Linux kernel and driver development training

BeaglePlay variant

Practical Labs

bootlin

https://bootlin.com

July 03, 2025

About this document

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

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/linux-kernel-beagleplay/linux-kernel-beagleplay-labs.tar.xz
$ tar xvf linux-kernel-beagleplay-labs.tar.xz
```

Lab data are now available in an `linux-kernel-beagleplay-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/`

Downloading kernel source code

Get your own copy of the mainline Linux kernel source tree

Setup

Create the `$HOME/linux-kernel-beagleplay-labs/src` directory.

Installing git packages

First, let's install software packages that we will need throughout the practical labs:

```
sudo apt install git gitk git-email
```

Git configuration

After installing `git` on a new machine, the first thing to do is to let `git` know about your name and e-mail address:

```
git config --global user.name 'My Name'
git config --global user.email me@mydomain.net
```

Such information will be stored in commits. It is important to configure it properly when the time comes to generate and send patches, in particular.

It can also be particularly useful to display line numbers when using the `git grep` command. This can be enabled by default with the following configuration:

```
git config --global grep.lineNumber true
```

Cloning the mainline Linux tree

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

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

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

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

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

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

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

Accessing stable releases

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

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

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

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

```
git branch -a
```

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

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

Kernel source code

Objective: Get familiar with the kernel source code

After this lab, you will be able to:

- Create a branch based on a remote tree to explore a particular stable kernel version (from the `stable` kernel tree).
- Explore the sources and search for files, function headers or other kinds of information…
- Browse the kernel sources with a tool like Elixir.

Choose a particular stable version

Let's work with a particular stable version of the Linux kernel. It would have been more logical to do this in the previous lab, but we wanted to get back to lectures while the `fetch` command was running.

First, let's get the list of branches on our `stable` remote tree:

```
cd ~/linux-kernel-beagleplay-labs/src/linux
git branch -a
```

As we will do our labs with the Linux 6.7 stable branch, the remote branch we are interested in is `remotes/stable/linux-6.7.y`.

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

```
make kernelversion
```

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

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

```
git checkout -b 6.7.bootlin stable/linux-6.7.y
```

Check the version again using the `make kernelversion` command to make sure you now have a 6.7.y version.

Exploring the sources manually

As a Linux kernel user, you will very often need to find which file implements a given function. So, it is useful to be familiar with exploring the kernel sources.

1. Find the Linux logo image in the sources[2].
2. Find who the maintainer of the MVNETA network driver is.
3. Find the declaration of the `platform_device_register()` function.

Tip: if you need the `grep` command, we advise you to use `git grep`. This command is similar, but much faster, doing the search only on the files managed by git (ignoring git internal files and generated files).

Use a kernel source indexing tool

Now that you know how to do things in a manual way, let's use more automated tools.

Try Elixir at `https://elixir.bootlin.com` and choose the Linux version closest to yours.

[2] Look for files in `logo` in their name. It's an opportunity to practise with the `find` command.

As in the previous section, use this tool to find where the `platform_device_register()` function is declared, implemented and even used.

Board setup

Objective: setup communication with the board and configure the bootloader.

After this lab, you will be able to:

- Access the board through its serial line.
- Configure the U-boot bootloader and a tftp server on your workstation to download files through tftp.

Getting familiar with the board

Take some time to read about the board features and connectors:

https://docs.beagleboard.org/latest/boards/beagleplay/01-introduction.html

Don't hesitate to share your questions with the instructor.

Download technical documentation

We are going to download documents which we will need during our practical labs.

The first document to download is the datasheet for the TI AM62x SoC family, available on https://www.ti.com/lit/gpn/am625. This document will give us details about pin assignments.

Secondly, download the Technical Reference Manual (TRM) for the TI AM62x SoC family, available on https://www.ti.com/lit/pdf/spruiv7. This document is more than 15000 pages long! You will need it too during the practical labs.

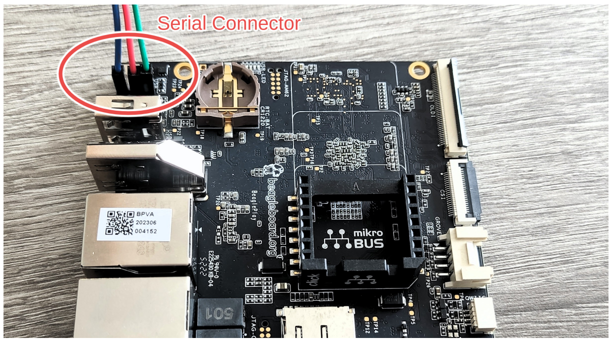
Last but not least, download the schematics for the BeaglePlay board:

https://openbeagle.org/beagleplay/beagleplay/-/blob/main/BeaglePlay_sch.pdf

Setting up serial communication with the board

The Beagle Play serial connector is a 3-pin header located right next to the board's USB-C port. Using your special USB to Serial adapter provided by your instructor, connect the ground wire (blue) to the pin labeled "G", the `TX` wire (red) to the pin labeled "RX" and the `RX` wire (green) to the pin labeled "TX" [3].

You always should make sure that you connect the `TX` pin of the cable to the `RX` pin of the board, and vice versa, whichever board and cables you use.



[3] See https://www.olimex.com/Products/Components/Cables/USB-Serial-Cable/USB-SERIAL-F/ for details about the USB to Serial adapter that we are using.

Once the USB to Serial connector is plugged in, a new serial port should appear: `/dev/ttyUSB0`. You can also see this device appear by looking at the output of `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/ttyUSB0`, you can also see that only `root` and users belonging to the `dialout` group have read and write access to this file. 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 *completely log out* from your session and log in again (no need to reboot). A workaround is to run `newgrp dialout`, but it is not global. You have to run it in each terminal.

Now, you can run `picocom -b 115200 /dev/ttyUSB0`, to start serial communication on `/dev/ttyUSB0`, with a baudrate of `115200`. If you wish to exit `picocom`, press `[Ctrl][a]` followed by `[Ctrl][x]`.

There should be nothing on the serial line so far, as the board is not powered up yet.

Remove any SD card from the Beagle Play, we will be booting from the board's eMMC.

It is now time to power up your board by plugging in the USB-C cable supplied by your instructor to your PC.

See what messages you get on the serial line. You should see U-boot start.

Bootloader interaction

Reset your board. Press the space bar in the `picocom` terminal to stop the U-boot countdown. You should then see the U-Boot prompt:

```
=>
```

You can now use U-Boot. Run the `help` command to see the available commands.

Type the `help saveenv` command to make sure that the `saveenv` command exists. We use it in these labs to save your U-Boot environment settings to the eMMC.

```
env default -f -a
saveenv
```

If you don't have this U-Boot prompt, it's probably because you are doing these labs on your own (i.e. without participating to a Bootlin course), you'll have to flash the eMMC from recovery mode.

Flashing the bootloader from recovery mode

This section can be skipped if you already have a U-Boot prompt.

Follow the installation instructions at https://github.com/bootlin/snagboot to install snagboot.

Make sure to install the snagboot udev rules as specified in the instructions:

```
$ snagrecover --udev > 50-snagboot.rules
$ sudo cp 50-snagboot.rules /etc/udev/rules.d/
$ sudo udevadm control --reload-rules
$ sudo udevadm trigger
```

Go to the bootloader folder where boot images have been precompiled for you:

```
$ cd ~/linux-kernel-beagleplay-labs/bootloader
```


First uncompress the `boot.img.gz` file in the current directory.

Put the Beagle Play into recovery mode by unplugging and replugging the USB-C power cable while pressing the USR button. Please beware that a warm reset performed with the reset button won't work!

Wait about 10 seconds for the board's recovery mode to start, then check that the following USB device is present:

```
$ lsusb | grep AM62x
Bus 003 Device 021: ID 0451:6165 Texas Instruments, Inc. AM62x DFU
```

Run snagrecover:

```
# snagrecover -s am625 \
          -F "{'tiboot3': {'path': 'tiboot3.bin'}}" \
  -F "{'tispl': {'path': 'tispl.bin'}}" \
  -F "{'u-boot': {'path': 'u-boot.img'}}"
```

After snagrecover is done, you should get a U-Boot console on the serial port. Using this console, start DFU:

```
=> setenv dfu_alt_info '0=system raw 0 524288000;1=system raw 0 307133 mmcpart 1'
=> dfu 0 mmc 0
```

Don't be upset by the following message, it's expected...

```
generic_phy_get_bulk : no phys property
```

Then from the host, flash the image:

```
$ snagflash -P dfu -p 0451:6165 -D 1:tiboot3_emmc.bin -D 0:boot.img
```

Once snagflash is done writing the boot image to eMMC, reset the board. You should get a U-Boot prompt!

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.

For these next steps, make sure that your beagleplay board is directly connected to your host PC through its ethernet port. 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 192.168.1.100
=> setenv serverip 192.168.1.1
```

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`[4]. 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 NetWork Manager'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 192.168.1.1/24
```

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 0x80000000 textfile.txt
```

The `tftp` command should have downloaded the `textfile.txt` file from your development workstation into the board's memory at location `0x80000000`[5].

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

```
=> md 0x80000000
```

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

[5]This location is part of the board SDRAM. If you want to check where this value comes from, you can check the SoC datasheet at https://www.ti.com/lit/ug/spruiv7a/spruiv7a.pdf. It's a big document (more than 12,000 pages). In this document, look for `Memory Map` and you will find the SoC memory map. You will see that the address range for the memory controller (*DDR16SS0_SDRAM*) starts at the address we are looking for. You can also try with other values in the RAM address range.

Kernel compiling and booting

Objective: compile and boot a kernel for your board, booting on a directory on your workstation shared by NFS.

After this lab, you will be able to:

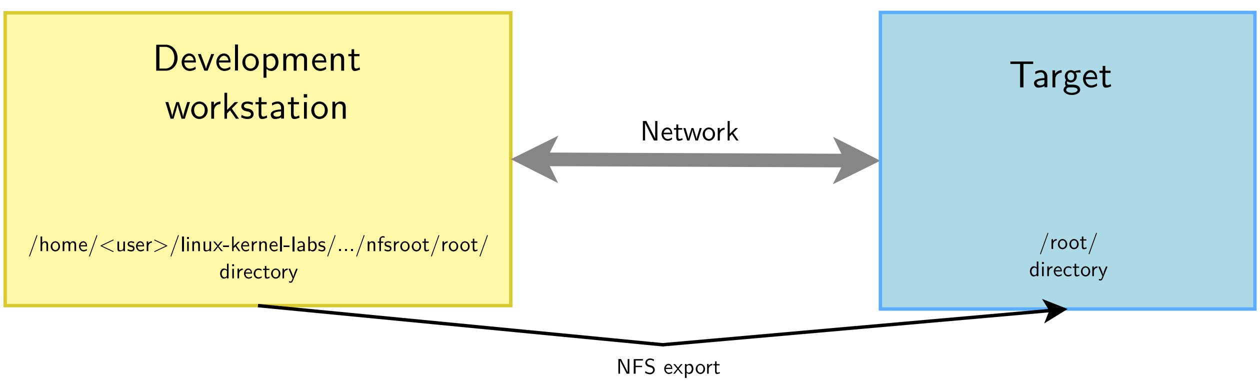
- Cross-compile the Linux kernel for the ARM platform.
- Boot this kernel on an NFS root filesystem, which is somewhere on your development workstation[6].

Lab implementation

While developing a kernel module, the developer wants to change the source code, compile and test the new kernel module very frequently. While writing and compiling the kernel module is done on the development workstation, the test of the kernel module usually has to be done on the target, since it might interact with hardware specific to the target.

However, flashing the root filesystem on the target for every test is time-consuming and would use the flash chip needlessly.

Fortunately, it is possible to set up networking between the development workstation and the target. Then, workstation files can be accessed through the network by the target, using NFS.



Setup

Go to the `$HOME/linux-kernel-beagleplay-labs/src/linux` directory.

Install packages needed for configuring, compiling and booting the kernel for your board:

```
sudo apt install libssl-dev bison flex
```

Cross-compiling toolchain setup

We are going to install a cross-compiling toolchain provided by Ubuntu:

```
sudo apt install gcc-aarch64-linux-gnu
```

Now find out the path and name of the cross-compiler executable by looking at the contents of the package:

```
dpkg -L gcc-aarch64-linux-gnu
```

[6] NFS root filesystems are particularly useful to compile modules on your host, and make them directly visible on the target. You no longer have to update the root filesystem by hand and transfer it to the target (requiring a shutdown and reboot).

Kernel configuration

Set the ARCH and CROSS_COMPILE definitions for the arm64 platform and your cross-compiler.

There is only one default configuration for arm64 platforms, which is called defconfig. Apply this configuration, 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 the SoCs except for the Texas Instruments Inc. K3 multicore SoC architecture.
- Disable CONFIG_DRM, which will skip support for many display controller and GPU drivers.
- Disable CONFIG_LEDS_GPIO, we will reenable it later as a demonstration.

Make sure that this configuration has CONFIG_ROOT_NFS=y (support booting on an NFS exported root directory).

Kernel compiling

Compile your kernel and generate the Device Tree Binaries (DTBs) (running 8 compile jobs in parallel):

```
make -j 8
```

Now, copy the Image.gz and k3-am625-beagleplay.dtb files to the TFTP server home directory (as specified in /etc/default/tftpd-hpa).

Setting up the NFS server

Install the NFS server by installing the nfs-kernel-server package. Once installed, edit the /etc/exports file as root to add the following lines, assuming that the IP address of your board will be 192.168.1.100:

```
/home/<user>/linux-kernel-beagleplay-labs/modules/nfsroot 192.168.1.100(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, restart the NFS server:

```
sudo exportfs -r
```

If there is any error message, this usually means that there was a syntax error in the /etc/exports file. Don't proceed until these errors disappear.

Boot the system

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

Do this by setting U-boot's bootargs environment variable (all in just one line):

```
setenv bootargs root=/dev/nfs rw ip=192.168.1.100:::::eth0 console=ttyS2,115200n8
  nfsroot=192.168.1.1:/home/<user>/linux-kernel-beagleplay-labs/modules/nfsroot,nfsvers=3,tcp
```

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

Now save this definition:

```
saveenv
```

If you later want to make changes to this setting, you can type the below command in U-boot:


```
editenv bootargs
```

Now, download the kernel image through `tftp`:

```
tftp 0x80000000 Image.gz
```

You'll also need to download the device tree blob:

```
tftp 0x83000000 k3-am625-beagleplay.dtb
```

Now, boot your kernel:

```
booti 0x80000000 - 0x83000000
```

This last command should show you an error message of this type:

```
kernel_comp_addr_r or kernel_comp_size is not provided!
```

This is because the boot image that we use, `Image.gz`, is compressed, and therfore, needs to be uncompressed by U-Boot before continue booting. To do so U-Boot needs to know the maximum size of the uncompressed image and where to store it.

If you look at the size of the uncompressed kernel (`Image` file), you can estimate that 32 MB (0x2000000) is a reasonable upper bound for the size of the uncompressed kernel, even with a more exhaustive configuration.

This gives us,

```
=> setenv kernel_comp_addr_r 0x85000000
=> setenv kernel_comp_size 0x2000000
=> saveenv
```

Now you can retry the `booti` command and see the kernel be uncompressed and then loaded.

If everything goes right, you should reach a login prompt (user: `root`, password: `root`). Otherwise, check your setup and ask your instructor for support if you are stuck.

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`.

Checking the kernel version

It's often a good idea to make sure you booted the right kernel. By mistake, you could have booted a kernel previously stored in flash (typically through a default boot command in U-Boot), or forgotten to update the kernel image in the TFTP server home directory.

This could explain some unexpected behavior.

There are two ways of checking your kernel version:

- By looking at the first kernel messages
- By running the `uname -a` command after booting Linux.

In both cases, you will not only know the kernel version, but also the date when the kernel was compiled and the name of the user who did it.

Similarly, you can also check the command line actually received by the kernel, either by looking at the first boot messages, or once you have reached a command line shell, by running `cat /proc/cmdline`.

Automate the boot process

To avoid typing the same U-boot commands over and over again each time you power on or reset your board, you can use U-Boot's `bootcmd` environment variable:

```
setenv bootcmd 'tftp 0x80000000 Image.gz; tftp 0x83000000 k3-am625-beagleplay.dtb; booti 0x80000000 - 0x83000000'
saveenv
```

Don't hesitate to change it according to your exact needs.

We could also copy the `Image.gz` file to the eMMC flash and avoid downloading it over and over again. However, detailed bootloader usage is outside of the scope of this course. See our Embedded Linux system development course and its on-line materials for details.

Save your kernel configuration

Now that you have a working (and satisfying) kernel configuration, you can save it under the `configs` folder:

```
make savedefconfig
cp defconfig arch/arm64/configs/beagleplay_defconfig
```

So if you later overwrite the `.config` file inadvertently, you can just get back to a working configuration by running:

```
make beagleplay_defconfig
```


Writing modules

Objective: create a simple kernel module

After this lab, you will be able to:

- Compile and test standalone kernel modules, which code is outside of the main Linux sources.
- Write a kernel module with several capabilities, including module parameters.
- Access kernel internals from your module.
- Set up the environment to compile it.
- Create a kernel patch.

Setup

Go to the `~/linux-kernel-beagleplay-labs/modules/nfsroot/root/hello` directory. Boot your board if needed.

Writing a module

Look at the contents of the current directory. All the files you generate there will also be visible from the target. That's great to load modules!

Add C code to the `hello_version.c` file, to implement a module which displays this kind of message when loaded:

```
Hello World. You are currently using Linux <version>.
```

... and displays a goodbye message when unloaded.

Suggestion: you can look for files in kernel sources which contain `version` in their name, and see what they do.

You may just start with a module that displays a hello message, and add version information later.

Caution: you must use a kernel variable or function to get version information, and not just the value of a C macro. Otherwise, you will only get the version of the kernel you used to build the module.

Building your module

The current directory contains a `Makefile` file, which lets you build modules outside a kernel source tree. Compile your module.

Testing your module

Load your new module file on the target. Check that it works as expected. Until this, unload it, modify its code, compile and load it again as many times as needed.

Run a command to check that your module is on the list of loaded modules. Now, try to get the list of loaded modules with only the `cat` command.

Adding a parameter to your module

Add a `who` parameter to your module. Your module will say `Hello <who>` instead of `Hello World`.

Compile and test your module by checking that it takes the `who` parameter into account when you load it.

Adding time information

Improve your module, so that when you unload it, it tells you how many seconds elapsed since you loaded it. You can use the `ktime_get_seconds()` function to achieve this.

You may search for other drivers in the kernel sources using the `ktime_get_seconds()` function. Looking for other examples always helps!

Following Linux coding standards

Your code should adhere to strict coding standards, if you want to have it one day merged in the mainline sources. One of the main reasons is code readability. If anyone used one's own style, given the number of contributors, reading kernel code would be very unpleasant.

Fortunately, the Linux kernel community provides you with a utility to find coding standards violations.

First install the `python3-ply` and `python3-git` packages.

Then run the `scripts/checkpatch.pl -h` command in the kernel sources, to find which options are available. Now, run:

```
~/linux-kernel-beagleplay-labs/src/linux/scripts/checkpatch.pl --file --no-tree hello_version.c
```

See how many violations are reported on your code, and fix your code until there are no errors left. If there are many indentation related errors, make sure you use a properly configured source code editor, according to the kernel coding style rules in `process/coding-style`.

Adding the hello_version module to the kernel sources

As we are going to make changes to the kernel sources, first create a special branch for such changes:

```
git checkout 6.7.bootlin
git checkout -b hello
```

Add your module sources to the `drivers/misc/` directory in your kernel sources. Of course, also modify kernel configuration and building files accordingly, so that you can select your module in `make xconfig` and have it compiled by the `make` command.

Run one of the kernel configuration interfaces and check that it shows your new driver lets you configure it as a module.

Run the `make` command and make sure that the code of your new driver is getting compiled.

Then, commit your changes in the current branch (try to choose an appropriate commit message):

```
cd ~/linux-kernel-beagleplay-labs/src/linux
git add <files>
git commit -as
```

- `git add` adds files to the next commit. It is mandatory to use for new files that should be added under version control.
- `git commit -a` creates a commit with all modified files that already under version control
- `git commit -s` adds a `Signed-off-by:` line to the commit message. All contributions to the Linux kernel must have such a line.

Create a kernel patch

You can be proud of your new module! To be able to share it with others, create a patch which adds your new files to the mainline kernel.

Creating a patch with `git` is extremely easy! You just generate it from the commits between your branch and another branch, usually the one you started from:


```
git format-patch 6.7.bootlin
```

Have a look at the generated file. You can see that its name reused the commit message.

If you want to change the last commit message at this stage, you can run:

```
git commit --amend
```

And run `git format-patch` again.

Describing Hardware Devices

Objective: learn how to describe hardware devices.

Goals

Now that we covered the Device Tree theory, we can explore the list of existing devices and make new ones available. In particular, we will create a custom Device Tree to describe the few extensions we will make to our BeaglePlay board.

Setup

Go to the `~/linux-kernel-beagleplay-labs/src/linux` directory. Check out the `6.7.bootlin` branch.

Now create a new `beagleplay-custom` branch starting from this branch, for your upcoming Device Tree changes on the Beagle Play.

Download a useful document sharing useful details about the Nunchuk and its connector:
https://bootlin.com/labs/doc/nunchuk.pdf

Create a custom device tree

To let the Linux kernel handle a new device, we need to add a description of this device in the board device tree.

As the Beagle Play device tree is provided by the kernel community, and will continue to evolve on its own, we don't want to make changes directly to the device tree file for this board.

The easiest way to customize the board DTS is to create a new DTS file that includes the Beagle Play DTS, and adds its own definitions.

So, create a new `arch/arm64/boot/dts/ti/k3-am625-beagleplay-custom.dts` file in which you just include the regular board DTS file. We will add further definitions in the next sections.

```
// SPDX-License-Identifier: GPL-2.0
#include "k3-am625-beagleplay.dts"
```

Modify the `arch/arm64/boot/dts/ti/Makefile` file to add your custom Device Tree, and then have it compiled with (`make dtbs`). Now, copy the new DTB to the tftp server home directory, change the DTB file name in the U-Boot configuration[7], and boot the board.

Setting the board's model name

Modify the custom Device Tree file to override the model name for your system. Set the `model` property to `Training Beagle Play`. 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.

[7] Tip: you just need to run `editenv bootcmd` and `saveenv`.

Driving LEDs

The BeaglePlay features five user LEDs (LED_USR0, …, LED_USR4) in the corner near the USB-C port.

Start by looking at the different description files and look for a node that would be defining the LEDs.

The five LEDs are actually supposed to be triggered by a driver matching the compatible gpio-leds. This is a generic driver which acts on LEDs connected to GPIOs. But as you can observe, despite being part of the in-use Device Tree, the LEDs remain off. The reason for that is the absence of driver for this node: nothing actually drives the LEDs even if they are described. So you can start by recompiling your kernel with CONFIG_LEDS_GPIO=y.

You should now see USR_LED0 blink with the CPU activity, USR_LED1 staying on, and the others staying off. If you look at the bindings documents Documentation/devicetree/bindings/leds/common.yaml and Documentation/devicetree/bindings/leds/leds-gpio.yaml, you'll notice we can tweak the default-state in order to make the three inactive user LEDs bright.

You will need to modify a shared DTSI file in order to do that. But because we do not want to impact other boards also using that same DTSI file, we might instead add a label to the leds container node. We could then reference this new label in our custom DTS and overwrite the default-state property of each LED subnode.

Reboot the board using the new DTS and observe the LEDs default states change. If you look again at the common file defining the LEDs, they are actually all linked to a linux,default-trigger. The default state only applies until the trigger starts its activity.

USR_LED0 is a heartbeat which you can enable or disable with CONFIG_LEDS_TRIGGER_HEARTBEAT. USR_LED1 is triggered by disk activity.

Managing I2C buses and devices

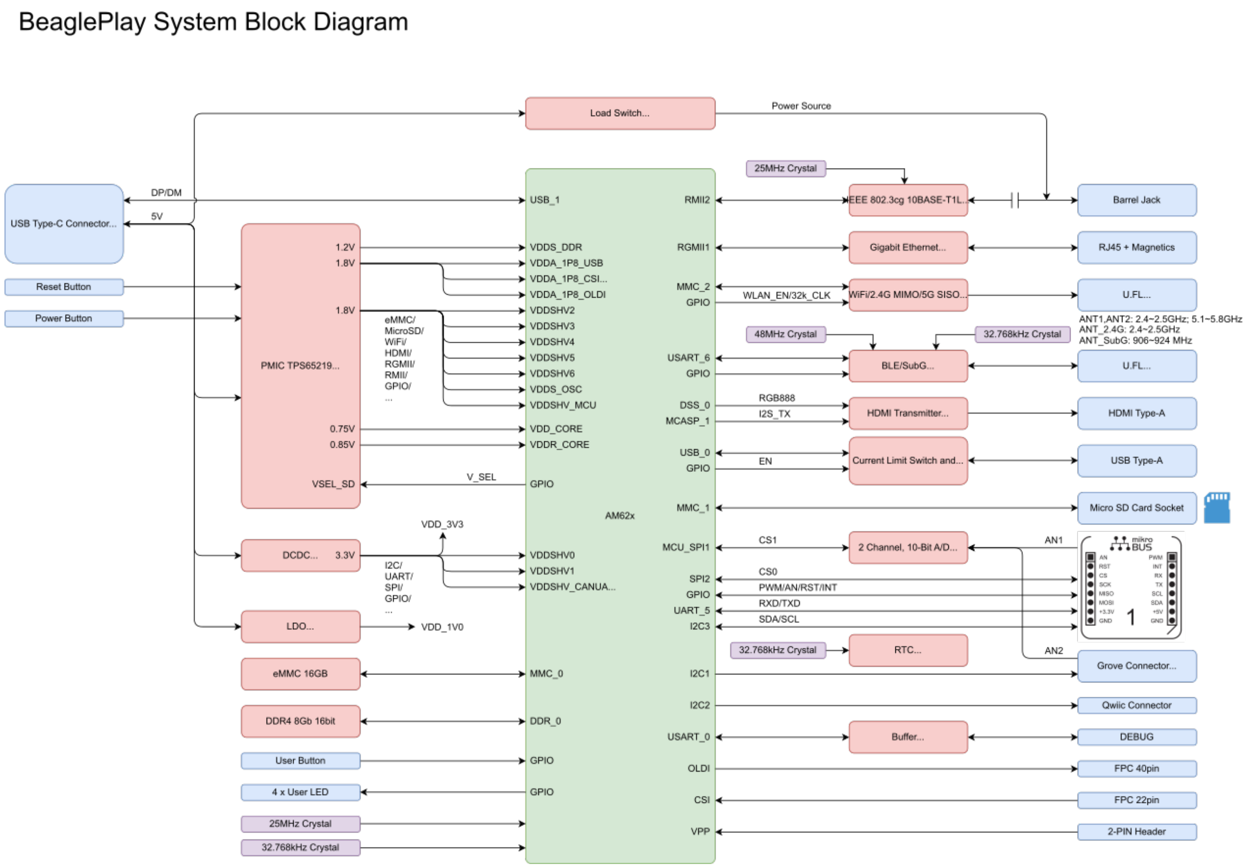
The next thing we want to do is connect an Nunchuk joystick to an I2C bus on our board. The I2C bus is very frequently used to connect all sorts of external devices. That's why we're covering it here.

Enabling an I2C bus

As shown on the below picture found on https://docs.beagleboard.org/latest/boards/beagleplay/03-design.html, the BeaglePlay has 3 I2C busses available through different connectors:

- I2C3: available on the mikroBUS connector
- I2C1: available on the Grove connector
- I2C2: available on the Qwiic connector

The AM62x SoC has three others I2C controllers, but they are not used on the BeaglePlay board. However because the default device-tree we are using enables all the I2C controllers except one, we expect the kernel to detect five I2C buses in total.



In this lab we will be using the I2C3 bus to connect the nunchuk because it is located on the mikroBUS connector and is easily accessible.

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

```
# i2cdetect -l
i2c-3 i2c OMAP I2C adapter I2C adapter
i2c-1 i2c OMAP I2C adapter I2C adapter
i2c-2 i2c OMAP I2C adapter I2C adapter
i2c-0 i2c OMAP I2C adapter I2C adapter
i2c-5 i2c OMAP I2C adapter I2C adapter
```

As the bus numbering scheme in Linux doesn't always match the one on the datasheets, let's check the base addresses of the registers of these controllers:

```
# ls -l /sys/bus/i2c/devices/i2c-*
lrwxrwxrwx 1 root root 0 Jan 1 02:02 /sys/bus/i2c/devices/i2c-0 -> ../../../devices/platform\
    /\
bus@f0000/20000000.i2c/i2c-0
lrwxrwxrwx 1 root root 0 Jan 1 02:02 /sys/bus/i2c/devices/i2c-1 -> ../../../devices/platform\
    /\
bus@f0000/20010000.i2c/i2c-1
lrwxrwxrwx 1 root root 0 Jan 1 02:02 /sys/bus/i2c/devices/i2c-2 -> ../../../devices/platform\
    /\
bus@f0000/20020000.i2c/i2c-2
lrwxrwxrwx 1 root root 0 Jan 1 02:02 /sys/bus/i2c/devices/i2c-3 -> ../../../devices/platform\
```



```
    /\
bus@f0000/20030000.i2c/i2c-3
lrwxrwxrwx 1 root root 0 Jan 1 02:02 /sys/bus/i2c/devices/i2c-5 -> ../../../devices/platform\
    /\
bus@f0000/bus@f0000:bus@4000000/4900000.i2c/i2c-5
```

Interpreting this output is not completely straightforward, but you can suppose that:

- I2C0 is at address `0x20000000`
- I2C1 is at address `0x20010000`
- I2C2 is at address `0x20020000`
- I2C3 is at address `0x20030000`
- I2C5 is at address `0x04900000`

Now let's double check the addressings by looking at the TI AM62x SoC datasheet, in the `Memory Map` section:

- I2C0 is indeed at address `0x20000000`
- I2C1 is indeed at address `0x20010000`
- I2C2 is indeed at address `0x20020000`
- I2C3 is indeed at address `0x20030000`
- I2C4 doesn't exist in the reference manual but corresponds to WKUP_I2C0 at address `0x2b200000`
- I2C5 doesn't exist in the reference manual but corresponds to MCU_I2C0 at address `0x04900000`

So luckily, the first 4 Linux I2C names correspond to the first 4 datasheet names.

Prepare the I2C device DT description

Before describing your nunchuk device, let's think about what will be needed:

- The device node should follow a standard pattern.

  The node name should be `joystick@addr`, the convention for node names is `<device-type>@<addr>`.

- We want to be able to fully identify the programming model.

  This is usually done using a unique compatible string. The compatible contains a vendor prefix and then a more specific string. We will use `nintendo,nunchuk`.

- We need to identify how to reach the device.

  This is the `reg` property and we should set it to the I2C address of the nunchuk. You will find the I2C slave address of the Nunchuk on the nunckuk document that we have downloaded earlier[8].

- (Optional) There are two types of nunchuks.

  There are white and black nunchuks, which don't expect the same initialization flow. We could imagine a boolean property named `nintendo,alternate-init` which will change the initialization logic. See the nunchuk pdf for details about the alternate flow.

Stopping here is sufficient as writing device-tree bindings is not strictly required to continue the labs, but if you feel comfortable you may want to write your own binding file, eg:

```
Documentation/devicetree/bindings/misc/nintendo,nunchuk.yaml
```

[8] This I2C slave address is enforced by the device itself. You can't change it.

Once you are confident with your bindings, you can even copy the examples from the `wrong-nunchuk-examples.yaml` (in the `nunchuk` labs folder) inside your bindings and verify they all pass/fail as expected!

```
make DT_SCHEMA_FILES=misc/nintendo,nunchuk.yaml dt_binding_check
```

Declare the Nunchuk device

As a child node to the `i2c3` bus, now declare an I2C device for the Nunchuk, following the above rules.

If you wrote an optional YAML binding, you can also double check your node:

```
make DT_SCHEMA_FILES=misc/nintendo,nunchuk.yaml dtbs_check
```

After updating the running Device Tree, explore `/sys/firmware/devicetree`, where every subdirectory corresponds to a DT node, and every file corresponds to a DT property. You can search for presence of the new `joystick` node:

```
# find /sys/firmware/devicetree -name "*joystick*"
/sys/firmware/devicetree/base/bus@f0000/i2c@20030000/joystick@52
```

You can also check the whole structure of the loaded Device Tree, using the Device Tree Compiler (`dtc`), which we put in the root filesystem:

```
# dtc -I fs /sys/firmware/devicetree/base/ > /tmp/dts
# grep -C10 nunchuk /tmp/dts
```

Once your new Device Tree seems correct, commit your changes. As you modified a shared file and a custom file, it is good practice to commit these changes in two different patches.

Configuring the pin muxing

Objective: learn how to declare and use a muxing state.

Goals

As part of the previous lab, we enabled an I2C controller and described a device plugged on the bus. In this lab we will cover how to ensure a proper communication between the two and be able to declare and use pinctrl settings.

Setup

Continue using the `beagleplay-custom` branch in the `~/linux-kernel-beagleplay-labs/src/linux` directory.

Probing the different busses

The Beagle Play device tree already correctly configures the pinmuxing state for the I2C3 bus. Before proceeding with this lab, we ask you to delete this pinmuxing configuration by adding two lines in your custom device tree:

```
 &main_i2c3 {
        status = "okay";
+       /delete-property/ pinctrl-0;
+       /delete-property/ pinctrl-names;
```

Reboot your board with these changes.

Now, let's use `i2cdetect`'s capability to probe a bus for devices. The I2C bus has no real discovery capability, but yet, the tool exploits a feature of the specification: when the master talks to a device, it starts by sending the target address on the bus and expects it to be acked by the relevant device. Iterating through all the possible addresses without sending anything after the address byte, looking for the presence of an Ack is what uses the tool to probe the devices. That is also why we get a warning when using it.

Let's start by probing 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 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: 50 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- 68 -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

We can see three devices on this internal bus:

- One at address `0x30`, indicated by `UU`, which means that there is a kernel driver actively driving this device.

- Two other devices at addresses `0x50` and `0x68`. We just know that they are currently not bound to a kernel driver.

Now try to probe I2C3 with `i2cdetect -r 3`.

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

Find pin muxing configuration information for i2c3

As you found in the previous lab, we now managed to have our nunchuk device enumerated on the `i2c3` bus.

However, to access the bus data and clock signals, we need to configure the pin muxing of the SoC.

If you open the Beagle Play hardware schematics and go to sheet 13, you'll see that the `I2C3_SCL` and `I2C3_SDA` signals are routed to pins A15 and B15 on the AM625 SoC.

Now open the AM625 datasheet (not the reference manual!) and go to table 6.1 in the "Pin Attributes" section. Search for pins A15 and B15 in this table, using the first column, not the second one.

Once you've found pins A15 and B15, you'll see that mux mode number 2 corresponds to the `I2C3_SCL` and `ISC3_SDA` signals.

In the third column, you'll find the addresses of the pad configuration registers for these pins. Register 0x000F41D0 configures A15 and register 0x000F41D4 configures B15.

We now know which registers we can write to to enable `i2c3` signals.

Multiplexing the I2C controller outputs correctly

Now that we know the register offsets, let's try to understand how they are used in existing code. Open the original device tree for the Beagle Play board and go to the `main_i2c3` node. You'll see a handle to a pinctrl node: `mikrobus_i2c_pins_default`. Look for this pinctrl node; you'll see the following description:

```
mikrobus_i2c_pins_default: mikrobus-i2c-default-pins {
                pinctrl-single,pins = <
                        AM62X_IOPAD(0x01d0, PIN_INPUT_PULLUP, 2) /* (A15) UART0_CTSn.I2C3_SCL */
                        AM62X_IOPAD(0x01d4, PIN_INPUT_PULLUP, 2) /* (B15) UART0_RTSn.I2C3_SDA */
                >;
        };
```

Here are details about the values:

- `0x01d0` and `0x01d4` are the offsets of the pad configuration registers to control muxing on the corresponding package pins. They correspond to the two register addresses that we previously found in the datasheet.
- Muxing mode 2, is set for both pins, which follows what we saw in the datasheet.
- `PIN_INPUT_PULLUP` puts the pin in pull-up mode (remember that our pins support both pull-up and pull-down). By design, an I2C line is never actively driven high, devices either pull the line low or let it floating. As we plug our device directly on the bus without more analog electronics, we need to enable the internal pull-up.

Now that pin muxing settings have been explained, you can remove the two `delete-property` lines that you added to your custom device tree.

Rebuild and update your DTB, then reboot the board. You should now be able to probe your bus:

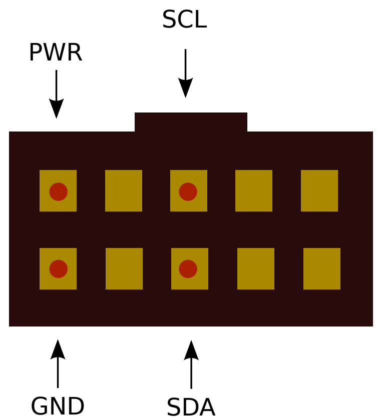
```
# i2cdetect -r 3
i2cdetect: WARNING! This program can confuse your I2C bus
Continue? [y/N] y
```

```
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00: -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

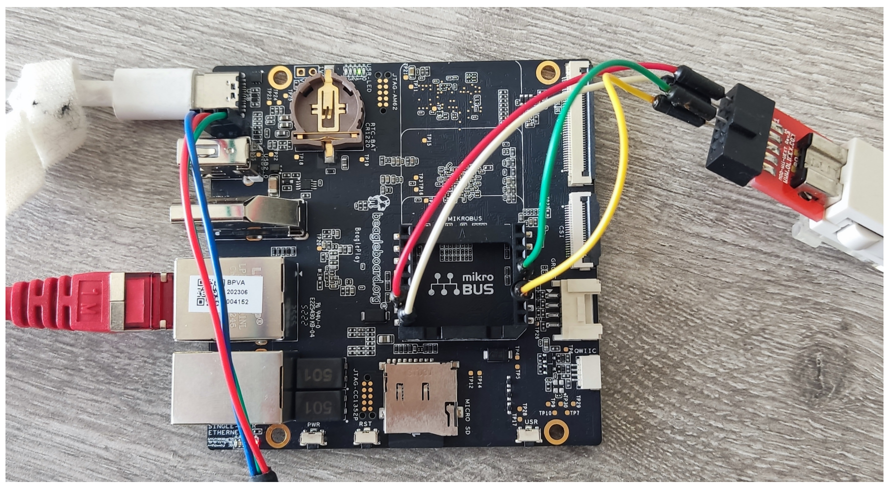
No devices are detected, because we did not wire the nunchuk yet.

Wiring the I2C device

Let's connect the Nunchuk provided by your instructor to the I2C3 bus on the board, using breadboard wires:



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



In this case, most of the labels on the Mikrobus connector correspond to the Nunchuk pin names. Just make sure that the PWR Nunchuk pin is connected to the 3.3V mikrobus pin.

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

```
# i2cdetect -r 3
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.

Using the I2C bus

Objective: Use the I2C bus to implement communication with the Nunchuk device

Goals

After this lab, you will be able to:

- Find your driver and device in `/sys`.
- Implement basic `probe()` and `remove()` driver functions and make sure that they are called when there is a device/driver match.
- Access I2C device registers through the bus.

Setup

Stay in the `~/linux-kernel-beagleplay-labs/src/linux` directory for kernel and DTB compiling (stay in the `beagleplay-custom` branch).

In a new terminal, go to `~/linux-kernel-beagleplay-labs/modules/nfsroot/root/nunchuk/`. This directory contains a Makefile and an almost empty `nunchuk.c` file.

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 `ttyS0` device file.
- 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.

Implement a basic I2C driver for the Nunchuk

It is now time to start writing the first building blocks of the I2C driver for our Nunchuk.

Relying on explanations given during the lectures, fill the nunchuk.c file to implement:

- probe() and remove() functions that will be called when a Nunchuk is found. For the moment, just put a call to pr_info() inside to confirm that these functions are called.
- Initialize a i2c_driver structure, and register the i2c driver using it. Make sure that you use a compatible property that matches the one in the Device Tree.

You can now compile your module and reboot your board, to boot with the updated DTB.

Driver tests

You can now load the /root/nunchuk/nunchuk.ko file. You need to check that the probe() function gets called then, and that the remove() function gets called too when you remove the module.

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

Also list /sys/bus/i2c/devices/. 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.

Device initialization

Now that we have checked that the probe() and remove() functions are called, remove the messages that you added to trace the execution of these functions.

The next step is to read the state of the nunchuk registers, to find out whether buttons are pressed or not, for example.

Before being able to read nunchuk registers, the first thing to do is to send initialization commands to it. That's also a nice way of making sure I2C communication works as expected.

In the probe routine (run every time a matching device is found):

1. Using the I2C raw API (i2c_master_send() and i2c_master_recv()), send two bytes to the device: 0xf0 and 0x55[9]. Make sure you check the return value of the function you're using. This could reveal

[9] The I2C messages to communicate with a wiimote extension are in the form: <i2c_address> <register> for reading and

communication issues. Using Elixir, find examples of how to handle failures properly using the same function.

(Optional) If you defined a nintendo,alternate-init property, you may want to check it's presence in the device tree using device_property_read_bool(), and derive the right initialization bytes from it.

2. Let the CPU wait for 1 ms by using the udelay() routine. Let's use Elixir again to find the right C headers to include...

   The Elixir results are a bit confusing here, because udelay() is defined in arch/<arch>/include/asm/delay.h files, but not in an include/linux/<file>.h> that is normally used in kernel code.

   However, look at include/linux/delay.h and you will see that it includes asm/delay.h which corresponds to the specific headers for the current architecture. So you need to include linux/delay.h.

   General rule: whenever the symbol you're looking for is defined in arch/<arch>/include/asm/<file>.h, you can include linux/<file>.h in your kernel code.

3. In the same way, send the 0xfb and 0x00 bytes now. This completes the nunchuk initialization.

Recompile and load the driver, and make sure you have no communication errors.

Read nunchuk registers

As the nunchuk does not feature any type of external signaling nor any internal bit to advertize a possible end-of-conversion status, the user is required to regularly poll the registers, each read triggering the next conversion. This leads to a specific situation: the first read triggers the first conversion but returns some data which can be considered garbage and safely discarded.

As a consequence, we will need to read the registers twice the first time!

To keep the code simple and readable, let's create a nunchuk_read_registers() function to read the registers once. In this function:

1. Start by putting a 10 ms delay by calling usleep_range(10000, 20000), guaranteed to sleep between 10 and 20 ms.[10] Such waiting time is needed to add time between the previous i2c operation and the next one.
2. Write 0x00 to the bus. That will allow us to read the device registers.
3. Add another 10 ms delay.
4. Read 6 bytes from the device, still using the I2C raw API. Check the return value as usual.

Reading the state of the nunchuk buttons

Back to the probe() function, call your new function twice.

After the second call, compute the states of the Z and C buttons, which can be found in the sixth byte that you read.

As explained on https://bootlin.com/labs/doc/nunchuk.pdf:

- bit 0 == 0 means that Z is pressed.

<i2c_address> <register> <value> for writing. The address, 0x52 is sent by the i2c framework so you only have to write the other bytes, the register address and if needed, the value you want to write. There are two ways to set up the communication. The first known way was with data encryption by writing 0x00 to register 0x40 of the nunchuk. With this way, you have to decrypt each byte you read from the nunchuk (not so hard but something you have to do). Unfortunately, such encryption doesn't work on third party nunchuks so you have to set up unencrypted communication by writing 0x55 to 0xf0 instead. This works across all brands of nunchuks (including Nintendo ones).

[10] That's better than using udelay() because it is not making an active wait, and instead lets the CPU run other tasks in the meantime. Moreover, this is better than using usleep() if your wait time is flexible because this function will try to group tasks wakeup rather than creating a specific timer to wake up that task. You'll find interesting details on how to sleep or wait in kernel code for specified durations in the kernel documentation: timers/timers-howto.

- `bit 0 == 1` means that `Z` is released.
- `bit 1 == 0` means that `C` is pressed.
- `bit 1 == 1` means that `C` is released.

Using boolean operators, write code that initializes a `zpressed` integer variable, which value is `1` when the `Z` button is pressed, and `0` otherwise. Create a similar `cpressed` variable for the `C` button[11].

The last thing is to test the states of these new variables at the end of the `probe()` function, and log a message to the console when one of the buttons is pressed.

Testing

Compile your module, and reload it. No button presses should be detected. Remove your module.

Now hold the `Z` button and reload and remove your module again:

```
insmod /root/nunchuk/nunchuk.ko; rmmod nunchuk
```

You should now see the message confirming that the driver found out that the `Z` button was held.

Do the same over and over again with various button states.

At this stage, we just made sure that we could read the state of the device registers through the I2C bus. Of course, loading and removing the module every time is not an acceptable way of accessing such data. We will give the driver a proper *input* interface in the next slides.

[11] You may use the `BIT()` macro, which will make your life easier. See Elixir for details.

Input interface

Objective: make the I2C device available to user space using the input subsystem.

After this lab, you will be able to:

- Expose device events to user space through an input interface, using the kernel based polling API for input devices (kernel space perspective).
- Handle registration and allocation failures in a clean way.
- Get more familiar with the usage of the input interface (user space perspective).

Add input event interface support to the kernel

This lab requires static input event interface support (`CONFIG_INPUT_EVDEV`) to be enabled in your kernel configuration, which is the default for the BeaglePlay. You can therefore continue to the next step.

Register an input interface

The first thing to do is to add an input device to the system. Here are the steps to do it:

- Declare a pointer to an `input_dev` structure in the `probe` routine. You can call it `input`. You can't use a global variable because your driver needs to be able to support multiple devices.
- Allocate such a structure in the same function, using the `devm_input_allocate_device()` function.
- Still in the `probe()` function, add the input device to the system by calling `input_register_device()`;

At this stage, first make sure that your module compiles well (add missing headers if needed).

When the module is loaded, you should get:

```
input: Unspecified device as /devices/platform/ocp/48000000.interconnect/48000000.interconnect:segment@0/4802
```

This `Unspecified device` string is actually expected as we haven't filled the fields of the `input` structure yet.

Handling probe failures

In the code that you created, make sure that you handle failure situations properly.

- Of course, test return values properly and log the causes of errors.
- In our case, we only allocated resources with `devm_` functions. Thanks to this, in case of failure, all the corresponding allocations are automatically released before destroying the `device` structure for each device. This greatly simplifies our error management code!

Implement the remove() function

In this function, we need to unregister and release the resources allocated and registered in the `probe()` routine.

Fortunately, in our case, there's nothing to do, as everything was allocated with `devm_` functions. Even the unregistration of the `input_dev` structure is automated.

Recompile your module, and load it and remove it multiple times, to make sure that everything is properly registered and automatically unregistered.

Add proper input device registration information

As explained before, we actually need to add more information to the input structure before registering it. So, add the below lines of code (still before device registration, of course):

```
input->name = "Wii Nunchuk";
input->id.bustype = BUS_I2C;

set_bit(EV_KEY, input->evbit);
set_bit(BTN_C, input->keybit);
set_bit(BTN_Z, input->keybit);
```

(Source code link: https://raw.githubusercontent.com/bootlin/training-materials/master/labs/kernel-i2c-input-interface/input-device-attributes.c)

Recompile and reload your driver. You should now see in the kernel log that the `Unspecified device` type is replaced by `Wii Nunchuk`.

Implement a polling routine

The nunchuk doesn't have interrupts to notify the I2C master that its state has changed. Therefore, the only way to access device data and detect changes is to regularly poll its registers.

So, it's time to implement a routine which will poll the nunchuk registers at a regular interval.

Create a `nunchuk_poll()` function with the right prototype (find it by looking at the definition of the `input_setup_polling()` function.)

In this function, you will have to read the nunchuk registers. However, as you can see, the prototype of the `poll_fn()` routine doesn't carry any information about the `i2c_client` structure you will need to communicate with the device. That's normal as the input subsystem is generic, and can't be bound to any specific bus.

This raises a very important aspect of the device model: the need to keep pointers between *physical* devices (devices as handled by the physical bus, I2C in our case) and *logical* devices (devices handled by subsystems, like the input subsystem in our case).

This way, when the `remove()` routine is called, we can find out which logical device to unregister (though that's not necessary in our case as logical device unregistration is automatic). Conversely, when we have an event on the logical side (such as running the polling function), we can find out which I2C device this corresponds to, to communicate with the hardware.

This need is typically implemented by creating a per device, *private* data structure to manage our device and implement such pointers between the physical and logical worlds.

Add the below global definition to your code:

```
struct nunchuk_dev {
        struct i2c_client *i2c_client;
};
```

Now, in your `probe()` routine, declare an instance of this structure:

```
struct nunchuk_dev *nunchuk;
```

Then allocate one such instead for each new device:

```
nunchuk = devm_kzalloc(&client->dev, sizeof(*nunchuk), GFP_KERNEL);
if (!nunchuk)
        return -ENOMEM;
```

(Source code link: https://raw.githubusercontent.com/bootlin/training-materials/master/labs/kernel-i2c-input-interface/private-data-alloc.c)

Note that we haven't seen kernel memory allocator routines and flags yet.

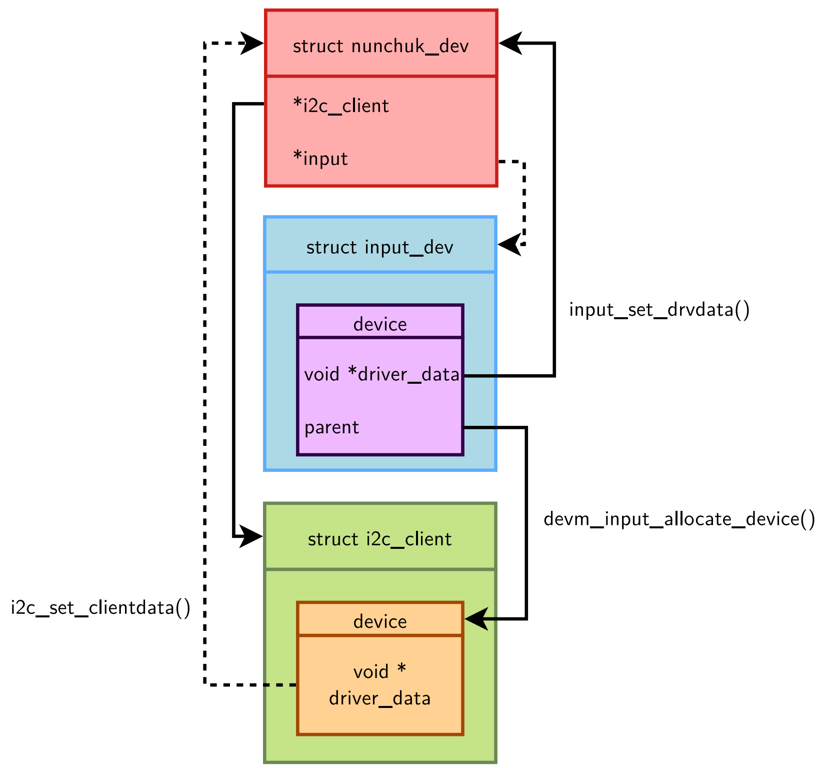
Also note that here there's no need to write an "out of memory" message to the kernel log. That's already done by the memory subsystem.

Now implement the pointers that we need:

```
nunchuk->i2c_client = client;
input_set_drvdata(input, nunchuk);
```

(Source code link: https://raw.githubusercontent.com/bootlin/training-materials/master/labs/kernel-i2c-input-interface/device-pointers.c)

Making the parallel with the lectures, here are the current links (the dotted lines show missing links that could be added in the future):



Make sure you add this code before registering the input device. You don't want to enable a device with incomplete information or when it is not completely initialized yet (there could be race conditions).

So, back to the `nunchuk_poll()` function, you will first need to retrieve the I2C physical device from the `input_dev` structure. That's where you will use your private `nunchuk` structure.

Now that you have a handle on the I2C physical device, you can move the code reading the nunchuk registers to this function. You can remove the double reading of the device state, as the polling function will make periodic reads anyway[12].

At the end of the polling routine, the last thing to do is post the events and notify the `input` core. Assuming that `input` is the name of the `input_dev` parameter of your polling routine:

```
input_report_key(input, BTN_Z, zpressed);
input_report_key(input, BTN_C, cpressed);
input_sync(input);
```

[12]During the move, you will have to handle communication errors in a slightly different way, as the `nunchuk_poll()` routine has a `void` type. When the function reading registers fails, you can use a `return;` statement instead of `return value;`

(Source code link: https://raw.githubusercontent.com/bootlin/training-materials/master/labs/kernel-i2c-input-interface/input-notification.c)

Now, back to the `probe()` function, the last thing to do is to declare the new polling function (see the slides if you forgot about the details) and specify a polling interval of 50 ms.

At this stage, also remove the debugging messages about the state of the buttons. You will get that information from the input interface.

You can now make sure that your code compiles and loads successfully.

Testing your input interface

Testing an input device is easy with the `evtest` application that is included in the root filesystem. Just run:

```
evtest
```

The application will show you all the available input devices, and will let you choose the one you are interested in (make sure you type a choice, `0` by default, and do not just type `[Enter]`). You can also type `evtest /dev/input/event0` right away. On some boards, the correct event device will be `event1`.

Press the various buttons and see that the corresponding events are reported by `evtest`.

Going further

Stopping here is sufficient, but if you complete your lab before the others, you can try to achieve the below challenges (in any order):

Supporting multiple devices

Modify the driver and Device Tree to support two nunchuks at the same time. You can borrow another nunchuk from the instructor or from a fellow participant.

Making sure that your driver does indeed support multiple devices at the same time is a good way to make sure it is implemented properly.

Use the nunchuk as a joystick in an ascii game

In this optional, challenge, you will extend the driver to expose the joystick part of the nunchuk, i.e. x and y coordinates.

We will use the *nInvaders* game, which is already present in your root filesystem.

Connect through SSH

nInvaders will not work very well over the serial port, so you will need to log to your system through `ssh` in an ordinary terminal:

```
ssh root@192.168.1.100
```

The password for the *root* user is `root`.

You can already play the `nInvaders` game with the keyboard!

Note: if you get the error `Error opening terminal: xterm-256color.` when running `nInvaders`, issue first the `export TERM=xterm` command.

Recompile your kernel

Recompile your kernel with support for the joystick interface (`CONFIG_INPUT_JOYDEV`).

Reboot to the new kernel.

Extend your driver

We are going to expose the joystick X and Y coordinates through the input device.

Add the below code to the `probe` routine:

```
set_bit(ABS_X, input->absbit);
set_bit(ABS_Y, input->absbit);
input_set_abs_params(input, ABS_X, 30, 220, 4, 8);
input_set_abs_params(input, ABS_Y, 40, 200, 4, 8);
```

(Source code link: https://raw.githubusercontent.com/bootlin/training-materials/master/labs/kernel-i2c-input-interface/declare-x-and-y.c)

See `input/input-programming` for details about the `input_set_abs_params()` function.

For the joystick to be usable by the application, you will also need to declare *classic* buttons:

```
/* Classic buttons */

set_bit(BTN_TL, input->keybit);
set_bit(BTN_SELECT, input->keybit);
set_bit(BTN_MODE, input->keybit);
set_bit(BTN_START, input->keybit);
set_bit(BTN_TR, input->keybit);
set_bit(BTN_TL2, input->keybit);
set_bit(BTN_B, input->keybit);
set_bit(BTN_Y, input->keybit);
set_bit(BTN_A, input->keybit);
set_bit(BTN_X, input->keybit);
set_bit(BTN_TR2, input->keybit);
```

(Source code link: https://raw.githubusercontent.com/bootlin/training-materials/master/labs/kernel-i2c-input-interface/declare-classic-buttons.c)

The next thing to do is to retrieve and report the joystick X and Y coordinates in the polling routine. This should be very straightforward. You will just need to go back to the nunchuk datasheet to find out which bytes contain the X and Y values.

Time to play

Recompile and reload your driver.

You can now directly play *nInvaders*, only with your nunchuk. You'll quickly find how to move your ship, how to shoot and how to pause the game.

Have fun!

Accessing I/O memory and ports

Objective: read / write data from / to a hardware device

Throughout the upcoming labs, we will implement a character driver allowing to write data to additional CPU serial ports available on the BeaglePlay, and to read data from them.

After this lab, you will be able to:

- Add UART devices to the board device tree.
- Access I/O registers to control the device and send first characters to it.

Setup

Go to your kernel source directory and continue working with the `beagleplay-custom` branch, this way we can keep the same custom Device Tree we already created, and build on top of it.

Add UART devices

In the following labs, we will be using UART5 and UART6, which are both routed to the Mikrobus connector.

Before developing a driver for these additional UARTs on the board, we need to find the corresponding Mikrobus pins.

First, open the Beagle Play hardware schematics and search for references to `UART5_RX`, `UART5_TX`, `UART6_RX` and `UART6_TX`. If you follow the UART5 signals, you'll see that they are already routed to the pins labeled "TX" and "RX" on the Mikrobus connector. For UART6, you'll see that the corresponding pins are used by the SPI2 bus by default, and that the pins for `UART6_TX` and `UART6_RX` are respectively routed to the MOSI (COPI) and MISO (CIPO) pins on the Mikrobus connector.

Go to the AM625 datasheet and find the pinmuxing settings for `UART6_TX` and `UART6_RX`.

The pinmuxing configuration is already done for UART5. For UART6, you'll have to add a pinmuxing section using the information you found in the datasheet. You'll also have to disable the `main_spi2` by setting its `status` property to `"disabled"` in your custom device tree.

```
&main_pmx0 {
	/* Pins COPI (TX) and CIPO (RX) of the mikrobus connector */
	main_uart6_pins: main_uart6_pins {
		pinctrl-single,pins = <
			AM62X_IOPAD(0x0198, PIN_INPUT_PULLUP, 3) /* (A19) MCASP0_AXR2.UART6_TXD */
			AM62X_IOPAD(0x0194, PIN_INPUT_PULLUP, 3) /* (B19) MCASP0_AXR3.UART6_RXD */
		>;
	};
};

&main_spi2 {
	status = "disabled";
};
```

(Source code link: https://raw.githubusercontent.com/bootlin/training-materials/master/labs/kernel-serial-iomem-beagleplay/uarts-pinctrl.dts)

Using a new USB-serial cable with male connectors, provided by your instructor, connect your PC to UART5. The wire colors are the same as for the cable that you're using for the console.

Then, declare the corresponding devices:

```
&main_uart5 {
	compatible = "bootlin,serial";
```

```
};

&main_uart6 {
        compatible = "bootlin,serial";
        pinctrl-names = "default";
        pinctrl-0 = <&main_uart6_pins>;
};
```

(Source code link: https://raw.githubusercontent.com/bootlin/training-materials/master/labs/kernel-serial-iomem-beagleplay/uarts.dts)

This is a good example of how we can override definitions in the Device Tree. `uart5` and `uart6` are already enabled and muxed in arch/arm64/boot/dts/ti/k3-am625-beagleplay.dts. In the above code, we just override the `compatible` property to use our driver instead of using the default one.

Compile and update your DTB.

Operate a platform device driver

Go to the `~/linux-kernel-labs/modules/nfsroot/root/serial/` directory. You will find a `serial.c` file which already provides a platform driver skeleton.

Add the code needed to match the driver with the devices which you have just declared in the device tree.

Compile your module and load it on your target. Check the kernel log messages, that should confirm that the `probe()` routine was called[13].

Create a device private structure

In the same way as in the nunchuk lab, we now need to create a structure that will hold device specific information and help keeping pointers between logical and physical devices.

As the first thing to store will be the base virtual address for each device, let's declare this structure as follows:

```
struct serial_dev {
        void __iomem *regs;
};
```

The first thing to do is allocate such a structure at the beginning of the `probe()` routine. Let's do it with the `devm_kzalloc()` function again as in the previous lab. Again, resource deallocation is automatically taken care of when we use the `devm_` functions.

So, add the below line to your code:

```
struct serial_dev *serial;
...
serial = devm_kzalloc(&pdev->dev, sizeof(*serial), GFP_KERNEL);
if (!serial)
        return -ENOMEM;
```

Get a base virtual address for your device registers

You can now get a virtual address for your device's base physical address, by calling:

```
serial->regs = devm_platform_ioremap_resource(pdev, 0);
if (IS_ERR(serial->regs))
        return PTR_ERR(serial->regs);
```

[13] Don't be surprised if the `probe()` routine is actually called twice! That's because we have declared two devices. Even if we only connect a serial-to-USB dongle to one of them, both of them are ready to be used!

What's nice is that you won't ever have to release this resource, neither in the `remove()` routine, nor if there are failures in subsequent steps of the `probe()` routine.

Make sure that your updated driver compiles, loads and unloads well.

Device initialization

Now that we have a virtual address to access registers, we are ready to configure a few registers which will allow us to enable the UART devices. Of course, this will be done in the `probe()` routine.

Accessing device registers

As we will have multiple registers to read, create a `reg_read()` routine, returning a `u32` value, and taking a `serial` pointer to a `serial_dev` structure and an `unsigned int` register offset.

Your prototype should look like:

```
static u32 reg_read(struct serial_dev *serial, unsigned int reg);
```

In this function, read from a 32 bits register at the base virtual address for the device, plus the register offset multiplied by 4.

All the UART register offsets have standardized values, shared between several types of serial drivers (see `include/uapi/linux/serial_reg.h`). This explains why they are not completely ready to use and we have to multiply them by 4 for K3 SoCs.

Create a similar `reg_write()` routine, writing an `int` value at a given register offset (don't forget to multiply it by 4) from the device base virtual address. The following code samples are using the `writel()` convention of passing the value first, then the offset. Your prototype should look like:

```
static void reg_write(struct serial_dev *serial, u32 val, unsigned int reg);
```

In the next sections, we will tell you what register offsets to use to drive the hardware.

Power management initialization

Add the below lines to the probe function:

```
pm_runtime_enable(&pdev->dev);
pm_runtime_get_sync(&pdev->dev);
```

And add the below line to the `remove()` routine:

```
pm_runtime_disable(&pdev->dev);
```

Line and baud rate configuration

After these lines, let's add code to initialize the line and configure the baud rate. This shows how to get a special property from the device tree, in this case `clock-frequency`:

```
/* Configure the baud rate to 115200 */
clk = devm_clk_get(&pdev->dev, NULL);
if (IS_ERR(clk)) {
        ret = PTR_ERR(clk);
        goto disable_runtime_pm;
}

uartclk = clk_get_rate(clk);

baud_divisor = uartclk / 16 / 115200;
reg_write(serial, 0x07, UART_OMAP_MDR1);
```

```
reg_write(serial, 0x00, UART_LCR);
reg_write(serial, UART_LCR_DLAB, UART_LCR);
reg_write(serial, baud_divisor & 0xff, UART_DLL);
reg_write(serial, (baud_divisor >> 8) & 0xff, UART_DLM);
reg_write(serial, UART_LCR_WLEN8, UART_LCR);
reg_write(serial, 0x00, UART_OMAP_MDR1);
```

(Source code link: https://raw.githubusercontent.com/bootlin/training-materials/master/labs/kernel-serial-iomem-beagleplay/uart-line-init.c)

Declare `baud_divisor` and `uartclk` as `unsigned int`.

FIFOs reset

The last thing to do is to reset the FIFOs:

```
/* Clear UART FIFOs */
reg_write(serial, UART_FCR_CLEAR_RCVR | UART_FCR_CLEAR_XMIT, UART_FCR);
```

(Source code link: https://raw.githubusercontent.com/bootlin/training-materials/master/labs/kernel-serial-iomem-beagleplay/uart-line-reset.c)

We are now ready to transmit characters over the serial ports!

If you have a bit of spare time, you can look at section 12.2.4 of the AM62x TRM for details about how to use the UART ports, to understand better what we are doing here.

Standalone write routine

Implement a C routine taking a pointer to a `serial_dev` structure and one character as parameters, and writing this character to the serial port, using the following steps:

1. Wait until the `UART_LSR_THRE` bit gets set in the `UART_LSR` register. You can busy-wait for this condition to happen. In the busy-wait loop, you can call the `cpu_relax()` kernel function to ensure the compiler won't optimise away this loop.
2. Write the character to the `UART_TX` register.

Add a call to this routine from your module `probe()` function, and recompile your module.

Open a new `picocom` instance on your new serial port (not the serial console):

```
picocom -b 115200 /dev/ttyUSB1
```

Load your module on the target. You should see the corresponding character in the new `picocom` instance, showing what was written to UART5.

You can also check that you also get the same character on UART6 (just connect to the UART6 pins instead of the UART5 ones).

Driver sanity check

Remove your module and try to load it again. If the second attempt to load the module fails, it is probably because your driver doesn't properly free the resources it allocated or registered, either at module exit time, or after a failure during the module `probe()` function. Check and fix your module code if you have such problems.

Output-only misc driver

Objective: implement the write part of a misc *driver*

After this lab, you will be able to:

- Write a simple misc driver, allowing to write data to the serial ports of your Beaglebone.
- Write simple `file_operations` functions for a device, including `ioctl` controls.
- Copy data from user memory space to kernel memory space and eventually to the device.
- You will practice kernel standard error codes a little bit too.

You must have completed the previous lab to work on this one.

Misc driver registration

In the same way we added an input interface to our Nunchuk driver, it is now time to give an interface to our serial driver. As our needs are simple, we won't use the *Serial framework* provided by the Linux kernel, but will use the *Misc framework* to implement a simple character driver.

Let's start by adding the infrastructure to register a *misc* driver.

The first thing to do is to create:

- A `serial_write()` write file operation stub. See the slides or the code for the prototype to use. Just place a `return -EINVAL;` statement in the function body, to signal that there is something wrong with this function.
- Similarly, a `serial_read()` read file operation stub.
- A `file_operations` structure declaring these file operations.

The next step is to create a `miscdevice` structure and initialize it. However, we are facing the same usual constraint to handle multiple devices. Like in the Nunchuk driver, we have to add such a structure to our device specific private data structure:

```
struct serial_dev {
        void __iomem *regs;
        struct miscdevice miscdev;
};
```

To be able to access our private data structure in other parts of the driver, you need to attach it to the `pdev` structure using the `platform_set_drvdata()` function. Look for examples in the source code to find out how to do it.

Now, at the end of the `probe()` routine, when the device is fully ready to work, you can now initialize the `miscdevice` structure for each found device:

- To get an automatically assigned minor number.
- To specify a name for the device file in *devtmpfs*. We propose to use:

  ```
  struct resource *res;
  [...]
  res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
  /* Error handling */
  [...]
  ```

```
name = devm_kasprintf(&pdev->dev, GFP_KERNEL, "serial-%x", res->start);
```

`devm_kasprintf()` allocates a buffer and runs `kasprintf()` to fill its contents. `platform_get_resource()` is used to retrieve the device physical address from the device tree. A much simpler solution to get a unique name for the device file would have been to use `&pdev->name`, but this would create unusual names for device files (e.g. `48024000.serial`).

- To pass the `file_operations` structure that you defined.
- To set the `parent` pointer to the appropriate value.

See the lectures for details if needed!

The last things to do (at least to have a *misc* driver, even if its file operations are not ready yet), are to add the registration and deregistration routines. That's typically the time when you will need to access the `serial_dev` structure for each device from the `pdev` structure passed to the `remove()` routine.

Make sure that your driver compiles and loads well, and that you now see two new device files in `/dev`.

At this stage, make sure you can load and unload the driver multiple times. This should reveal registration and deregistration issues if there are any.

Check in `/sys/class/misc` for an entry corresponding to the registered devices, and within those directories, check what the `device` symbolic link is pointed to. Check the contents of the `dev` file as well, and compare it with the major/minor number of the device nodes created in `/dev` for your devices.

Implement the write() routine

Now, add code to your write function, to copy user data to the serial port, writing characters one by one.

The first thing to do is to retrieve the `serial_dev` structure from the `miscdevice` structure itself, accessible through the `private_data` field of the open file structure (`file`).

At the time we registered our *misc* device, we didn't keep any pointer to the `serial_dev` structure. However, as the `struct miscdevice` structure is accessible through `file->private_data`, and is a member of the `serial_dev` structure, we can use a magic macro to compute the address of the parent structure:

```
struct miscdevice *miscdev_ptr = file->private_data;
struct serial_dev *serial = container_of(miscdev_ptr, struct serial_dev, miscdev);
```

See https://radek.io/2012/11/10/magical-container_of-macro/ for interesting implementation details about this macro.

This wouldn't have been possible if the `struct miscdevice` structure was allocated separately and was just referred to by a pointer in `serial_dev`, instead of being a member of it.

Another possibility, but more complicated, would have been to access the `parent` device pointer in `struct miscdevice`, which then through the `platform_get_drvdata()` function would have given us access to the `serial_dev` structure containing the virtual address of the device. There are always multiple possibilities in kernel programming!

Now, add code that copies (in a secure way) each character from the user space buffer to the UART device.

Once done, compile and load your module. Test that your `write` function works properly by using:

```
echo "test" > /dev/serial-<...>
```

The `test` string should appear on the remote side (i.e. in the `picocom` process connected to one of the UARTS.

If it works, you can triumph and do a victory dance in front of the whole class!

Make sure that both UART devices work on the same way.

You'll quickly discover that newlines do not work properly. To fix this, when the user space application sends `"\n"`, you must send `"\n\r"` to the serial port[14].

Module reference counting

Start an application in the background that writes nothing to the UART:

```
cat > /dev/serial-<...> &
```

Now, with `lsmod`, look at the reference count of your module: it is still 0, even though an application has your device opened. This means that you can `rmmod` your module even when an application is using it, which is not correct and can quickly cause a kernel crash.

To fix this, we need to tell the kernel that our module is in charge of this character device. This is done by setting the `.owner` field of `struct file_operations` to the special value `THIS_MODULE`.

After changing this, make sure the reference counter of your module, visible through `lsmod`, gets incremented when you run an application that uses the UART.

Ioctl operations

We would like to maintain a count of the number of characters written through the serial port. In order to do this, register a counter variable in the main driver structure, and increment it when appropriate. Then, we need to implement two `unlocked_ioctl()` operations:

- `SERIAL_RESET_COUNTER`, which as its name says, will reset the counter to zero
- `SERIAL_GET_COUNTER`, which will return the current value of the counter in a variable passed by address.

Two test applications (in source format) are already available in the `root/serial/` NFS shared directory. They assume that `SERIAL_RESET_COUNTER` is ioctl operation `0` and that `SERIAL_GET_COUNTER` is ioctl operation `1`.

Compile them:

```
aarch64-linux-gnueabi-gcc -static -o serial-get-counter serial-get-counter.c
```

```
aarch64-linux-gnueabi-gcc -static -o serial-reset-counter serial-reset-counter.c
```

The new executables are then ready to run on your target. They take as argument the path to the device file corresponding to your UART.

[14]See https://en.wikipedia.org/wiki/Newline for details about the newline (\n) and carriage return (\r) characters

Sleeping and handling interrupts

Objective: learn how to register and implement a simple interrupt handler, and how to put a process to sleep and wake it up at a later point

During this lab, you will:

- Register an interrupt handler for the serial controller of the Beaglebone.
- Implement the read() operation of the serial port driver to put the process to sleep when no data are available.
- Implement the interrupt handler to wake-up the sleeping process waiting for received characters.
- Handle communication between the interrupt handler and the read() operation.

Setup

This lab is a continuation of the *Output-only misc driver lab.* Use the same kernel, environment and paths!

Register the handler

Declare an interrupt handler function stub. Then, in the module probe function, we need to register this handler, binding it to the right IRQ number.

Nowadays, Linux is using a virtual IRQ number that it derives from the hardware interrupt number. This virtual number is created through the `irqdomain` mechanism. The hardware IRQ number to use is found in the device tree.

First, retrieve the unique IRQ number to request:

```
int irq;
irq = platform_get_irq(pdev, 0);
```

Then, pass the interrupt number to `devm_request_irq()` along with the interrupt handler to register your interrupt in the kernel.

Then, in the interrupt handler, just print a message and return `IRQ_HANDLED` (to tell the kernel that we have handled the interrupt).

You'll also need to enable interrupts. To do so, in the `probe()` function, write `UART_IER_RDI` to the `UART_IER` register.

Compile and load your module. Send a character on the serial link (just type something in the corresponding `picocom` terminal, and look at the kernel logs: they are full of our message indicating that interrupts are occurring, even if we only sent one character! It shows you that interrupt handlers should do a little bit more when an interrupt occurs.

Enable and filter the interrupts

In fact, the hardware will replay the interrupt until you acknowledge it. Linux will only dispatch the interrupt event to the rightful handler, hoping that this handler will acknowledge it. What we experienced here is called an `interrupt flood`.

Now, in our interrupt handler, we want to acknowledge the interrupt. On the UART controllers that we drive, it's done simply by reading the contents of the `UART_RX` register, which holds the next character received. You can display the value you read to see that the driver will receive whatever character you sent.

Compile and load your driver. Have a look at the kernel messages. You should no longer be flooded with interrupt messages. In the kernel log, you should see the message of our interrupt handler. If not, check your code once again and ask your instructor for clarification!

Load and unload your driver multiple times, to make sure that there are no registration / deregistration issues.

Sleeping, waking up and communication

Now, we would like to implement the `read()` operation of our driver so that a user space application reading from our device can receive the characters from the serial port.

First, we need a communication mechanism between the interrupt handler and the `read()` operation. We will implement a very simple circular buffer. So let's add a device-specific buffer to our `serial_dev` structure.

Let's also add two integers that will contain the next location in the circular buffer that we can write to, and the next location we can read from:

```
#define SERIAL_BUFSIZE 16

struct serial_dev {
        void __iomem *regs;
        struct miscdevice miscdev;
        unsigned int counter
        char rx_buf[SERIAL_BUFSIZE];
        unsigned int buf_rd;
        unsigned int buf_wr;
};
```

In the interrupt handler, store the received character at location `buf_wr` in the circular buffer, and increment the value of `buf_wr`. If this value reaches `SERIAL_BUFSIZE`, reset it to zero.

In the `read()` operation, if the `buf_rd` value is different from the `buf_wr` value, it means that one character can be read from the circular buffer. So, read this character, store it in the user space buffer, update the `buf_rd` variable, and return to user space (we will only read one character at a time, even if the user space application requested more than one).

Now, what happens in our `read()` function if no character is available for reading (i.e., if `buf_wr` is equal to `buf_rd`)? We should put the process to sleep!

To do so, add a `wait_queue_head_t` wait queue to our `serial_dev` structure, named for example `wait`. In the `read()` function, keep things simple by directly using `wait_event_interruptible()` right from the start, to wait until `buf_wr` is different from `buf_rd`[15].

Last but not least, in the interrupt handler, after storing the received characters in the circular buffer, use `wake_up()` to wake up all processes waiting on the wait queue.

Compile and load your driver. Run `cat /dev/serial-<...>` on the target, and then in `picocom` on the development workstation side, type some characters. They should appear on the remote side if everything works correctly!

Don't be surprised if the keys you type in Picocom don't appear on the screen. This happens because they are not echoed back by the target.

[15] A single test in the `wait_event_interruptible()` function is sufficient. If the condition is met, you don't go to sleep and read one character right away. Otherwise, when you wake up, you can proceed to the reading part.

Locking

Objective: practice with basic locking primitives

During this lab, you will:

- Practice with locking primitives to implement exclusive access to the device.

Setup

Continue to work with the `serial` driver.

You need to have completed the previous two labs to perform this one.

On the kernel side, recompile it with the following option:

- `CONFIG_DEBUG_ATOMIC_SLEEP`: this will allow you to be (loudly) warned if a function which may sleep is called from atomic context, while sleeping is not allowed. Otherwise, in practice, if the function that may sleep does not need to, you might not notice it!

Adding appropriate locking

We have two shared resources in our driver:

- The buffer that allows to transfer the read data from the interrupt handler to the `read()` operation.
- The device itself. It might not be a good idea to mess with the device registers at the same time and in two different contexts.

Therefore, your job is to add a spinlock to the driver, and use it in the appropriate locations to prevent concurrent accesses to the shared buffer and to the device.

Please note that you don't have to prevent two processes from writing at the same time: this can happen and is a valid behavior. However, if two processes write data at the same time to the serial port, the serial controller should not get confused.

DMA

Objective: learn how to use the `dma-mapping` *API to handle DMA buffers and coherency, as well as the* `dmaengine` *API to deal with DMA controllers through a generic abstraction*

During this lab, you will:

- Setup streaming mappings with the `dma` API
- Configure a DMA controller with the `dmaengine` API
- Configure the hardware to trigger DMA transfers
- Wait for DMA completion

Setup

This lab is a continuation of all the previous *serial* labs. Use the same kernel, environment and paths!

Preparing the driver

We will use DMA in the write path. As we will receive data from userspace, we will need a bounce buffer, so we can create a second buffer named `tx_buf` of the same size as `rx_buf` in our `serial_dev` structure.

As we will also need the `resource` structure with the MMIO physical addresses from outside of the `->probe()`, it might be relevant to save the `resource` pointer used to derive the `miscdev` name into the `serial_dev` structure.

Finally, the device-model `struct device *` contained in the platform device will soon be very useful as well, so we can save it in our `struct serial_dev *` object.

Before going further, re-compile and test your driver.

The `serial_write` callback and `serial_fops` can now be renamed `serial_write_pio` and `serial_fops_pio`, while we will implement a new callback named `serial_write_dma` and a new set of file operations called `serial_fops_dma` which uses this callback for `.write` and keeps the same values for other fields. This new set of file operations should be used by default.

Let's now create two helpers supposed to initialize and cleanup our DMA setup. We will call `serial_init_dma()` right before registering the `misc` device. In the `->probe()` error path and in the remove callback, we will call `serial_cleanup_dma()`. Make sure that errors are handled correctly and returned to the caller. A special case should be handled when no DMA channel is available (with the `-ENODEV` code returned) in order to fallback to the the `serial_fops_pio` file operations.

Prepare the DMA controller

The AM62x UART controller is internally wired to a DMA controller named PKTDMA. So we will have to deal with the `dmaengine` API in order to prepare DMA transfers on the controller side. The idea of this API is to fully abstract the characteristics of the DMA controller.

In a complete driver we should probably use the helpers checking capabilities. Let's just skip this part and assume the two IPs are compatible and the addressing masks properly set to 32-bit.

The BeaglePlay device tree does not describe DMA channels for UART5 and UART6, so we will have to add the channels to our custom device tree. References to DMA channels in the device tree have the following form:

```
dmas = <[controller reference] [dma-cells parameters]>
```

Go to your kernel source tree and open the `Documentation/devicetree/bindings/dma/ti/k3-pktdma.yaml` file, which contains the bindings for the AM62x PKTDMA controller. Find the description of the `#dma-cells` property. You will see that two cells are required: the first one is the thread ID for the UART controller and the second one is the ASEL value for the channel.

In our case, the ASEL value is 0, so we only need the thread ID. This can be found in the TRM and on other TI documents online, but as it is quite difficult to find, we will give the values to you:

- `UART_5` has a TX thread ID of 0xc405
- `UART_6` has a TX thread ID of 0xc406

With this information in hand, we can add the description of the DMA channels in the device tree:

```
&main_uart5 {
        compatible = "bootlin,serial";
        dmas = <&main_pktdma 0xc405 0>;
        dma-names = "tx";
};

&main_uart6 {
        compatible = "bootlin,serial";
        pinctrl-names = "default";
        pinctrl-0 = <&main_uart6_pins>;
        dmas = <&main_pktdma 0xc406 0>;
        dma-names = "tx";
};
```

(Source code link: `https://raw.githubusercontent.com/bootlin/training-materials/master/labs/kernel-serial-dma-beagleplay/uarts-dma.dts`)

This channel will be used by all the `dmaengine` helpers, so better save it in our `serial_dev` structure.

```
struct serial_dev {
        ...
        struct dma_chan *txchan;
};
```

Don't forget to update your device tree and reboot your board!

We can now configure the DMA controller with details about the upcoming transfers:

- memory to device transfers
- the source will be memory, we will map buffers when they come, there is no particular constraint on this side
- the destination is the UART Tx FIFO, we will ask the DMA to transfer the bytes one after the other (hardware signaling already handles the internal "flow")
- we shall not use the UART Tx FIFO directly, to be generic we shall use `dma_map_resource()` first (and save it in `serial_dev` to be able to unmap it later)

```
struct dma_slave_config txconf = {};

serial->fifo_dma_addr = dma_map_resource(dev, serial->res->start + UART_TX * 4,
                                         4, DMA_TO_DEVICE, 0);
if (dma_mapping_error(dev, serial->fifo_dma_addr)) ...

txconf.direction = DMA_MEM_TO_DEV;
```



```
txconf.dst_addr_width = DMA_SLAVE_BUSWIDTH_1_BYTE;
txconf.dst_addr = serial->fifo_dma_addr;
ret = dmaengine_slave_config(serial->txchan, &txconf);
if (ret) ...
```

The cleanup helper should on its side call `dmaengine_terminate_sync()` just to be sure no transfer is ongoing, right before un-mapping the FIFO with `dma_unmap_resource()` and releasing the DMA channel with `dma_release_channel()`.

It is time to recompile your driver and see if any header is missing...

Prepare the UART controller

On its side, the UART controller must assert some signals to drive the DMA flow. We must enable the controlling logic on the Tx DMA channel, by enabling `DMACTL` in mode 3. We also configure the UART to transmit all the bytes as soon as they get in.

```
#define OMAP_UART_SCR_DMAMODE_CTL3 0x7
#define OMAP_UART_SCR_TX_TRIG_GRANU1 BIT(6)

/* Enable DMA */
reg_write(serial, OMAP_UART_SCR_DMAMODE_CTL3 | OMAP_UART_SCR_TX_TRIG_GRANU1,
          UART_OMAP_SCR);
```

Process user write requests

It is now time to deal with user buffers again.

Before doing anything in the `write` hook, we shall fill-in the `serial_dev` structure with:

- a `bool txongoing` flag to prevent concurrent uses of the same Tx DMA channel (would be possible by queuing new requests, but let's keep this implementation simple) while not holding any lock for the full duration of the operation.
- a `struct completion txcomplete` object to asynchronously inform the write thread that the DMA transaction is over (very much like we did with the `waitqueue` in the interrupt lab). This object shall be initialized with `init_completion(&serial->txcomplete)`.

```
struct serial_dev {
        ...
        struct dma_chan *txchan;
        bool txongoing;
        struct completion txcomplete;
};
```

In the write hook, we shall first check if the DMA channel has been properly retrieved. If not, we should definitely fallback to the PIO implementation.

Then, in order to simplify the code, we will no longer deal with concurrent operations. In order to safely serialize writes, we can start and end the write hook with something like:

```
/* Prevent concurrent Tx */
spin_lock_irqsave(&serial->lock, flags);
if (serial->txongoing) {
        spin_unlock_irqrestore(&serial->lock, flags);
        return -EBUSY;
}
serial->txongoing = true;
spin_unlock_irqrestore(&serial->lock, flags);
```

```
...

spin_lock_irqsave(&serial->lock, flags);
serial->txongoing = false;
spin_unlock_irqrestore(&serial->lock, flags);
```

The first step in this `->write()` hook is to use `serial->tx_buf` as bounce buffer by copying the user data using `copy_from_user()`. Let's handle up to `SERIAL_BUFSIZE` bytes at a time. One can use `min_t()` to derive the right amount of bytes to deal with.

Now we can remap the buffer. We have a single buffer so we can use `dma_map_single()`. The output value is a `dma_addr_t`. Save this value as we will reuse it. Also do not forget to check its validity with `dma_mapping_error()`.

We now have all the missing information compared to the `serial_init_dma` step, like the `dma_addr_t` of the buffer and its length. Let's create a descriptor filled with all the default information known by the DMA controller plus the additional details we can now provide:

```
struct dma_async_tx_descriptor *desc;

desc = dmaengine_prep_slave_single(serial->txchan, dma_addr,
                                   len, DMA_MEM_TO_DEV,
                                   DMA_PREP_INTERRUPT | DMA_CTRL_ACK);
if (!desc) ...
```

We can now use the returned descriptor to register a callback. This callback will just call `complete()` over the completion object. Which also means this completion object could be re-initialized while we register the callback, just in case.

The DMA transfer contained in the descriptor can now be queued into the DMA controller queue:

```
dma_cookie_t cookie;

cookie = dmaengine_submit(desc);
ret = dma_submit_error(cookie);
if (ret) ...
```

The transfer can be triggered. This is usually an operation that is only required on the DMA controller side, but remember here we also need to trigger it on the UART controller side:

```
dma_async_issue_pending(serial->txchan);
```

The transfer being asynchronous, it is finally required to wait for completion with one of the `wait_for_completion()` variants, and to call `dma_unmap_single()` right after it.

You can now test your driver. Try sending strings of various length and observing how the serial port's behavior changes.

Kernel debugging mechanisms and kernel crash analysis

Objective: Use kernel debugging mechanisms and analyze a kernel crash

In this lab, we will continue to work on the code of our serial driver.

dev_dbg() and dynamic debugging

Add a `dev_dbg()` call in the `write()` operation that shows each character being written (or its hexadecimal representation) and add a similar `dev_dbg()` call in your interrupt handler to show each character being received.

Check what happens with your module. Do you see the debugging messages that you added? Your kernel probably does not have `CONFIG_DYNAMIC_DEBUG` set and your driver is not compiled with `DEBUG` defined, so you shouldn't see any message.

Now, recompile your kernel with the following options:

- `CONFIG_DYNAMIC_DEBUG`: this will allow you to see debugging messages.
- `CONFIG_DEBUG_INFO`: this option will make it possible to see source code in disassembled kernel code. We will need it in a later part of this lab, but enabling it now will allow to avoid recompiling the whole kernel again.

Also recompile the kernel module to have it built against the updated kernel config in order to take in account the new enabled options.

Once this is done, in U-Boot, add `loglevel=8` to the kernel command line to get the debugging messages directly in the console (otherwise you will only see them in `dmesg`).

Now boot your updated kernel.

The dynamic debug feature can be configured using `debugfs`, so you'll have to mount the `debugfs` filesystem first. Then, after reading the dynamic debug documentation in the kernel sources, do the following things:

- List all available debug messages in the kernel.
- Enable all debugging messages of your serial module, and check that you indeed see these messages.
- Enable just one single debug message in your serial module, and check that you see just this message and not the other debug messages of your module.

Now, you have a good mechanism to keep many debug messages in your drivers and be able to selectively enable only some of them.

debugfs

After using `debugfs` for controlling the dynamic debug feature, let's add a new entry in this filesystem. Modify your driver to add:

- A directory called after a unique name per device in the `debugfs` filesystem.
- And file called `counter` inside this directory of the `debugfs` filesystem. This file should allow to see the contents of the `counter` variable of your module.

Recompile and reload your driver, and check that in `/sys/kernel/debug/<unique name>/counter` you can see the amount of characters that have been transmitted by your driver.

Kernel crash analysis

Setup

Go to the `~/linux-kernel-beagleplay-labs/modules/nfsroot/root/debugging/` directory.

Compile the `drvbroken` in this directory, and load it on your board. See it crashing in a nice way.

Analyzing the crash message

Analyze the crash message carefully. Knowing that on ARM, the `PC` register contains the location of the instruction being executed, find in which function does the crash happen, and what the function call stack is.

Using Elixir or the kernel source code, have a look at the definition of this function. This, with a careful review of the driver source code should probably be enough to help you understand and fix the issue.

Locating the exact line where the error happens

Even if you already found out which instruction caused the crash, it's useful to use information in the crash report.

If you look again, the report tells you at what offset in the function this happens. Let's disassemble the code for this function to understand exactly where the issue happened.

That's where we need a kernel compiled with `CONFIG_DEBUG_INFO` as we did at the beginning of this lab. This way, the kernel is compiled with `$(CROSSCOMPILE)gcc -g`, which keeps the source code inside the binaries.

You could disassemble the whole `vmlinux` file and work with the `PC` absolute address, but it is going to take a long time.

Instead, using Elixir, you'll find that the crash happens in an function defined in assembly, called by a function implemented in C. Find the `.c` source file where the C function is implemented.

In the kernel sources, you can then find and dissassemble the corresponding `.o` file:

```
aarch64-linux-gnu-objdump -DS file.o > file.S
```

Another way to do this is to use gdb-multiarch[16]:

```
sudo apt install gdb-multiarch
gdb-multiarch vmlinux
(gdb) set arch aarch64
(gdb) set gnutarget elf64-littlearm
(gdb) disassemble function_name
```

Then, in the disassembled code, find the start address of the function, and using an hexadecimal calculator, add the offset that was provided in the crash output. That's how you can find the exact assembly instruction where the crash occured, together with the C code it was compiled from. Looking at the addresses handled by this code, you can now guess what is wrong in the data passed to the stack of kernel functions called by the broken module.

A little understanding of assembly instructions on the architecture you are working on helps, but seeing the original C code should answer most questions.

Note that the same technique works if the error comes directly from the code of a module. Just dissassemble the `.o` file the `.ko` file was generated from.

[16] gdb-multiarch is a new package supporting multiple architectures at once. If you have a cross toolchain including gdb, you can also run arm-linux-gdb directly.

