We can see four devices on this internal bus at the addresses `0x38`, `0x3c`, `0x3d`, and `0x3f`, indicated by `UU`, which means that a kernel driver is actively managing these devices.

Now try to probe I2C2through `i2cdetect -r 1`.

You will see that the command will run very slowly because of failing to connect to the bus. That's because the corresponding signals are not exposed yet to the outside connectors through pin muxing.

So, get back to your Device Tree and remove the `/delete-property/` line. Recompile your Device Tree and reboot.

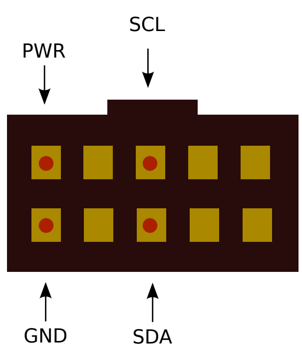
You should now be able to probe your bus:

```
# i2cdetect -r 1
i2cdetect: WARNING! This program can confuse your I2C bus
Continue? [y/N] y
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

No device is detected yet, because this bus is just used for external devices. It's time to add one though.

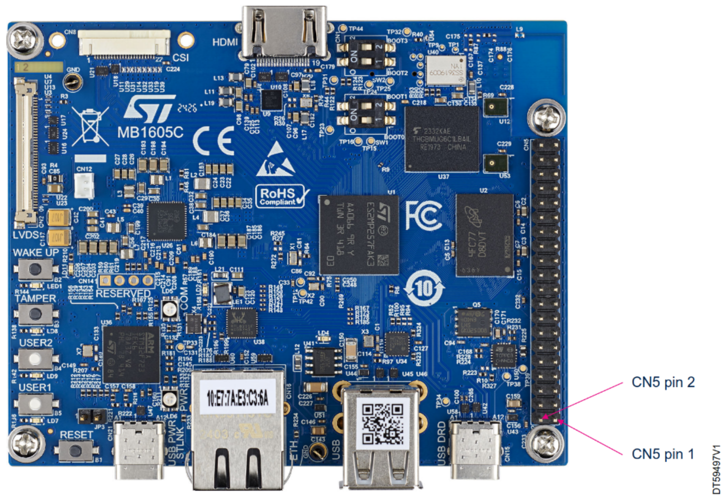
Adding and enabling an I2C device

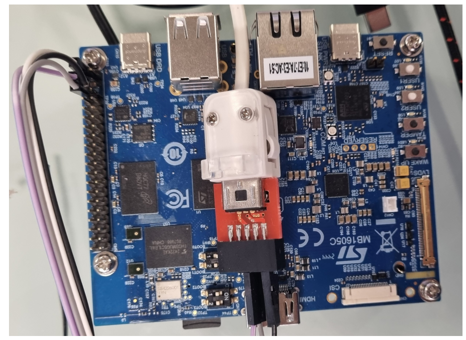
Let's connect the Nunchuk provided by your instructor to the I2C2bus on the board, using breadboard wires:



Nunchuk i2c pinout
(UEXT connector from Olimex, front view)

Figure 19. GPIO connectors





- Connect the Nunchuk PWR pin to pin 1 (3V3).
- Connect the Nunchuk GND pin to pin 6 (GND).
- Connect the Nunchuk SCL pin to pin 5 (PZ4 - I2C8_SCL).
- Connect the Nunchuk SDA pin to pin 3 (PZ3 - I2C8_SDA).

If you didn't do any mistake, your new device should be detected at address `0x52`:

```
# i2cdetect -r 1
i2cdetect: WARNING! This program can confuse your I2C bus
Continue? [y/N] y
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- 52 -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

We will later compile an out-of-tree kernel module to support this device.

Plugging a USB audio headset

In the next labs, we are going to play audio using a USB audio headset. Let's see whether our kernel supports such hardware by plugging the headset provided by your instructor.

Before plugging the device, look at the output of `lsusb`:

```
# lsusb
Bus 001 Device 001: ID 1d6b:0002 Linux 6.6.48-g3682d604ecbd ehci_hcd EHCI Host Controller
Bus 001 Device 002: ID 0424:2422
```

Now, when you plug the USB headset, a number of messages should appear on the console, and running `lsusb` again should show an additional device:

```
# lsusb
Bus 001 Device 001: ID 1d6b:0002 Linux 6.6.48-g3682d604ecbd ehci_hcd EHCI Host Controller
Bus 001 Device 003: ID 1b3f:2008 GeneralPlus USB Audio Device
Bus 001 Device 002: ID 0424:2422
```

You can see the vendor and product ID in the form `1a2b:1234`. Of course, this depends on the actual USB audio device that you used.

The device also appears in `/sys/bus/usb/devices/`, in a directory whose name depends on the topology of the USB bus. When the device is plugged in the kernel messages show:

```
usb 1-1.X: new full-speed USB device number X using ehci-platform
```

So if we go in `/sys/bus/usb/devices/1-1.X`, we get the *sysfs* representation of this USB device:

```
# cd /sys/bus/usb/devices/1-1.X
# cat idVendor
1a2b
# cat idProduct
1234
# cat manufacturer
C-Media Electronics Inc.
# cat product
USB Audio Device
```

However, while the USB device is detected, we currently do not have any driver for this device, so no actual sound card is detected.

Enabling, installing and using in-tree kernel modules

Go back to the kernel source directory.

The Linux kernel has a generic driver supporting all USB audio devices supporting the standard USB audio class. This driver can be enabled using the `CONFIG_SND_USB_AUDIO` configuration option. Look for this parameter in the kernel configuration, and you should find that it is already enabled as a module.

So, instead of compiling the corresponding driver as a built-in, that's a good opportunity to practice with kernel modules.

So, compile your modules:

```
make modules
```

Then, following details given in the lectures, install the modules in our NFS root filesystem (`$HOME/embedded-linux-stm32mp2-labs/tinysystem/nfsroot`).

Also make sure to update the kernel image (`make zImage`), and reboot the board. Indeed, due to the changes we have made to the kernel source code, the kernel version is now `6.6.<x>-dirty`, the *dirty* keyword indicating that the Git working tree has uncommitted changes. The modules are therefore installed in `/lib/modules/6.6.<x>-dirty/`, and the version of the running Linux kernel must match this.

After rebooting, try to load the module that we need:

```
modprobe snd-usb-audio
```

By running `lsmod`, see all the module dependencies that were loaded too.

You can also see that a new USB device driver in `/sys/bus/usb/drivers/snd-usb-audio`. This directory shows which USB devices are bound to this driver.

You can check that `/proc/asound` now exists (thanks to loading modules for ALSA, the Linux sound subsystem), and that one sound card is available:

```
# cat /proc/asound/cards
 0 [Device ]: USB-Audio - USB Audio Device
                   C-Media Electronics Inc. USB Audio Device at usb-5800d000.usb-1.1, full \
                       speed
```

Check also the `/dev/snd` directory, which should now contain some character device files. These will be used by the user-space libraries and applications to access the audio devices.

Modify your startup scripts so that the `snd-usb-audio` module is always loaded at startup.

We cannot test the sound card yet, as we will need to build some software first. Be patient, this is coming soon.

Compiling and installing an out-of-tree kernel module

The next device we want to support is the I2C Nunchuk. There is a driver in the kernel to support it when connected to a Wiimote controller, but there is no such driver to support it as an I2C device.

Fortunately, one is provided in `$HOME/embedded-linux-stm32mp2-labs/hardware/data/nunchuk/nunchuk.c`. You can check Bootlin's Linux kernel and driver development course to learn how to implement all sorts of device drivers for Linux.

Go to this directory, and compile the out-of-tree module as follows:

```
make -C $HOME/embedded-linux-stm32mp2-labs/kernel/linux M=$PWD
```

Here are a few explanations:

- The `-C` option lets `make` know which Makefile to use, here the toplevel Makefile in the kernel sources.
- `M=$PWD` tells the kernel Makefile to build external module(s) from the file(s) in the current directory.

Now, you can install the compiled module in the NFS root filesystem by passing the `modules_install` target and specifying the target directory through the `INSTALL_MOD_PATH` variable:

```
make -C $HOME/embedded-linux-stm32mp2-labs/kernel/linux \
    M=$PWD \
    INSTALL_MOD_PATH=$HOME/embedded-linux-stm32mp2-labs/tinysystem/nfsroot \
    modules_install
```

You can see that this installs out-of-tree kernel modules under `lib/modules/<version>/updates/`.

Back on the target, you can now check that your custom module can be loaded:

```
# modprobe nunchuk
[ 4317.737978] nunchuk: loading out-of-tree module taints kernel.
```

See `kbuild/modules` in kernel documentation for details about building out-of-tree kernel modules.

However, run `i2cdetect -r 1` again. You will see that the Nunchuk is still detected, but still not driven by the kernel. Otherwise, it would be signaled by the `UU` character. You may also look at the `nunchuk.c` file and notice a `Nunchuk device probed successfully` message that you didn't see when loading the module.

That's because the Linux kernel doesn't know about the Nunchuk device yet, even though the driver for this kind of devices is already loaded. Our device also has to be described in the Device Tree.

You can confirm this by having a look at the contents of the `/sys/bus/i2c` directory. It contains two subdirectories: `devices` and `drivers`.

In `drivers`, there should be a `nunchuk` subdirectory, but no symbolic link to a device yet. In `devices` you should see some devices, but not the Nunchuk one yet.

Declaring an I2C device

To allow the kernel to manage our Nunchuk device, let's declare the device in the custom Device Tree for our board. The declaration of the I2C2bus will then look as follows:

```
&i2c2 {
        status = "okay";
        clock-frequency = <100000>;

        nunchuk: joystick@52 {
                compatible = "nintendo,nunchuk";
                reg = <0x52>;
        };
};
```

Here are a few notes:

- The `clock-frequency` property is used to configure the bus to operate at 100 KHz. This is supposed to be required for the Nunchuk.
- The Nunchuk device is added through a child node in the I2C controller node.
- For the kernel to *probe* and drive our device, it's required that the `compatible` string matches one of the `compatible` strings supported by the driver.
- The `reg` property is the address of the device on the I2C bus. If it doesn't match, the driver will probe the device but won't be able to communicate with it.

Recompile your Device Tree and reboot your kernel with the new binary.

You can now load your module again, and this time, you should see that the Nunchuk driver probed the Nunchuk device:

```
# modprobe nunchuk
[ 66.680455] nunchuk: loading out-of-tree module taints kernel.
[ 66.687645] input: Wii Nunchuk as /devices/platform/soc/40015000.i2c/i2c-1/1-0052/input/\
    input3
[ 66.695421] Nunchuk device probed successfully
```

List the contents of `/sys/bus/i2c/drivers/nunchuk` once again. You should now see a symbolic link corresponding to our new device.

Also list `/sys/bus/i2c/devices/` again. You should now see the Nunchuk device, which can be recognized through its `0052` address. Follow the link and you should see a symbolic link back to the Nunchuk driver!

We are not ready to use this input device yet, but at least we can test that we get bytes when buttons or the joypad are used. In the below command, use the same number as in the message you got in the console (`event3` for `input3` for example):

```
# cat /dev/input/event3 | od -x
```

Caution: using `od` directly on input event files should work but is currently broken with the Musl library. We are investigating this issue.

We will use the Nunchuk to control audio playback in an upcoming lab.

Setting the board's model name

Modify the custom Device Tree file one last time to override the model name for your system. Set the `model` property to stm32mp2media player. Don't hesitate to ask your instructor if you're not sure how.

Recompile the device tree, and reboot the board with it. You should see the new model name in two different places:

- In the first kernel messages on the serial console.
- In `/sys/firmware/devicetree/base/model`. This can be handy for a distribution to identify the device it's running on. By the way, you can explore `/sys/firmware/devicetree` and find that every subdirectory corresponds to a DT node, and every file corresponds to a DT property.

Committing kernel tree changes

Now that our changes to the kernel sources are over, create a branch for your changes and create a patch for them. **Please don't skip this step** as we need it for the next labs.

First, if not done yet, you should set your identity and e-mail address in git:

```
git config --global user.email "linus@bootlin.com"
git config --global user.name "Linus Torvalds"
```

This is necessary to create a commit with the `git commit -s` command, as required by the Linux kernel contribution guidelines.

Let's create the branch and the patch now:

```
git checkout -b bootlin-labs
git add arch/arm64/boot/dts/st/stm32mp257f-dk-custom.dts arch/arm64/boot/dts/st/Makefile
git commit -as -m "Custom DTS for Bootlin lab"
```

We can now create the patch:
`git format-patch -1`

This should generate a `0001-Custom-DTS-for-Bootlin-lab.patch` file.

Creating the branch will impact the versions of the kernel and the modules. Compile your kernel and install your modules again (not necessary for the Nunchuk one for the moment) and see the version changes through the new base directory for modules.

To save space for the next lab, remove the old directory under `lib/modules` containing the "dirty" modules.

Don't forget to update the kernel your board boots.

That's all for now!

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.

Setup

Go to the `$HOME/embedded-linux-stm32mp2-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://gitlab.com/buildroot.org/buildroot.git
cd buildroot
```

Now checkout the tag corresponding to the latest 2025.02.<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.

Board specific configuration

As we will want Buildroot to build a kernel with a custom configuration, and our custom patch, so let's add our own subdirectory under `board`:

```
mkdir -p board/bootlin/training
```

Then, copy your kernel configuration and kernel patch:

```
cp ../../kernel/linux/.config board/bootlin/training/linux.config
cp ../../kernel/linux/0001-Custom-DTS-for-Bootlin-lab.patch \
   board/bootlin/training/
```

We will configure Buildroot to use this kernel configuration.

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`: `AArch64 (little endian)`
  - `Target Architecture Variant`: `cortex-A35`
  - `Floating point strategy`: `FP-ARMv8`
- `Toolchain`
  - `Toolchain type`: `External toolchain`
  - `Toolchain`: `Custom toolchain`
  - `Toolchain path`: use the toolchain you built: `/home/<user>/x-tools/aarch64-training-linux-musl` (replace `<user>` by your actual user name)
  - `External toolchain gcc version`: `14.x`
  - `External toolchain kernel headers series`: Set it to match the version of the kernel headers chosen when building the toolchain, or chose 6.12.x or later if the kernel headers used in the toolchain are indeed 6.12 or newer.
  - `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 `Custom tarball`
  - Set `URL of custom kernel tarball` to `$(call github,STMicroelectronics,linux)v6.6-stm32mp-r1.tar.gz`
  - Set `Custom kernel patches` to `board/bootlin/training/0001-Custom-DTS-for-Bootlin-lab.patch`
  - Set `Kernel configuration` to `Using a custom (def)config file)`
  - Set `Configuration file path` to `board/bootlin/training/linux.config`
  - Select `Build a Device Tree Blob (DTB)`
  - Set `In-tree Device Tree Source file names` to `st/stm32mp257f-dk-custom`
- `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:
      - Only keep `speaker-test`
    - Select `mpd`, and in the submenu:
      - Keep only `alsa`, `vorbis` and `tcp sockets`
    - Select `mpd-mpc`.
  - `Hardware handling`
    - Select `evtest`
      This userspace application allows to test events from input devices. This way, we will be able to test the Nunchuk by getting details about which buttons were pressed.
- `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-stm32mp2-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.

Loading the USB audio module

You can check that no kernel module is loaded yet. Try to load the `snd_usb_audio` module from the command line.

This should work. Check that Buildroot has deployed the modules for your kernel in `/lib/modules`.

Let’s automate this now!

Look at the `/etc/inittab` file generated by Buildroot (ask your instructor if you have any questions), and at the contents of the `/etc/init.d/` directory, in particular of the `rcS` file.

You can see that `rcS` executes or sources all the `/etc/init.d/S??*` files. We can add our own which will load the toplevel modules that we need.

Let’s do this by creating an *overlay directory*, typically under our board specific directory, that Buildroot will add after building the root filesystem:

```
mkdir -p board/bootlin/training/rootfs-overlay/
```

Then add a custom startup script, by adding an `etc/init.d/S03modprobe` executable file to the overlay directory, with the below contents:

```
#!/bin/sh
modprobe snd-usb-audio
```

Then, go back to Buildroot’s configuration interface:

- `System configuration`
  - Set `Root filesystem overlay directories` to `board/bootlin/training/rootfs-overlay`

Build your image again. This should be quick as Buildroot doesn’t need to recompile anything. It will just apply the root filesystem overlay.

Update your `nfsroot` directory, reboot the board and check that the `snd_usb_audio` module is loaded as expected.

You can run `speaker-test` to check that audio indeed works.

Testing music playback with mpd and mpc

The next thing we want to do is play real sound samples with the *Music Player Daemon (MPD)*. So, let's add music files [10] for MPD to play:

```
mkdir -p board/bootlin/training/rootfs-overlay/var/lib/mpd/music
cp ../data/music/* board/bootlin/training/rootfs-overlay/var/lib/mpd/music
```

Update your root filesystem. Thanks to NFS, you don't need to restart your system.

Using the `ps` command, check that the `mpd` server was started by the system, as implemented by the `/etc/init.d/S95mpd` script.

If that's the case, you are now ready to run `mpc` client commands to control music playback. First, let's make `mpd` process the newly added music files. Run this command on the target:

```
# mpc update
```

You should see the files getting indexed, by displaying the contents of the `/var/log/mpd.log` file:

```
Jan 01 00:04 : exception: Failed to open '/var/lib/mpd/state': No such file or directory
Jan 01 00:15 : update: added /2-arpent.ogg
Jan 01 00:15 : update: added /6-le-baguette.ogg
Jan 01 00:15 : update: added /4-land-of-pirates.ogg
Jan 01 00:15 : update: added /3-chronos.ogg
Jan 01 00:15 : update: added /1-sample.ogg
Jan 01 00:15 : update: added /7-fireworks.ogg
Jan 01 00:15 : update: added /5-ukulele-song.ogg
```

You can also check the list of available files:

```
# mpc listall
1-sample.ogg
2-arpent.ogg
5-ukulele-song.ogg
3-chronos.ogg
7-fireworks.ogg
6-le-baguette.ogg
4-land-of-pirates.ogg
```

To play files, you first need to create a playlist. Let's create a playlist by adding all music files to it:

```
# mpc add /
```

You should now be able to start playing the songs in the playlist:

```
# mpc play
```

Here are a few further commands for controlling playback:

- `mpc toggle`: toggle between pause and playback modes.
- `mpc next`: switch to the next song in the playlist.

[10] For the most part, these are public domain music files, except a small sample file... See the `README.txt` file in the directory containing the files.

- `mpc prev`: switch to the previous song in the playlist.
- `mpc volume +5`: increase the volume by 5%
- `mpc volume -5`: reduce the volume by 5%

The volume control commands won't work right away. You probably noticed the following error logs when MPD started:

```
Starting mpd: Jan 01 00:00 : server_socket: bind to '0.0.0.0:6600' failed (continuing anyway,
because binding to '[::]:6600' succeeded): Failed to bind socket: Address
in use
Jan 01 00:00 : exception: Failed to open '/var/lib/mpd/database': No such file or directory
Jan 01 00:00 : output: No 'audio_output' defined in config file
Jan 01 00:00 : output: Successfully detected a alsa audio device
```

These are due to invalid configuration options. We will add a custom configuration for MPD, as the standard one provided by Buildroot doesn't perfectly fit our use case. We will simply add this file to our overlay:

```
cp ../data/mpd.conf board/bootlin/training/rootfs-overlay/etc/
```

Run Buildroot again and update your root filesystem. Here again, you don't need to reboot. It's sufficient to restart MPD to make it read the new configuration file:

```
# /etc/init.d/S95mpd restart
```

You can now make sure that modifying the volume works.

Later, we will compile and debug a custom MPD client application.

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.

Adding a Buildroot package

We would also like to build our Nunchuk external module with Buildroot. Fortunately, Buildroot has a `kernel-module` infrastructure to build kernel modules.

First, create a `nunchuk-driver` subdirectory under `package` in Buildroot sources.

The first thing is to create a `package/nunchuk-driver/Config.in` file for Buildroot's configuration:

```
config BR2_PACKAGE_NUNCHUK_DRIVER
        bool "nunchuk driver"
        depends on BR2_LINUX_KERNEL
        help
                Linux Kernel module for the I2C Nunchuk.
```


Then add a line to `package/Config.in` to include this file, for example right before the line including `package/nvidia-driver/Config.in`, so that the alphabetic order of configuration options is kept.

Then, the next and last thing you need to do is create `package/nunchuk-driver/nunchuk-driver.mk` describing how to build the package:

```
NUNCHUK_DRIVER_VERSION = 1.0
NUNCHUK_DRIVER_SITE = $(HOME)/embedded-linux-stm32mp2-labs/hardware/data/nunchuk
NUNCHUK_DRIVER_SITE_METHOD = local
NUNCHUK_DRIVER_LICENSE = GPL-2.0

$(eval $(kernel-module))
$(eval $(generic-package))
```

Then, configure Buildroot to build your package, run Buildroot and update your root filesystem.

Can you load the `nunchuk` module now? If everything's fine, add a line to `/etc/init.d/S03modprobe` for this driver, and update your root filesystem once again.

Testing the Nunchuk

Now that we have the `nunchuk` driver loaded and that Buildroot compiled `evtest` for the target, thanks to Buildroot, we can now test the input events coming from the Nunchuk.

```
# evtest
No device specified, trying to scan all of /dev/input/event*
Available devices:
/dev/input/event0: pmic_onkey
/dev/input/event1: Logitech Inc. Logitech USB Headset H340 Consumer Control
/dev/input/event2: Logitech Inc. Logitech USB Headset H340
/dev/input/event3: Wii Nunchuk
Select the device event number [0-3]:
```

Enter the number corresponding to the Nunchuk device.

You can now press the Nunchuk buttons, use the joypad, and see which input events are emitted.

By the way, you can also test which input events are exposed by the driver for your audio headset (if any), which doesn't mean that they physically exist.

Commit your changes

As we are going to reuse our Buildroot changes in the next labs, let's commit them into a branch:

```
git checkout -b bootlin-labs
git add board/bootlin/ package/nunchuk-driver/ package/Config.in
git commit -as -m "Bootlin lab changes"
```

Going further

If you finish your lab before the others

- For more music playing fun, you can install the `ario` or *cantata* MPD client on your host machine (`sudo apt install ario`, `sudo apt install cantata`), configure it to connect to the IP address of your target system with the default port, and you will also be able to control playback from your host machine.

System Integration - Using systemd

Objectives: Get familiar with the systemd *init system.*

Goals

Compared to the previous lab, we go on increasing the complexity of the system, this time by using the *systemd* init system, and by taking advantage of it to add a few extra features, in particular ones that will be useful for debugging in the next lab.

Setup

Since *systemd* requires the GNU C library, we are going to make a new Buildroot build in a new working directory, and using a different cross-compiling toolchain.

So, enter the `$HOME/embedded-linux-stm32mp2-labs/integration` directory.

Make a new clone of Buildroot from the existing local Git repository, and checkout our `bootlin-labs` branch:

```
git clone $HOME/embedded-linux-stm32mp2-labs/buildroot/buildroot
cd buildroot
git checkout bootlin-labs
```

Root filesystem overlay

Remove `etc/init.d/` from the root filesystem overlay. It was adapted to *BusyBox init*, not to *systemd*:

```
rm -r board/bootlin/training/rootfs-overlay/etc/init.d/
```

Buildroot configuration

Configure Buildroot as follows:

- `Target options`
  - Select the same architecture and CPU settings as in the previous lab.
- `Toolchain`
  - `Toolchain type`: `External toolchain`
  - `Toolchain`: `Bootlin toolchains`
    This time, we will use a Bootlin ready-made toolchain for *glibc*, as this is necessary for using *systemd*.
  - `Toolchain origin`: `Toolchain to be downloaded and installed`
  - `Bootlin toolchain variant`: `aarch64 glibc bleeding-edge 2024.05-1`
  - Select `Copy gdb server to the Target`
- `System configuration`
  - `Init system`: `systemd`
  - `Root filesystem overlay directories`: `board/bootlin/training/rootfs-overlay`
- `Kernel`

  - Enable `Linux Kernel`
  - Set `Kernel version` to `Custom tarball`
  - Set `URL of custom kernel tarball` to `$(call github,STMicroelectronics,linux)v6.6-stm32mp-r1.tar.gz`
  - Set `Custom kernel patches` to `board/bootlin/training/0001-Custom-DTS-for-Bootlin-lab.patch`
  - Set `Kernel configuration` to `Using a custom (def)config file)`
  - Set `Configuration file path` to `board/bootlin/training/linux.config`
  - Select `Build a Device Tree Blob (DTB)`
  - Set `In-tree Device Tree Source file names` to `st/stm32mp257f-dk-custom`
- `Target packages`
  - `Audio and video applications`
    * We won't need `alsa-utils` this time.
    * Select `mpd`, and in the submenu:
      · Keep only `alsa`, `vorbis` and `tcp sockets`
    * Select `mpd-mpc`.
  - `Hardware handling`
    * Select `nunchuk driver`
  - `Networking applications`
    * Select `dropbear`, a lightweight SSH server used instead of OpenSSH in most embedded devices. You don't need to enable client support (building an SSH client).
- `Filesystem images`
  - Select `tar the root filesystem`

Build and test the new system

Now build the full system.

Once the build is over, generate the dependency graph again and find out the new dependencies introduced by using *systemd*.

To test the new system, create a new `nfsroot` directory, extract the new root filesystem into it, and boot your board on it through NFS.

You should see the system booting through *systemd*, with all the *systemd* targets and system services starting one by one, with a total boot time which looks slower than before. That's because the system configuration is more complex, but also more versatile, being ready to run more complex services and applications.

You can ask *systemd* to show you the various services which were started:

```
# systemctl status
```

You can also check all the mounted filesystems and be impressed:

```
# mount
```


Inspecting the system

On the target, look at the contents of /lib/systemd. You will see the implementation of most *systemd* targets and services.

In particular, check out /lib/systemd/user/ containing some unnecessary targets in our case such as bluetooth.target.

However, check the mpd.service file for our MPD server. This should help you to realize all the options provided by *systemd* to start and control system services, while keeping the system secure and their resources under control.

You won't be able to match this level of control and security in a "hand-made" system.

Note: you may notice that a "systemd-network-generator" unit fails to start. It is due to systemd failing to parse correctly the ip *parameter from the kernel commandline. You can circumvent this issue by setting the* autoconf *field (currently not set at all) of the* ip *parameter to* none*. You can refer to nfsroot documentation to learn more about this option.*

Understanding automatic module loading with Udev

Check the currently loaded modules on your system. Surprise: both the Nunchuk and USB audio modules are already loaded. We didn't have anything to set up and *systemd* automatically loaded the modules associated to connected hardware.

Let's find out why...

On the target, go to /lib/udev/rules.d. You will find all the standard rules for *Udev*, the part of *systemd* which handles hardware events, takes care of the permissions and ownership of device files, notifies other userspace programs, and among others, load kernel modules.

Open 80-drivers.rules, which is the rule allowing *Udev* to load kernel modules for detected devices. Here is its most important line:

ENV{MODALIAS}=="?*", RUN{builtin}+="kmod load '$env{MODALIAS}'"

This is when the modules.alias file comes into play. When a new device is found, the kernel passes a MODALIAS environment variable to *Udev*, containing which bus this happened on and the attributes of the device on this bus. Thanks to the module aliases, the right module gets loaded. We already explained that in the lectures when talking about the output of make modules_install.

Find where the modules.alias file is located and you will find the two lines that allowed to load our snd_usb_audio and nunchuk modules:

```
...
alias usb:v*p*d*dc*dsc*dp*ic01isc01ip*in* snd_usb_audio
alias usb:v2B53p0031d*dc*dsc*dp*ic*isc*ip*in* snd_usb_audio
...
alias of:N*T*Cnintendo,nunchuk nunchuk
```

For snd_usb_audio, there are many possible matching values, so it's not straightforward to be sure which matched your particular device.

However, you can find in *sysfs* which MODALIAS was emitted for your device:

```
# cd /sys/class/sound/card0/device
# ls -la
# cat modalias
usb:v1B3Fp2008d0100dc00dsc00dp00ic01isc01ip00in00
```

With a bit of patience, you could find the matching line in the modules.alias file.

If you want to see the information sent to *Udev* by the kernel when a new device is plugged in, here are a few debugging commands.

First unplug your device and run:

```
# udevadm monitor
```

Then plug in your headset again. You will find all the events emitted by the kernel, and with the same string (with `UDEV` instead of `KERNEL`), the time when *Udev* finished processing each event.

You can also see the `MODALIAS` values carried by these events:

```
# udevadm monitor --env
```

As far as the Nunchuk is concerned, we cannot easily remove it from the Device Tree and add it back, but it's easier to find its `MODALIAS` value:

```
# cd /sys/bus/i2c/devices
# ls -la
```

Here you will recognize our Nunchuk device through its `0x52` address.

```
# cd 1-0052
# ls -la
# cat modalias
of:NjoystickT(null)Cnintendo,nunchuk
```

Here the bus is `of`, meaning *Open Firmware*, which was the former name of the Device Tree. When an event was emitted by the kernel with this `MODALIAS` string, the `nunchuk` module got loaded by *Udev* thanks to the matching alias.

This actually happened when *systemd* ran the *coldplugging* operation: at system startup, it asked the kernel to emit hotplug events for devices already present when the system booted:

```
[  OK  ] Finished Coldplug All udev Devices.
```

On non-x86 platforms, that's typically for devices described in the Device Tree. This way, both *static* and *hotplugged* devices can be handled in the same way, using the same *Udev* rules.

Testing your system

Make sure that audio playback still works on your system:

```
# mpc update
# mpc add /
# mpc play
```

If it doesn't, look at the *systemd* logs in your serial console history. *systemd* should let you know about the failing services and the commands to run to get more details.

Application development and application debugging

Objective: compile an application against a Buildroot build space and debug it remotely.

Setup

We will continue to use the same root filesystem.

Our goal is to compile and debug our own *MPD* client. This client will be driven by the Nunckuk to switch between audio tracks, and to adjust the playback volume.

However, this client will be used together with `mpc`, as it won't be able to create the playlist and start the playback. It will just be used to control the volume and switch between songs. So, you need to run `mpc` commands first before trying the new client:

```
mpc update
mpc add /
mpc pause
```

We will use the new client to resume playback.

Compile your own application

Go to the `$HOME/embedded-linux-stm32mp2-labs/appdev` directory.

In the lab directory the file `nunchuk-mpd-client.c` contains an application which implements a simple MPD client based on the *libmpdclient* library. As *mpc* is also based on this library, Buildroot already compiled it and added it to our root filesystem. What's special in this application is that it allows to drive music playback through our Nunchuk.

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). This way, we can compile our application outside of Buildroot, as often as we want.

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

```
$ export PATH=$HOME/embedded-linux-stm32mp2-labs/integration/buildroot/output/host/bin:$PATH
```

Let's try to compile the application:

```
$ aarch64-linux-gcc -o nunchuk-mpd-client nunchuk-mpd-client.c
```

The compiler complains about undefined references to some symbols in *libmpdclient*. This is normal, since we didn't tell the compiler to link with this library. So let's use `pkg-config` to query the *pkg-config* database about the list of libraries needed to build an application against *libmpdclient*[11]:

[11] Normally, `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, but here there is an open issue with Buildroot 2024.02 which forced us to set `PKG_CONFIG_PATH` to make it point to where the `.pc` files are found.

```
$ export PKG_CONFIG_PATH=$HOME/embedded-linux-stm32mp2-labs/integration/buildroot/output/host\
    /aarch64-buildroot-linux-gnueabihf/sysroot/usr/lib/pkgconfig
$ aarch64-linux-gcc -o nunchuk-mpd-client nunchuk-mpd-client.c \
$(pkg-config --libs libmpdclient)
```

Copy the `nunchuk-mpd-client` executable to the `/root` directory of the root filesystem, and then strip it.

Back to target system, try to run the program:

```
# /root/nunchuk-mpd-client
ERROR: didn't manage to find the Nunchuk device in /dev/input. Is the Nunchuk driver loaded?
```

Enable debugging tools

In order to debug our application, let's make Buildroot build some debugging tools for our root filesystem. This is also an opportunity to enable `perf`, that we are using later on during this lab. Go back to the Buildroot configuration interface and enable the following options:

- `Kernel`
  - In `Linux Kernel Tools`, select `perf`
- `Target packages`
  - `Debugging, profiling and benchmark`
    - Select `ltrace`
    - Select `strace`

Then rebuild and update your NFS root filesystem.

Using strace

Let's run the program through the `strace` command to find out why this happens.

You should see that it's trying to access files that don't exist. Once you've found what's wrong, fix the code (or ask your instructor for help if needed), then rebuild the program and run it again:

```
# /root/nunchuk-mpd-client
ERROR: didn't manage to find the Nunchuk device in /dev/input. Is the Nunchuk driver loaded?
```

Ouch, same problem again!

You can run the program again through `strace`, and check that the right paths are now accessed, but the cause of the issue won't be easy to find.

Using ltrace

Let's run the program through `ltrace` now. We will be able to see the shared library calls.

Take your time to study the `ltrace` output. That's interesting information! Back to our issue, the last lines of output should make the issue pretty obvious.

Fix the bug in the code, recompile the program, copy it to the target, strip it and start it again.

You should now be able to use the new client, driving the server through the following Nunchuk inputs:

- Joystick up: volume up 5%
- Joystick down: volume down 5%
- Joystick left: previous song
- Joystick right: next song

- Z (big) button: pause / play
- C (small) button: quit client

Have fun with the new client. You'll just realize that quitting causes the program to crash with a segmentation fault. Let's debug this too.

Using gdbserver from the command line

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

Compile `nunchuk-mpd-client.c` again with the `-g` (`g` means *gdb*) 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 the program under `gdbserver`. `gdbserver` will listen on a TCP port for a connection from `gdb` on the host, and will control the execution of `nunchuk-mpd-client` according to the `gdb` commands:

```
=> gdbserver localhost:2345 /root/nunchuk-mpd-client
```

On the host side, run `aarch64-linux-gdb` (also found in your toolchain):

```
$ aarch64-linux-gdb nunchuk-mpd-client
```

`gdb` starts and loads the debugging information from the `nunchuk-mpd-client` binary (in the `appdev` directory) which 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-stm32mp2-labs/integration/\
    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.

In our case, we'll just start the program, press the `C` button to quit to cause the the segmentation fault:

```
(gdb) continue
```

After the segmentation fault, you can 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 `libmpdclient`, called by our program. You will also get the number of the line in the program which caused this. This should help you to find the bug in our application.

Once you found it, don't fix it yet. We are going to make further experiments around this segmentation fault.

Post mortem analysis

By default `systemd` disables generating `core` files, so we need to re-enable the generation of `core` files by running on the target:

- `echo core.%p > /proc/sys/kernel/core_pattern`, which will replace the `|/bin/false` that was set by *systemd* during boot
- `ulimit -c unlimited` to make sure no size limit is imposed on core files

Run `nunchuk-mpd-client` again, and exit by pressing `C` on the joystick, it should generate the crash, which in turn should cause a `core.<pid>` file to be generated.

Once you have such a file, inspect it with `aarch64-linux-gdb` on the host as explained in the lectures.

Don't be surprised, the below warnings are expected:

```
warning: Can't open file /root/nunchuk-mpd-client during file-backed mapping note processing
warning: Can't open file /usr/lib/libc.so.6 during file-backed mapping note processing
warning: Can't open file /usr/lib/libmpdclient.so.2.22 during file-backed mapping note processing
warning: Can't open file /usr/lib/ld-linux-armhf.so.3 during file-backed mapping note processing
warning: core file may not match specified executable file.
```

In the `gdb` shell, set the `sysroot` setting as previously, and then generate a backtrace to see where the program crashed. You can even see the value of all variables in the different function contexts of your program:

```
(gdb) bt full
```

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

Editing and remote compiling with VS Code

Installing software

We are going to use Visual Studio Code to do the remote debugging again, and eventually fix and recompile our program.

The first thing to do is install VS Code. This package is only available as a *snap package*:

```
$ sudo snap install --classic code
```

Preparing the target for debugging with VSCode

We will use Visual Studio Code to modify and recompile our client program, and also to update and run the binary on the target. Of course, we will use a simple solution, as we won't be able to spend too much time learning about all the possibilities offered by VS Code.

For our purpose, a good solution is SSH, which allows to copy files (through the `scp` command) and to run remote commands. We already included the *Dropbear* SSH server in our root filesystem.

We just need to implement password-less SSH access, to keep things simple:

- If you don't have an SSH key yet (look at `~/.ssh/`), generate a password-less one with the `ssh-keygen` command. By default, this creates two files in `~/.ssh/`: `id_rsa` (private key) and `id_rsa.pub` (public key).
- Then create a `/root/.ssh/authorized_keys` file **on the target** with the line in `id_rsa.pub`.
- Then, fix permissions on the target, as Dropbear is quite strict about them:

  ```
  # chmod -R go-rwx /root
  # chown -R root:root /root
  ```

Then, you can test that SSH works without a password:


```
ssh root@192.168.0.100
```

If you face trouble, you can check the Dropbear logs on the target:

```
journalctl -fu dropbear
```

Additionally, before being able to debug the our application on the target, we need to make sure that a gdbserver instance is running and can be accessed by VSCode. We can add a small systemd service to handle this. The minimal code to enable such service looks like this:

```
[Unit]
Description=GDB server for application debugging
After=network.target
Wants=network.target

[Service]
Type=exec
ExecStart=/usr/bin/gdbserver --multi :3333
Restart=always

[Install]
WantedBy=multi-user.target
```

Copy this snippet and paste it in a new `gdbserver.service` file onto the target filesystem, in `/usr/lib/systemd/system/`. Then enable the service and make it start automatically during each boot:

```
systemctl daemon-reload && systemctl enable --now gdbserver
```

Compiling and debugging the program from VS Code

The `appdev` directory already contains a `.vscode` directory with ready made settings for code editing and for compiling and debugging our application. Here are these files:

- `.vscode/c_cpp_properties.json`: settings for the code editor.
- `.vscode/tasks.json`: definition of some standard project management tasks, like "build", "clean" and "deploy" tasks
- `.vscode/launch.json`: these are the settings for remote debugging.
- `.vscode/extensions.json`: some needed extensions pinned as "recommended" so that you can find those easily in the plugins menu
- `.env`: some user-defined to let all the files above know how to access the target.

First, start VS Code:

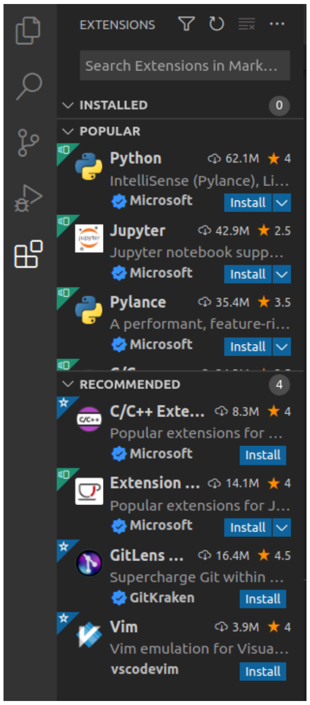
```
$ code
```

Use `File` → `Open Folder` to open the `appdev` directory.

The first thing to do is to make sure that some needed extensions are installed. To do so, switch to the plugins section in the vertical tab, then search and install the following extensions:

- `C/C++` extension from Microsoft (`ms-vscode.cpptools`)
- `Tasks Shell Input` extension (`augustocdias.tasks-shell-input`)

Those extensions should appear on top of the "Recommended" section.

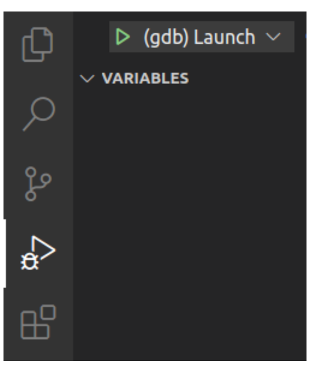


Then open the `.env` file and if needed, update the `TARGET_IP` variable to make it match the IP address of the target.

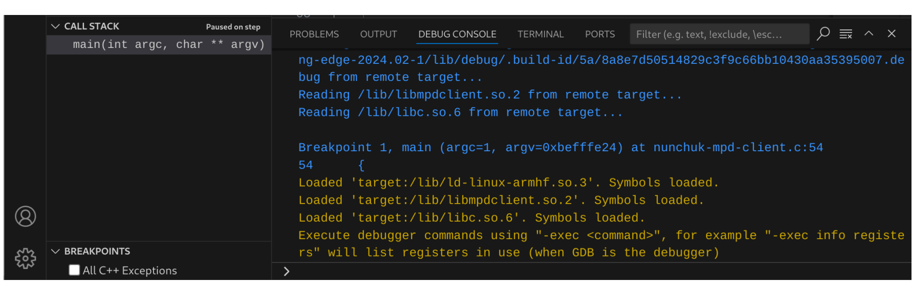
You are now ready to build and debug your application. Start by clicking on the `nunchuk-mpd-client.c` file in the left column to open it in VS Code. Build it thanks to the tasks configured in VSCode: bring the Command Palette by typing `Ctrl+Shift+P`, then type "Run Build Task", then Enter.

Note: you can also use the `Ctrl+Shift+B` *shortcut to execute the default build action*

You can start debugging the program by clicking on the `Run and Debug` tab, and then on the green "Play" button on top:



In the debug console, you should see that debugging has started:



Note: VSCode will automatically rebuild and redeploy the application each time you start a debugging session.

Then, start using the Nunchuk to control playback, and when you try to quit with the `C` button, VS Code should now see the segmentation fault:

You can then look at variables, the call stack, browse the code...

To stop debugging, you should use `Run` → `Stop Debugging`.

By studing the the code, you should eventually find that what's causing the segmentation fault is the call to `free()` in the test for the `C` button. Remove this line, save the file through the `File` menu (otherwise nothing will change), and then compile and run the application again. This time, there should be no more segmentation fault when you hit the `C` button.

If you are ahead of time, don't hesitate to spend more time with VS Code, for example to add breakpoints and execute the program step by step.

Profiling the application with perf

Let's make a quick attempt at profiling our application with the `perf` command:

```
perf record /root/nunchuk-mpd-client
```

Use your application and leave it when you are done.

This stores profiling data in a `perf.data` file. One way to extract information from it is to run the below command in the same directory (the one containing `perf.data`):

```
perf report
```

See the time spent in various kernel (`[k]`) and userspace (`[.]`) functions. The details of the kernel functions is not visible, but additional symbol information can be added by building the kernel with the `CONFIG_KALLSYMS_ALL` option enabled.

Now, let's profile the whole system. First, make sure that the system is currently playing audio. Then SSH to your board and run `perf top` (working better through SSH) to see live information about kernel and userspace functions consuming most CPU time.

This is interactive, but hard to analyze. You can also run `perf record` for about 30 seconds, followed by `perf report` to have a useful summary of system wide activity for a substantial amount of time.

This was a very brief start at practising with `perf`, which offers many more possibilities than we could see here.

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`, `ltrace` and `perf`.
- You can use `perf` as a system wide profiler too.
- You can leave a small `gdbserver` program (about 400 KB) on your target that allows to debug target applications, using a standard `gdb` debugger on the development host, or a graphical IDE such as VS Code.
- 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.

Going further: packaging your application with Meson

Now that our application is ready, the next thing to do is to properly integrate it into our root filesystem. This is a nice opportunity to see how to do this with *Meson* and leverage Buildroot's infrastructure to cross-compile *Meson* based packages.

Still in the main `appdev` directory, create a `nunchuk-mpd-client-1.0` directory and copy the `nunchuk-mpd-client.c` file to it.

In this new directory, all you have to do is create a very simple `meson.build` file:

```
project('nunchuk-mpd-client', 'c', version: '1.0')
libmpdclient_dep = dependency('libmpdclient', version: '>= 2.16')
executable('nunchuk-mpd-client', 'nunchuk-mpd-client.c',
           dependencies: libmpdclient_dep, install: true)
```

Note that `install: true` is necessary to get the executable installed by `ninja install`.

Now, the next thing is to add a new package to the Buildroot source tree:

- Create a `nunchuk-mpd-client` directory under `package`.
- In this directory, create a `Config.in` file. You can reuse the one from the `mpd-mpc` package (the *mpc* client) which also depends on *libmpdclient*.
- Modify `package/Config.in` to source this new file in the `Audio and video applications` submenu.
- Last but not least, create the `nunchuk-mpd-client.mk` file with the following contents:

```
################################################################################
#
# nunchuk-mpd-client
#
################################################################################

NUNCHUK_MPD_CLIENT_VERSION = 1.0
NUNCHUK_MPD_CLIENT_SITE = $(HOME)/embedded-linux-stm32mp2-labs/appdev/nunchuk-mpd-client-1.0
NUNCHUK_MPD_CLIENT_SITE_METHOD = local
NUNCHUK_MPD_CLIENT_DEPENDENCIES = host-pkgconf libmpdclient

$(eval $(meson-package))
```

All you have to do now is to enable the `nunchuk-mpd-client` package in Buildroot's configuration, run `make`, update the root filesystem and check on the target that `/usr/bin/nunchuk-mpd-client` exists and runs fine.

All this was pretty straightforward, wasn't it? *Meson* rocks!

Congratulations, you've reached the end of all our labs. Try to look back, and see how much experience you've gained in these last days.

